CHALMERS

# Building a scalable social game server

*Master of Science Thesis*

PETER KLAESSON

Department of Applied Information Technology
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden, 2013

Report No. 2013:008
ISSN: 1651-4759

# Abstract

When building web applications with a broad target group, there is always a possibility that the user count will increase dramatically in a short period of time. Especially when building social application with a potential network effect. In such cases it is important to forestall scalability problems by implementing a scalable design from start. Often the scalability problems emerge at first when it is too late and the application has to be rewritten. Methods, architectures and techniques for building scalable applications are needed as well as methods for verifying scalability before the application is launched. This thesis explores the issue of scalability through the development and testing of an online word game. In the process of doing so, the thesis discusses the definition of scalability, how it is measured and how it is achieved is discussed through a literature study. Further on a scalable architecture is derived from the literature and a complete word game is implemented using the JavaScript based framework node.js. In addition to the server side application, a web browser client is developed to test the server functionality. For server persistence, the NoSQL key value store database Couchbase is used to allow horizontal storage scaling. Player simulations are implemented to mimic real game play to render server load. The simulations record measures such as response time when playing the game at different scaling parameters such as the number of web servers and database nodes in the database cluster at varying number of players. As a result of the implementation, a functional, playable game is completed. The results show some indication of scalability as decreased response time when increasing the number of web server instances. An unidentified problem with the database cluster prevented the simulations going beyond 8000 players as the database cluster becomes the bottleneck. The planned time was not sufficient to solve the unexpected database cluster problem and thus, the final measurements to determine the level of scalability could not be completed. But the conceptual architectural design can be regarded as horizontally scalable with support from the literature.

# Contents

# 1 Definitions and terms

| | |
|---|---|
| Dawg | Directed acyclic word graph |
| DHT | Distributed hash table |
| NoSQL | Unstructured storage of data, could be a simple key - value storage but also offer map reduce and indexing of the content of the data (depending of the implementation). |
| Ply | A turn taken by one of the players. |
| Social game | A game that is played among friends in a social networks such as Facebook. Often it features the ability to challenge your friends either by competing for the best score on a billboard or head on head |
| Virtual cloud instance | An instance of a virtual machine that resides in a data center |
| IaaS | Infrastructure as a Service |
| RPC | Remote Procedure Call |
| SaaS | Software as a Service |

# 2 Introduction

There are approximately 2 billion Internet users worldwide [1]. When developing a web service, each of those 2 billion Internet users is also a potential user of that service. For example, Facebook started in 2004 and by 2008 had a monthly growth of 178% and by now, 2012 have over 845 million users [2]. In the wake of Facebook, many other social applications emerged, i.e. twitter and social games, which can be used in either computer web browser or in handheld devices such as smartphones and tablets. Developing these applications has led to new challenges regarding the ability to spread the computational load on multiple machines and in some cases even multiple data centers. During the growth of Facebook, many novel architectural and technical inventions were required as new bottlenecks were discovered [3]. When scalability problems occur, they can lead to loss of profits as users experience the site as unusable if loading times are too long or if the site is unresponsive. There are a lot of solutions to scalability problems, such as lightweight event driven web servers (e.g. Node.js [4]), message passing languages (such as Erlang [5]), message queues for inter-process communication (such as RabbitMQ [6], ZeroMQ [7] etc.) and in memory key value data storage (such as Redis [8], Couchbase [9] etc.) for fast storage. Often startups seem to redo the same mistakes and neglect the importance of designing for scalability in an early stage. Thus, every start-up project in the field of social games desire quick scaling of their service, especially if they manage to initiate user network effects.

## 2.1 Purpose

The purpose of this thesis is to investigate how to build a scalable distributed web application, a social game, using modern tools and frameworks within a case study.

## 2.2 Limitations

The thesis will discuss common bottlenecks when building scalable applications and also motivate the choices behind the implementation of the game server. It will not cover all existing techniques of databases and web servers, only those that are relevant to the specific problem or in the problem domain.

Besides implementing the server, an HTML/JavaScript client will be implemented for supporting the development of the server.

For the player simulations and scalability tests, computational power will be bought as virtual cloud instances that can be bought per hour, either using Amazons infrastructure EC2 [10] or Ipeer elastic cloud instance [11]

## 2.3 Assumptions

When testing the scalability it is assumed that it will not be required to test for extreme number of users, but model the scalability as a function of the number of machines required for a given number of users.

# 3 Background

In December 2011 Otocolobus AB, a start-up company resided in Gothenburg released a crossword game called Rex Verbi/Ordkungen [12] for the iPad. It features the ability to play against friends either by passing the iPad around or playing against an artificial intelligence implemented on the device. Screenshot from the game can be seen in Figure 1. There is currently a need to make it possible for players to connect to other players through the Internet, to bring more social aspect to the game. As there are plans to make the game available for other platforms a server has to be implemented supporting RPC (Remote procedure call) via HTTP and placing all network game logic on the server to prevent cheating. The choice of HTTP RPC is based on the plans for porting the game for the web browser and connect it to Facebook as a Facebook application, also HTTP is less likely to be blocked by and firewalls or proxy servers as other protocols could be victims of.

