Simplified scalable server architecture

Edgar Klerks

21-08-2012

Abstract

Streetking is build upon an heterogenous, scalable architecture, which has six different loosely coupled components. Five components are server side only. The sixth component is client side only.

The server side components are: The media storage service, the load-balancing proxy server, game logic application server, the authorization system and the database storage system. The client side pillar is the user interface. The main server language is Haskell, which has excellent multithreading capabilities comparable to Erlang.

In this document the design of the system will be described globally along with an analysis of it's weaknesses and strengths.

Contents

1 Introduction

This an overview of the system, because of the complexity of the system, the overview is a somewhat simplified (but correct) representation of the reality.All the components are actually composed out of small subsystems .

1.1 Choice of language and tools

1.1.1 backend

We use [Haskell] as server language together with the [Snap] framework . Haskell is mostly used at universities and financial institutes, it's relative unknown to the mainstream developer. Therefore we will describe the language somewhat more extensive than the other tools and languages.

We have also looked into PHP, but because the quirky features, performance issues and many security issues we decided against it. Perl was another candidate, but was too slow. C++ was not high level enough for a webapplication and relative unsafe. Node.js was a serious candidate, but was still in beta phase, when we started.

For the database we have choosen [PostgreSQL], which is the most advanced open source database engine. It rivals Oracle in features. MySQL was lacking features we need for our application.

For database pooling we use [PGPool II], which is the standard load balancer for PostgreSQL.

The operating system we use is [FreeBSD] 9, which is very stable and secure and has excellent concurrency support.

1.1.2 frontend

In the frontend we use javascript and jquery and a variant of actionscript in the 3D Renderer ([Unity 3D engine]). For the 3D renderer we also looked into WebGL, but found the perfomance disappointing. We examined Dart (the new web language from google), but wasn't flexible enough and has a rather painful syntax.

1.1.3 Haskell

Introduction Haskell is an designed language, which exists since 1990. It is a lazy purely functional compiled language with a very advanced and static strong type system (based on System-F). A purely functional language has its foundations in the typed lambda calculus. This means that function has a precise mathematical definition. This definition is roughly, that a function should always return the same result for the same parameters and may not modify the state of the program. This means it is more easy to reason about a function. This also means that the order of two independent functions execution doesn't

matter, the compiler is free to order the execution order in the resulting program. The program is essentially a set of mathematical equations, which can be freely transformed in more optimized code.

There is a catch. A function cannot have side-effects. A side-effect is a transformation of the state of the world in the program. For example changing a variable is a side effect, talking to the database is a side-effect and opening a socket is a side effect. Everything, which uses IO (Input/Output) is a side effect. Now one will wonder, how can one actually communicate with a haskell program, if it is not allowed to communicate?

Well, because it is allowable under strict conditions:

- The side effect should evaluate in the IO environment it cannot escape. It should be encapsulated from the pure part.
- The order of execution in the IO environment should be maintained by the compiler. (We don't want to close a file and then write to it)
- In the IO environment it should be possible to read a result of an IO computation.

When these conditions are satisfied, we still have a pure language, with a special IO value. -The runtime system will evaluate the IO value- This value is called an IO monad. A monad is a construct from category theory. A monad makes it possible to simulate side-effects and let one build it's own imperative language. See for more information about the details of [the IO monad].

Haskell has a full implementation of Software transactional memory ([STM]). This is a concurrency model, which doesn't suffer from classical concurrency pitfals such as deathlocks.

The typesystem A strong typesystem prevents the user to give a wrong parameter to a function. A function, which accepts an natural number cannot accept a string in a strongly typed system. Haskell has a strong static typesystem. This means it can catch type errors compile time. This reduces the amount of unit tests that are needed, because we don't need to test for unexpected types. This makes haskell very safe to use. A haskell function can be typed by the programmer or the compiler can infer it for the programmer. We choose to add the types ourself, because it helps document the code.

Advantages All this functionality leads to several advantages.

- Parallization of code is free.
- Concurrency is easy with STM.
- Haskell is perfomance wise between C++ and Java7 server.
- Smaller amount of unit tests.
- Code is self-documenting.

- The code is very modular.
- Complex code is easy maintable.
- Haskell is extensively used in research articles.

Disadvantages

- Some mathematical intuition is needed.
- Programmers are more difficult to find.
- Sometimes simple code in an empirative language is very involved in haskell.