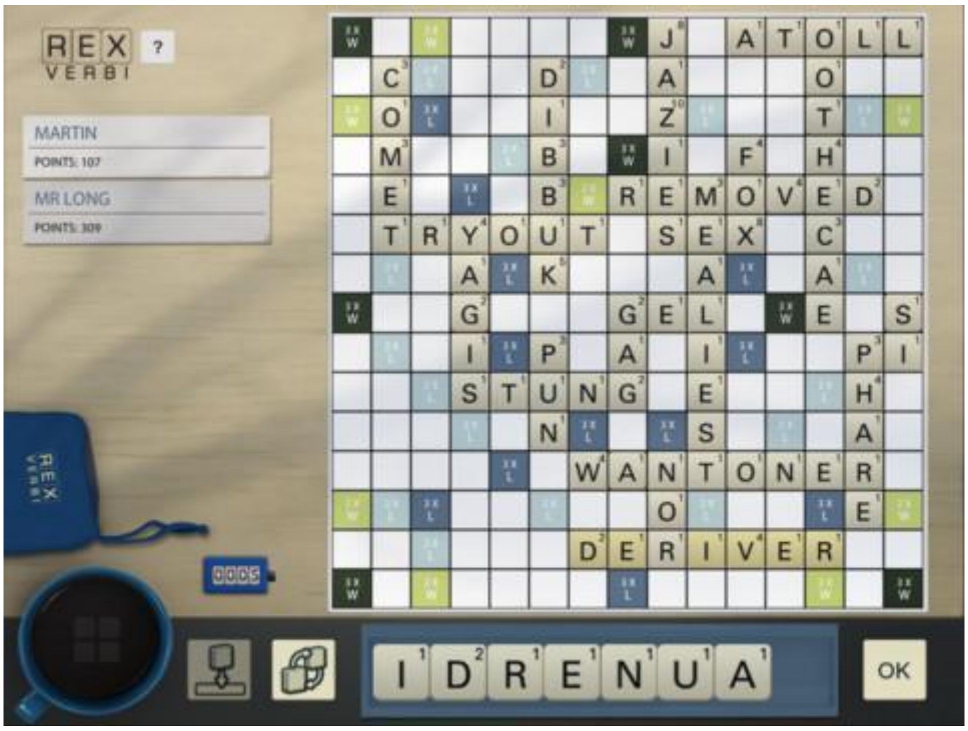


*Figure 1*

The sections 3.1, 3.2 and 3.3 describes other social games and the problems faced when subjected to a large user growth.

## 3.1 Farmville

*Farmville* was at its time considered to be the largest game in the world with 75 million monthly players and 28 million daily players. 4 days after it was launched in 2009 it had 1 million daily players, and passed 10 million daily players after 60 days and 28 million daily players after 9 month [13]. Farmville is a game that is played within the Facebook platform and a lot of data is exchanged between Farmville servers and Facebook. Farmville is a farming simulation where the players manage a farm, plowing land, planting, growing and harvesting crops. The players are able to add playing friends as neighbors to their farm and grow certain crops cooperatively. At peaks, the traffic between Farmville and Facebook reached 3 GB/s. The implementation of Farmville if primarily based on open source components and the core is built using LAMP (Linux, Apache, MySQL and PHP). In contrast to other large applications such as Google and Facebook, Farmville is very write intensive. The ratio between reads and writes is 1:3 and a majority of all requests modifies the users' states. High latencies are considered to kill your application, Farmville solves this by adding cache in front every high latency component.

## 3.2 Words with friends

*Words with friends* was released for the iPhone in July 2009. Word with friends is a crossword puzzle game, just like Rex Verbi. In April of 2012, words with friends had about 20 million players, and was also released for many other platforms such as Android and Facebook [14]. With the initial rapid grows of the user base, the scalability issues had to be solved as they occurred. As these fixes were made, it came to the extent where the fixes became “bandages on bandages”. Eventually it got to the point where the engineers agreed upon rewriting the entire back-end. After 2-3 month of work involving 20 engineers, there was a new structure with a new database ready that would replace the old back-end [15]. The switch would require the existence of two parallel systems meanwhile the users were migrated to the new system, and parts of the user base would be taken offline to perform a safe migration. The migration itself would be a huge risk, requiring a lot of effort during a long period of time. Upon a planning meeting, the risk of switching the system was regarded as to big, and the past 2-3 month of work was thrown away in favor of choosing another, less risky path where the system would be fixed part by part without jeopardizing the users. The latter path would take about 1-2 month longer then the initial plan. One problem with the database was that their moves column had grown too large, approaching the maximum representation of an int. At one point they had only a few weeks before the limit would be reached. On the client side, the move identifier was changed to a long and all the users had to upgrade their client. On the server side, they first considered changing to a new storage method for the moves, but this would either require the existence of two parallel systems or migrating all the data to a new system. Instead, they choose to create an additional table and store the overflowing moves in the new column. This approach worked well even when the additional table was full and the lesson learned was that sometimes the easiest solution is the best. It is not revealed what architecture or database they use for the implementation, other than the choice of ruby on rails for their initial back-end. This game is very similar to the purpose of this thesis.

## 3.3 Happy Hospital

Wooga, a startup located in Germany, developed Happy Hospital in 2010. The goal was to hit one million daily active players, using a small production cluster that is cheap and has a small operational overhead. Happy Hospital is a social game developed for the Facebook platform where the players treat wounded animals. For the implementation they chose using Ruby for the application servers, HA-proxy for the load balancer and Redis as a database. For the initial launch they used only one instance of Redis, but because Redis needs to reside all object in memory, all the data would eventually not fit in the computer's RAM. After a few iterations of trying to write the data to disk, relieving the database of data, they came to the point where they had to partition their Redis instance. By May of 2011, they reached 450000 daily players using only 7 machines, each with a 6 core Intel i7 processor and 24GB RAM. Two of the machines where used to run Redis instances. The peak traffic was measured to 3000 requests per second and 35000 database operations per second. The system managed to remain a response time below 10 ms from the client.  Some days they had a user growth of 50000 new players a day [16]-[17].

# 4 Theory

This chapter discusses previous work regarding the definitions of scalability and how to measure scalability. There is no single general definition of scalability and no single way of measuring. Different ways of obtaining a scalable applications are discussed in 4.3, which is the most important part of this chapter. The definitions and measurements will not be used extensively throughout the thesis, but merely to show that there are different ways to define and measure scalability.

## 4.1 Definition of Scalability

Scalability is defined as "the ability of something, esp. a computer system, to adapt to increased demands" in Collins English Dictionary [18]. This section will explore different definitions of scalability in an attempt to get a definition of what scalability is and how it is defined.

In *What is scalability* [19], M D. Hill tries to find a useful, rigorous definition of scalability. A formal definition of scalability is stated as the ability to efficiently utilize n CPUs in a system for a problem x using the definition.

$$time(n, x) = time\ required\ to\ execute\ work\ x\ on\ n\ CPUs$$
$$efficiency(n, x) = \frac{\frac{time(1,x)}{n}}{time(n, x)}$$

A system is scalable if

$$efficiency(n, x) = 1\ for\ all\ algorithms\ n, x$$

This definition is not found useful since most systems are not completely parallel as stated by Amdahl [20]. He further evaluates an enhanced version of the definition which is using a theoretical parallel machine rather than a real sequential machine, but does not find this very useful either since it is hard to choose a proper theoretical machine [19]. Further on, an attempt to define scalability as "a system is scalable if all its algorithms are evaluated as scalable" is regarded as too much work for system architects and implementers, to be useful. He also questions the view on asymptotic scalability, where all systems are limited to its sequential part, as being useful, since it targets arbitrarily large systems. A system that scales poorly for a large system could very well scale good enough for a smaller system, which is good enough in many cases. The final conclusion of the paper is that the community should either stop using the definition of scalability or find a rigorous definition of it. It is hard to get a clear definition of how to define scalability ant that the scalability should be defined within a certain context.

In the paper *Defining and measuring scalability* [21], two methodologies are explored for defining and measuring scalability. 1) Scaled speedup and 2) fixed-time size up. Scaled speedup is defined as the state when processors and workload are scaled in proportion to one another, i.e. doubling the number of processors decreases the computational time by half. Fixed time size up is defined as the case when an increased problem size can be solved within the same time frame through scaling up the architecture.

Bondi [22], introduces four types of scalability: load scalability, space scalability, space-time scalability and structural scalability in an attempt to identify their impact on performance.

Load scalability is defined as the ability to utilize resources properly under light, moderate and heavy loads without unproductive resource consumption. Space scalability is defined as a system in which memory consumption does not grow to intolerable levels as the number of item it supports increases. Space-time scalability is defined as a system that continues to function gracefully even though the number of items it encompasses increases by an order of magnitude. Structural scalability is defined as a system that does not limit the number of item it could encompass, or at least not within a chosen time span.

Further, these definitions are not completely independent and there are overlapping aspects. For instance the space-time scalability require space scalability and structural scalability.

In [23], the scalability of different architectures are discussed and defined, Scale-up is defined as scaling up a deployment on large shared-memory servers with multiple CPUs. Scale-out is defined as a deployment on multiple small interconnected servers (a cluster). These definitions are also used in the gigaspaces whitepaper [24] where scale-up is defined as concurrent programming which utilizes multi core systems in the context of a single architecture and scale out as distributed programming where the load is distributed across multiple machines over a network.