The last point can be a problem, but there are more haskell programmers than jobs at the moment. Haskell also works as a quality-of-programmer filter, because it has a strong academic background and a steep learning curve.

1.1.4 Snap framework

The [Snap] framework is a HTTP server framework. We used it to build several custom HTTP servers. It is reliable and a lot faster than a apache/php setup [Snap benchmark]. Snap gives control about every aspect of the HTTP server, while it is very minimal.

1.1.5 Jquery

JQuery is compatible with all browsers. It is a general framework to build interactive user interfaces in a webbrowser. With little modification, it is possible to run the same code on mobile devices. (Through JQuery Mobile).

1.1.6 Javascript

Javascript is a browser language, which is used in all the browsers. This is the logical choice for web programming. Ideas in javascript nicely integrates with haskell, because both languages can be considered as a functional language.

2 Components

We have decided to create a multi-tier architecture. Every component is on it's own server or on several servers. Communication between components is done through protocols. This makes it relative easy to scale out the system. It is also easy to shatter a component even futher or create new components. The servers can communicate with each other through a peer-to-peer storage system and message system. The first is used to communicate public common information, the latter is used for targetted information.

2.1 Media server

The media storage server stores and caches all the images in the game. It uses etags and more traditional caching techniques to reduce the load of the server. Multiple media storage servers can be hooked to a proxy server. If no image is found, a default image will be served.

2.2 Proxy server

The proxy server is at the heart of the scalability. It transparently routes requests to the backend servers. The proxy server uses http pipelining and chunked encoding for efficient use of the connection. The proxy server makes use of the [Iteratee] library, which allows incremental parsing techniques to send the request as fast as possible to the backend servers. The proxy server starts sending the request as soon as the header has been parsed. The server facilitates security through authorization and balancing of workload.

2.2.1 Load balancing

The load balancing is a simple round robin algorithm. A server can attach and detach itself anytime, thus making dynamic growth of the cluster possible.

2.2.2 Authorization

The proxy server is the controlling part of the authorization system. It can allow or disallow a role to access a resource based on the permission table. [1]

It is connected to the overcoupling user information storage[2], which is also part of the authorization system.

2.3 Game logic application server

The game logic application server (game server) enforces the rules and logic of the game. It exposes a couple of game resources. [3] The resources update the state of the server and report ill defined behaviour back to the client. The server is pragmatically stateless. Gamestate is stored in database, but temporary information is kept in p2p storage. If the server crashes, no information is lost. A stateless architecture is a scalable architecture, because there is no shared state, which creates interdependence of paralell components. This means it is possible to run multiple instances of the game server at the same time.

The game server uses [JSON-RPC] over HTTP to communicate with the outside world.

[1] This is a text document, which describes the access rights for the roles. The permissions are modelled after CRUD

[2] See 2.4 for more information

[3] Resources as used in the HTTP protocol

2.4 Authorization system

The authorization is a complex system. It stores and distributes the shared user information in the system to all nodes. It is splitted up into three parts:

- Storage of security tokens and user information.
- Authorization of client via internal system or third party system.
- Make a statement about the access rights of the user.

These systems are implemented in different servers and form together a network over the servers. Authorization of the client is implemented in the game server. The proxy can make a statement about the access rights of the user.

2.4.1 peer-to-peer storage

The storage of security tokens is a distributed insert-only peer-to-peer storage network, which communicates over a network protocol. We have used [ZeroMQ] 3.1 as backbone for the implementation. This solution is faster than communicating trough the database. The peer-to-peer nodes can be divided in three types, *routing nodes, query nodes and insert nodes*. Routing nodes only store and distribute information, but don't have access to the outside world. These are implemented in nodes, which don't need the login information. The query nodes can request and store information. These nodes are implemented in the proxy servers. Insert nodes accept new information (login information) and provide it to the routing nodes for request. These are erroneously (see 2.3) implemented in the game logic application servers.

Every peer-to-peer node store its current information on disk. The network handles connections asynchronously, a client may do multiple queries over one connection.

We have also looked into existing NoSQL solutions, but found the approach to heavy weight.