These definitions are also somewhat related to the definitions of horizontal and vertical scaling. Vertical scaling corresponds to scale-up and horizontal scaling corresponds to scale-out. These definitions are used in [25] defining the scalability of databases. Vertical scaling is defined as master-slave replication of databases. Data is written to the master and propagated to the slaves as where the read operations are distributed among the slaves. Horizontal scaling is where the data is spread over many databases using hash values to decide on which instance to store/read the data. In the latter case, both reading and writing is spread over multiple machines.

Vertical, horizontal scaling and scale-up, scale out are also mentioned and used in [26] and [27], which discusses scalable databases.

The latter definitions, scale-up and scale-out are more loosely defined compared to the earlier definitions, as they target defining scalability as an implementation rather than measuring and comparing performance of algorithms. That may be the case why Hill failed defining scalability in [19], since he focused on ways to define scalability as a measure of performance rather than defining system architectures as scalable by design. This shows that there is no generally accepted definition of what scalability is, but rather that scalability has to be defined within its context. Further, if scalability is to be defined as performance in relation to problem size and resources, a proper measurement has to be defined since otherwise there is no way to evaluate scalability of a system.

## 4.2 Measuring scalability

To verify and predict scalability of software, proper metrics are required. There have been several attempts described in literature to measure scalability. Due to its simplicity, the speed-up metric [28] is among the most commonly used, which just measures the speed-up gained by adding CPU's. But often there are other important parameters, e.g. the problem size [29], system costs [28] or architecture that has to be considered.

### 4.2.1 Speed-up

The speedup metric is one of the most frequently discussed and used definitions. It is defined as how the time of performing a job decreases with the number of processors. The ideal linear speedup is: $S(N) = N,$ where $N$ is the number of processors.

In 1967, Gene Amdahl published a paper [20] regarding the speed-up metrics in which he stated that there is a maximum expected speedup of a program due to its sequential fraction. The sequential fraction is the part of the program that cannot be parallelized. From this paper "Amdahl's law" was derived, as stated:

$S(N)$= Expected speedup, given $N$ processors
$s$ = Sequential fraction of the program ($s = 1 - p$)
$p =$ Parallelizable fraction of the program

$$S(N) = \frac{s+p}{s+\frac{p}{N}} = \frac{1}{s+\frac{p}{N}}$$

$$S(N) \rightarrow \frac{1}{s}, \qquad when\ N \rightarrow\ infinity$$

The final statement shows that there is a theoretical maximum speedup of a program when increasing the number of processors, which is limited by the sequential part of the program.

In 1988, John L. Gustafson questioned the fixed sequential part of a program since the sequential part is not proportional to the problem size, as showed in [29]. The proposed speed-up metric as an alternative to Amdahl's law is the *scaled speedup*, which considers increased the problem size when scaling up the number of processors.
Scaled speedup $= N + (1 - N) \cdot s$

### 4.2.2 Efficiency

$E(N) = \frac{S(N)}{N}$, is the work rate per processor, where N is the number of processors. The ideal efficiency is 1. [28]

### 4.2.3 Scalability

$\sim(N_1, N_2) = \frac{E(N_2)}{E(N_1)}$, is the ratio of efficiency between two system scales $N_1$ and $N_2$.
The ideal value of $\sim(N_1, N_2)$ is 1. [28]

The Efficiency is defined in the previous section.

### 4.2.4 Isoefficiency

Since most systems gain a sub linear speedup when adding processors (as stated by Amdahl's law above), isoefficiency [30] relates the problem size to the number of processors to maintain a fixed efficiency. Increasing the number of processors decrease efficiency and increasing the problem size increase efficiency. The question is at what rate to increase problem size in conjunction to the number of processors. The isoefficiency function can be used to compare a range of problem sizes and number of processors to find the best combination. It also accounts for processor speed and interconnection bandwidth between processors.

Beneath follows a derivation of the general isoefficiency function.

Definitions:
$W$ = Problem size (the number of operation required by the best sequential algorithm to solve the problem)
$T_1$ = Sequential execution time
$T_p$ = Parallel execution time
$T_0$ = Total execution time that is not the sequential execution time, $T_0 \propto W$
$p$ = Number of processors
$t_c$ = The cost of executing one operation in $W$
$pT_p$ = All the time spent on the processors.
This time is equal to the sequential part $T_1$ and the overhead $T_0$, which gives us

$$pT_p = T_1 + T_0 \Leftrightarrow T_p = \frac{T_1 + T_0}{p}$$

The speedup is defined as the proportion between sequential time and the parallel time:

$$S = \frac{T_1}{T_p} = p\frac{T_1}{T_1 + T_0}$$

The efficiency is defined as

$$E = \frac{S}{p} = \frac{T_1}{T_1 + T_0} = \frac{1}{1+\frac{T_0}{T_1}}$$

Since $T_1$ is the sequential execution time we get

$$T_1 = Wt_c$$

And by substitution we get the isoefficiency function as:

$$E = \frac{1}{1 + \frac{T_0}{Wt_c}}$$

If p increases while W is constant, the efficiency decreases because the total overhead $T_0$ increases with p. If W increases, while p is constant, the efficiency increases because $T_0$ grows slower than $O(W)$ (for scalable parallel systems). The efficiency can be maintained at a value between 0 and 1 if $W$ is also increased when $p$ is increased. The rate at which $W$ is to be increased with respect to p is different for different systems. Efficiency can be maintained at a desired value $(0 < E < 1)$ if $\frac{T_0}{W}$ is constant.

When using isoefficiency to analyze a system, $W$, $t_c$ and $t_0$ has to be derived from the system (algorithm and architecture). In [30], an isoefficiency analysis is made for analyzing parallel matrix multiplication on a hypercube multiprocessor hardware. Using this analysis, it is concluded that for that particular problem, to problem size has to grow at a rate of $O(p^2)$.

### 4.2.5 Isospeed

In [31], a metric is produced with the premises that it should be a meaningful and quantitative performance metric that should be under subject for evaluation and comparing. It should also consider both architecture and algorithms as a pair, since there can be scalability overhead in both architecture and algorithm.

The isospeed metric is used to decide scalability while maintaining average unit speed. The average unit speed is (achieved speed of the system / N (the number of processors)). The scale parameters are system size of N processors and work size W. A system is regarded as scalable if average speed can be maintained while increasing the number of CPUs while also increasing the work size in proportion to the number of CPUs. The problem size W is generally referred to as the number of total floating point operations.

A formal definition of the metric and definitions:
$N$ = Initial number of processors
$N'$ = Number of processor when scaled up
$W$ = Work, when $N$ processors are used
$W'$ = Work when $N' > N$ processors are used while maintaining average unit speed

Isospeed function: $\sim(N, N') = \frac{N'W}{NW'}$
In the ideal case, when there is no communication overhead etc.

$W' = \frac{N'W}{N}$ and $\sim(N, N') = 1$
In the general case: $\sim(N, N') < 1$

It is also shown that the quantities system size, problem size (work) and speed are interrelated and reflects the inherent parallel degradation of the algorithm and architecture. This is done by using three different approaches to obtain scalability of two different algorithm-architectural combinations.

### 4.2.6 Strategy-based scalability (productivity - cost)

In [28], a metric for distributed systems is presented. Unlike the scalability metric, [28] defines a metric that considers number of users as scalability factor. It also takes consideration to cost and throughput to define scalability. A system is scalable if between configuration A and B if productivity keeps pace with costs when the system is changed from configuration A to configuration B.

Definition:
$N = \text{Number of users}$
$\lambda(N) = \text{Throughput in responses / seconds}$
$f(N) = \text{Average value of each response}$
$c(N) = \text{Cost at scale } N$

$$F(N) = \lambda(N) \cdot \frac{f(N)}{c(n)}$$

$\sim(N_1, N_2) = \frac{F(N_2)}{F(N_1)}$ where $N_1$ and $N_2$ are two different scales

The ideal case is $\sim(N1, N2) = 1$, but it can be set arbitrarily. A system is considered scalable if $\sim(N_1, N_2) > 0.8$

### 4.2.7 Isospeed-efficiency

In [32], the problem of scalability metric using heterogeneous systems is discussed. The study of previous work concludes that few papers regards heterogeneous systems and that neither isospeed nor isoefficiency is sufficient for heterogeneous systems. A heterogeneous system is a system where the nodes or CPU do not have equal computational power and performs at different speeds.

The proposed isospeed-efficiency metric extends isospeed and is combined with the isoefficiency metric to handle heterogeneous systems.

The definition of isospeed-efficiency:

A new metric is needed to measure the performance of a node, $C_i$. This metric is measured using benchmark software.

$$C_i \ = \ \text{marked speed for node } i.$$
$$C \ = \ \text{Marked speed for the complete system, sum of all nodes.}$$

$$C = \sum_{i=0}^{p} C_i \ \text{for a system with } p \text{ nodes.}$$

Further, definitions of the measurements

$W \ = \ \text{work}$
$T \ = \ \text{execution time}$
$S \ = \ \text{achieved speed,}$

$$S = \frac{W}{T}$$

The achieved speed $S$ varies with system size and problem size
Then we define the speed efficiency metric $E_s$ which is central for the isospeed-efficiency function

$$E_S \ = \frac{S}{C} \ = \frac{W}{TC}$$

A system is scalable if achieved speed-efficiency can remain constant with increased system ensemble size provided that problem size can be increased with system size.

Let $C$, $W$ and $T$ be the initial system size, workload and execution time.
Let $C'$, $W'$ and $T'$ be the increased system size, workload and new execution time.