2.5 Database storage

As database storage engine we use [PostgreSQL] 9.1. We have looked into NoSQL solutions, but the game data has strong relational properties[4], thus making this solution unusable. The game logic application server communicates with the database through [PGPool II], which is a load balancer with a fail safe strategy build in. PGPool II behaves identical as a normal PostgreSQL server. It can also parallelize queries on the fly by rewriting them and has an advanced caching system. As replication mechanism, we used the internal PostgreSQL synchronous stream replication.

[4] a user has many cars has many parts has a part_type has many parameters etc.

2.6 User interface

The user interface is completely independent from the backend. It features it own template system and event system. It doesn't contain any game logic. It uses solemny asynchronous requests to communicate with the backend server. It uses a 3D renderer ([Unity 3D engine]) for the rendering of the cars. It is possible to create new user interfaces as long as they can handle HTTP and JSON-RPC.

2.7 Communication between components

All the components communicate over the network and are on seperate servers. We use for the HTTP traffic JSON-RPC. For other components, we use binary protocols. In section 5.0.1 there is a topology of a network setup. This doesn't have to be the actual setup, we don't have an optimal topology yet.

3 Security

This section describe security considerations and measures, which are taken in the nodes.

3.1 Code management

We use mercurial for our repositories. These repositories are backed up every hour incrementally. The code is only reachable in the office. Mercurial runs through SSH. We use public private key pairs for the connections.

3.2 Physical servers

All components except for the proxy servers are contained in a virtual private network separated from the internet. The servers are reachable via ssh for the sysadmin. The VPN is shielded by a firewall, which filter all ports, except for the ports needed for the services. We run the server nodes on a several virtual servers in vmware. The servers are all backed up every hour incrementally. It is possible to take snapshots from servers, so we can easily add new servers to the application. A consideration is that the traffic of the internal servers is unencrypted, this is problematic if the security of one of the proxy servers is breached.

3.3 Database

The database is backed up every hour. We run multiple databases via PostgreSQL synchronous stream replication and have a fail over mechanism (through PGPool II). This means, when a database goes down, we startup automatically a standby database and another master will be choosen. We have an online

backup mechanism through write ahead logs ([Wal logs]). Replication is done via synchronous stream replication.

PGPool II has fail over protection. It is running a hot standby server. When it fails, the ip addresses of both instances are swapped.

Data is never deleted only archived in the database.

We make extensive use of foreign keys in the database to prevent inconsistent data.

3.4 General security of servers

All the servers are protected for slow client attacks by timeouts. All servers have a maximum of data they accept. The memory usage of a server is constant for each connection (due to iteratee's). All servers, that uses the database.

3.5 Security of components

3.5.1 Media server

The media is secured by a proxy server. The server itself doesn't feature any encryption. The media server is used to store images from the game. But a potential attacker can construct a special request to send an not game related image. It is not possible to send anything else than an image, we check the magic bytes [5]of the image.

The image size is limited. If a user tries to send more, the connection is killed. The media server isMUNIN protected from slow client attacks, there is a timeout time on the connection. The media server has a connection to the database, but uses a fixed pool to avoid startvation of other resources.

3.5.2 Proxy server

The proxy server is exposed to the outside world, but allows HTTPS for encryption of passwords. The proxy server is protected from slow client attacks, there is a timeout time on the connection. The proxy server has a connection to the database, but uses a fixed pool to avoid startvation of other resources. The proxy server has a access control system based on the simple operations create, read, update and delete (CRUD) expressed as the HTTP methods:

HTTP Method	Permission
Get	Read
Put	Create
Post	Update
Delete	Delete

Every user has a role. For each resource there is defined what the permissions are for a role. Only resources, which have a definition can be reached by the

[5]These are the first 4/5 bytes of an image. This is an almost unique signature of the filetype.

proxy server. The proxy server is always runned in pairs. If the active proxy server fails, the hot (already running) standby server will be actived.

3.5.3 Game logic application server

The game logic application server is a computation heavy of the services. To balance the workload we use load-balancing proxy. When a game server crashes, the proxy server will ignore it, until it comes up again.

The game server uses a database pool to avoid starving the database. Computations in the server (race calculations etc) are limited by time. This will prevent the server from becoming unresponsive. but will break off the connection of the users sometimes. If there is a VPN breach it is possible to directly talk to the backend server. Then a potential attacker can assume anybody or listen to the unencrypted traffic.