Isospeed-efficiency condition: $\frac{W}{TC} \ = \ \frac{W'}{T'C'}$
The isospeed-efficiency function: $\sim(C, C') \ = \ \frac{C'W}{CW'}$
In the ideal situation we will have $\sim(C, C') \ = \ 1$ but generally $\sim(C, C') < 1$

In the special case where we have an homogeneous system, since all $C_i$ are equal, $C = pC_i$ and $C' = p'C_i$

$\sim(C, C') \ = \ \frac{C'W}{CW'} = \frac{p'W}{pW'}$, which is the isospeed scalability metric.

When using the isospeed-efficiency metric all nodes has to be benchmarked to determine $C_i$, in the general case this is measured using floating point operation per second (FLOPS).

### 4.2.8 Measuring in cloud services (Infrastructure as a Service)

In recent years, IaaS (Infrastructure as a service) has emerged as a way to buy computer resources. It offers the potential of buying resources on demand with little or no initial costs in contrast to buying and owning own hardware. The infrastructure is located in large data centers with thousands of computers, and these systems and the services they provide are typically given the prefix *cloud* since users do not know (or care) where they are located. There are several cloud resource providers e.g. Amazon EC2 [10], Microsoft Azure [33] and Ipeer elastic cloud [11], where cloud instances can be bought and paid for per hour. A cloud instance is a virtual machine that could be run on any of the computers in the datacenter. The resources allocated by the virtual machine can vary and it is possible to pay more to allocate more resources. In general, a physical machine resides many virtual machines, hence virtual machines cannot be regarded as homogeneous when regarding performance. Even the performance of a single machine may vary depending on the load of the underlying hardware.

A proposed metric targeting cloud scalability of SaaS (Software as a service), is presented in [34]. The proposed metric measures the resources consumed and the definition also account for variance of the system performance. The metric PC is defined as performance change when workload changes.

Definition:

$$PC = \frac{PRR(t) \cdot W(t)}{PRR(t') \cdot W(t')}$$

To define the performance change $PC$ from workload $t$ to workload $t'$, we need to define $PRR$ and $W$

$W(t)$ is defined as the work performed at workload $t$
$PRR$, performance/resource ratio is defined as $PRR \; = \frac{1}{T_w} \cdot \frac{1}{C_R}$, where $T_w$ is the processing time and $C_R$ is the total consumed resources for the system

$T_w = T_q + T_e$ ($T_q$ is the queuing time and $T_e$ is the execution time), $T_w$ is the total wait time.
$C_R = \sum R_i \cdot T_i$ ($R\_i$ is the resource allocation of resource i and $T_i$ is the time for allocation of resource $i$, hence $R_i \cdot T_i$ is the resource consumption of instance $i$)

The second metric defined is the performance variance $PV$, $PV$ can be measured using $PC$ and calculate the standard variance of the $PC$ in multiple runs.

$$PV \; = \; E\left[\left(PC_i \; - \frac{1}{n} \cdot \sum_{i=1}^{n} PC_i\right)^2\right]$$

The ideal case for a scalable system is $PV = 0$ and $PC = 1$.

## 4.3 Obtaining scalability

If scalability is to be considered when building applications from ground up, there are not only technical aspect to consider [35], but rather the development process and organization also have to be designed to scale. A larger software may require a larger number of developers and a widely used system requires more maintenance. In the previous sections we have discussed how to define and measure scalability, but the key question of this thesis is how to implement to obtain scalable applications. In this section we will discuss ideas and conclusions from previous attempts and experience from papers targeting the subject. First of we will discuss different aspects of obtaining scalability such as how to organize the development team and touch the subject of eliciting measurable requirements. Then we move on to the technical aspects of how implementation, discussing different architectural strategies.

### 4.3.1 Performance engineering

Performance is a non-functional requirement of the software, performance is often measured as response time when using the software. When fulfilling performance requirements, the development is about optimizing rather than adding new functions. A performance requirement could be that a system should perform a task within a given time threshold. I.e. “When user exchange tiles, the response time should be within 800 ms.” or “When user logs in, the screen should be loaded within 2 seconds”. Performance requirements should not be vague such as “The page should load as fast as possible”, because such a requirement is not measurable [36].

Best Practices for Software Performance Engineering [36] presents 24 “best practices“, for software performance engineering divided in four categories: project management, performance modelling, performance measurement and techniques.

*Organization, management*
The organizational part of the requirement engineering is about estimating performance risks, i.e. the use of new technologies, the impact of architectural choices etc. It should also be about tracking costs and benefits of applying performance engineering; integrating performance engineering into the development process; establish quantitative performance objectives and ensure that developers have the correct training and tools for implementing performance engineering. The practices are aimed for project and software managers and performance engineering specialists.

*Performance modelling*
Performance modelling is about designing the software architecture. It is better to spend time evaluating architectures and design models before implementation, then on refactoring and redesigning. Best and worst case scenarios should be used when estimating the resource requirements for the software execution. If the best case is not satisfactory, there are severe problems with the design and a feasible alternative has to be found. If the worst case is satisfactory, the development can advance to the next phase in the development process.

Using real measurements of prototyped models and implementations is recommended to validate performance and resource consumption at an early stage and reuse them during the development process. These measurements can be used to ensure that changes and refactoring of the implementation do not invalidate the models.

*Measurement*
Measurements are about validating the performance requirements. It is important to use both representative and reproducible measurements. The measurements have to correspond to the requirements such as response time and workload. Profile the code by measuring execution time of certain code sections, especially critical components, which needs to be identified as early as possible. It is crucial to pay extra attention to these parts so that they are not discovered late in the development when it is more expensive to address them. The measurements should also be used frequently during the development process to ensure that changes do not invalidate the model.

*Techniques*
The techniques are about how to work as a team to achieve a common goal of performance.

Scalability, as it is a non-functional requirement could be viewed in the light of performance engineering. In conjunction to measuring how well a system scales, performance measurements can be used to assure that the systems behaves as expected when scaled to accommodate a large number of users.

Scalability requirements have to be defined and elicited. A Case Study in Eliciting Scalability Requirements [37] proposes goal-oriented requirements engineering, referred to as GORE. GORE is about a natural elicitation of the stakeholders goals. Goals can be defined either as high level (strategic concerns) or low level (technical). The goals, as for scalability can be i.e. quality of service, performance and response time. Evaluation of different solutions is made with these goals in mind. The goals cannot always be satisfied in an absolute sense since there are often technical limitations. The paper exemplifies goals as performing a task in a given time frame based on certain assumptions in the context of a fraud detection system. The drawbacks of using GORE for eliciting scalability requirements presented are mostly about choosing the right assumptions, such as expected workload. A goal can vary over time since its premises can change, i.e. performance requirements changes.

### 4.3.2 Scalable architectures

In 4.1 we defined horizontal and vertical scaling and their synonyms scaling out and scaling up. [23] compares the two strategies Scale-up and scale-out by implementing an emerging commercial application. The application is a search engine using map/reduce functions to index documents in a nutch/lucene index. The two approaches initially used two different hardware setups, approximately equally priced. For the vertical scaling, a larger IBM multi core machine is used and in the horizontal scaling, a cluster of smaller IBM machines is used. The chosen problem is easy to parallelize. The documents are divided into smaller parts and

distribute the parts among many nodes or CPU cores which runs the map/reduce functions or the query. As conclusion it is stated that the scale out strategy gave 4 times the performance compared to the scale up strategy, when measuring queries per second. The systems used where of approximately the same cost. The scale out strategy even performed better than the scale up strategy when running within a single machine. Hence “Scale out in a box” gives better performance than scale up for this specific problem. The stated downside using the scale out strategy, is that scaling out requires more administrative resources since it requires more virtual systems to be administrated. When scaling up, only one instance is required.
In *Performance scalability of a multi-core web server* [38], the authors investigate the scalability of web servers, scaling up on up to eight CPU cores on a single machine. They define scalability as the ability increase the performance when increasing resources, in this case the speed gain and number of CPU cores. The measured scalability of eight cores compared to using only one core is 4.8x (the ideal case is 8x). In comparison, scaling up to four cores gives close to 4x the performance. They identifies some bottlenecks in the architecture and the main bottleneck is that the address bus gets saturated when using all of the eight cores. But they state that if the problem with the address bus is solved, web servers are well positioned to scale up on even larger number of CPUs.
In [25], Rightscale proposes a scalable reference architecture, derived from collected knowledge and experiences of “successful deployments” of scalable applications on cloud platforms (IaaS). It is stated that despite there is no single architecture suitable for all situations. There is a reference architecture that can be used as a foundation. The suggested architecture, which has been commonly used by Rightscale consists of four tiers. A load balancer tier, an application tier, a cache tier and a database tier, Figure 2.