3.5.4 Peer to peer storage / Authorization

The authorization system uses a distributed peer-to-peer network as storage medium. This is protected by the VPN, but is not encrypted. It uses a binary protocol over ZeroMQ internal protocol, which stores the routing information in the protocol. An attacker which finds it way into the VPN can create several packets which visit a lot of nodes, this will cause a considerable slowdown. The attacker can listen to the p2p nodes and discover the cookies and with some calculations the associated id of the user. This is not easy due the asynchronous nature of the network, but not impossible.

An attacker can when he breached the VPN, poison the p2p storage with conflicting keys. The attacker cannot read all the values from the network.

A collision of cookies would make the storage network inconsistent. But this is an unlikely event. To generate a cookie we first generate some data with a wultiply with carry random generator with a period of 2^8222 and then compress it with a tiger hash, which is a 192 bits hash. Everytime the server is restarted the seed is renewed. The collision of stored key is almost 0.

Keys have an experation time. This time is quite short to avoid storage and memory problems of the key-value stores.

3.5.5 Monitoring

We monitor the database, pgpool II and the several servers on a very crude level. (Are they alive or not). We also monitor disk usage and cpu time for every server.

The system is not fine grained enough at the moment for high traffic situations. In the servers we use the peer-to-peer message system for sending system monitoring messages, but we don't have a central system to register all the events jet. We need an unifying service for the various components. This is somewhat difficult, because all the monitor tools have their own protocols. We have looked into [Munin], which looked very promising. We also would like to

capture first and second order deriviate from several monitor variables to allow faster reaction on crisis.

We don't have "real" realtime monitoring at the moment even though the peer-to-peer message system is real time.

3.5.6 Passwords

Passwords are stored with a salt and hashed by the TIGER hashing algorithm. At this moment, this gives enough security. There is no known attack on the TIGER hash algorithm to this date. The salt protects against a rainbow table attack. The salt is hardcoded and compiled into the server. Due to the fractured nature of haskell generated machine code, it would be unlikely that someone can find the key easily, but it is not impossible.

4 Performance

TODO describe all performance tests and results.

The performance tests are all done on a single computer. Except for the holistic tests, which are done on the servers.

4.1 Proxy server

4.2 Game logic application server

4.3 Peer to peer storage

4.3.1 Memory backend

This is the storage backend of the peer-to-peer storage. It uses a protocol over a concurrent channel to communicate with the p2p-storage. We have used software transactional memory package to implement it. We have tested three criteria. A query on an empty storage. A insert on a storage and a query time on a filled storage.

Test (times)	Time	Stdev
Query empty (10000)	123.6 ms	15.45 ms
Insert (10000)	161.1 ms	9.8 ms
Search (10000)	132.6 ms	12.12 ms

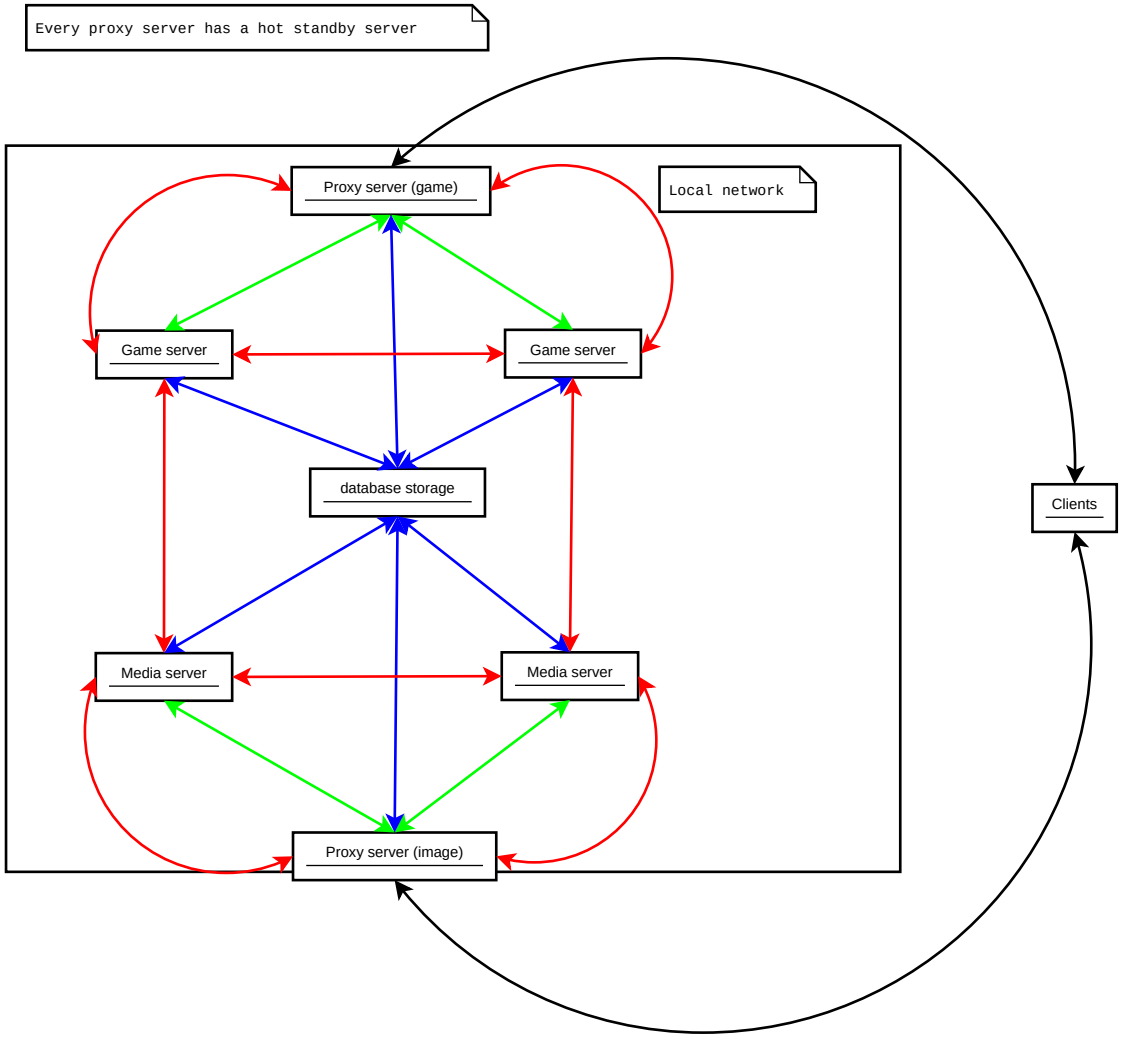
This is quite good. A single insert takes 16.1 microsecond. A search query cost us 13.2 microsecond. These are local times. Network traversal is not tested.

4.4 Whole system

To test the whole system. If first had to design a small tool for creating enough load. I created a small botnetwork to generate enough load.

5 Diagrams

5.0.1 Example server topology



This is a example topology to show a possible servers setup. We still need to determine an optimal topology. The servers in the box are not exposed to the outside world. Only the proxy servers can communicate with the clients. The peer to peer storage is a fine grained network between the nodes to allow a effective distribution of the information. The proxy server has a hot standby server, ready to be activated in times of failure.

Red arrow signifies peer-to-peer communcation between nodes. *Blue arrow* is communication with the database. *Green arrow* is internal JSON-RPC over HTTP communication between the proxy servers and application servers. *Black arrow* is external communication to the proxy servers.

6 TODO

- Adminstration server is not described in the document.
- Tell some more about the peer to peer message system.
- Haskell part is a bit dense.
- Describe performance tests and results.

References

[the IO monad]	http://www.haskell.org/haskellwiki/IO_inside
[Unity 3D engine]	http://unity3d.com
[Haskell]	http://www.haskell.org/haskellwiki/Haskell
[Snap]	http://snapframework.com/
[STM]	http://en.wikipedia.org/wiki/Software_transactional_memory
[JSON-RPC]	http://json-rpc.org/
[PostgreSQL]	http://www.postgresql.org/
[PGPool II]	http://www.pgpool.net/mediawiki/index.php/Main_Page
[ZeroMQ]	http://www.zeromq.org/
[Iteratee]	http://okmij.org/ftp/Streams.html#iteratee
[Wal logs]	http://www.postgresql.org/docs/9.1/static/wal.html
[Munin]	http://munin-monitoring.org
[Snap benchmark]	http://snapframework.com/blog/2010/11/17/snap-0.3-benchmarks
[System F]	http://en.wikipedia.org/wiki/System_F
[FreeBSD]	http://www.freebsd.org/