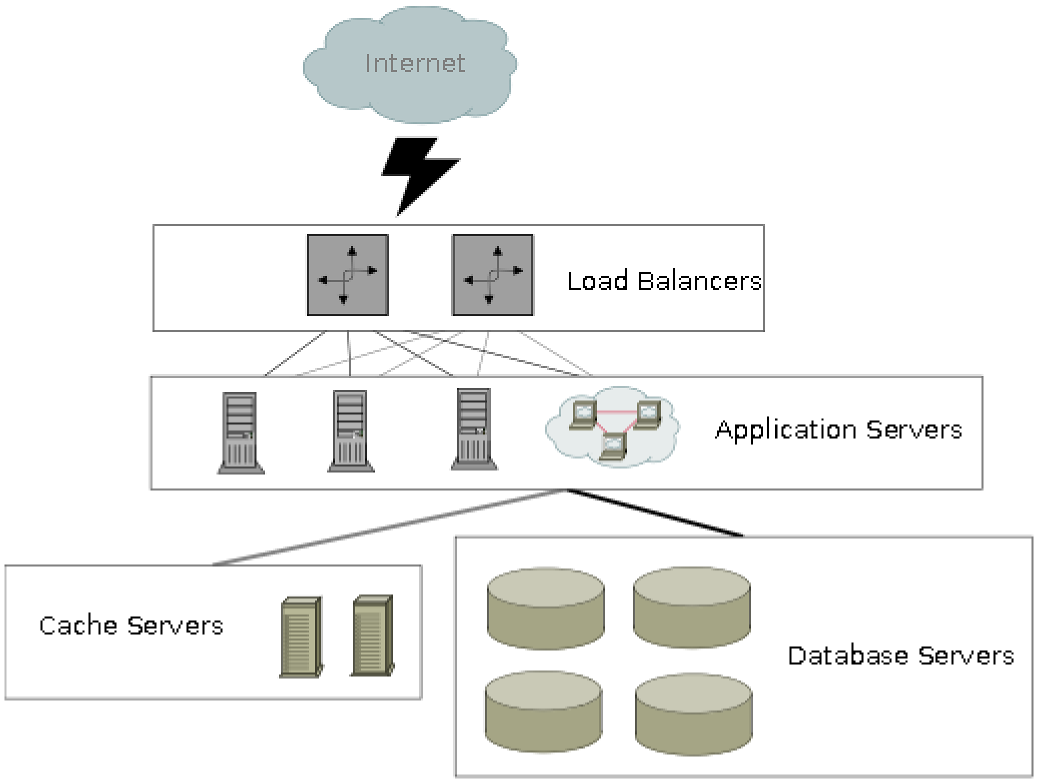


*Figure 2*

The load balancer tier distributes the load among the application servers by partitioning the connections to the application servers. The application tier handles the connections and performs the data processing. It can, in addition to web servers that handles the user requests, consist of dedicated applications servers to handle data processing. It is recommended to scale the application tier automatically based on metrics on the servers such as CPU utilization, memory consumption and system load. When a threshold is met the number of servers could either be increased or decreased to meet the current demand. It is also possible to scale up each server instead of scaling out the server count. The caching tier is used for read-intensive applications and to relieve the database server from load. The caching tier would preferably consist of servers with a large amount of memory, since the data needs to reside within the computers RAM to provide low latencies. The database tier should consist of instances with high CPU-allocation (if relational databases are used) and high I/O-performance. It is sufficient to scale up the servers to gain better performance. When the load cannot be met by a single instance, horizontal scaling has applied. For relational databases, it is easy to achieve better read performance by adding readable slaves to the database. For true horizontal scaling, it is required to partitioning the data among many servers. For relational databases this requires partitioning of the database tables. This might not always possible, since SQL-servers rely on the ability to join data between tables, and the tables has to be stored within the same database.

### 4.3.3 Scaling the database tier

A truly scalable database system, both in terms of space and performance, has to be able to run on multiple machines. One machine can only reside a finite storage and computational capacity. The storage and computations has to be spread to several machines.
Traditional RDBMS (Relational Database Management System) databases have a strong consistency constraint referred to as ACID (Automatic, Consistent, Isolated, and Durable). The different aspects of the consistency constraint are described below:

- *Atomic:* The transaction is performed in sequence and allows rollback if one of the commands in the transaction fails. Either all commands are applied to the database or none are.
- *Consistent:* The database is in a consistent state both before and after the transaction.
- *Isolated:* Since a transaction is Atomic, parallel transaction should not affect other concurrent transactions.
- *Durable:* Immediately after a transaction the database changes should be persistent and saved to the database.

RDBMS systems traditionally run on a single machine and in order to scale up the capacity of storage and operations per second the strategy is to replace that machine with a faster machine with more memory [27]. Although, in recent years, new versions of traditional RDBMS, designed to scale horizontally have emerged. These databases have limitations, and recommends not using all of the features in the SQL because it renders in slow operations. Instead small-scope operations and transactions are recommended, operations and transactions that does not span over multiple nodes [26].

There a new set of databases referred to as NOSQL databases, which has emerged in recent years and used where ACID transactions are not required. NOSQL databases offers better performance and horizontal scaling by relaxing some of the transaction constraints in RDBMS [27].  Most NOSQL databases are designed to operate in distributed environments. In the section about the CAP theorem below, the difficulties of database models in distributed environments are discussed.

*The CAP theorem*
In the year 2000, Eric brewer presented the CAP theorem at Symposium of Principles of Distributed Systems. It is stated that it is only possible to get two out of three from Consistency, Availability and Partition-tolerance. The theorem is proved in [39] and the concepts of the three are explained below.
Consistency: All the nodes see exactly the same data at all time
Availability: If a node goes down, the system is still operational.
Partition-tolerance: If communication is lost between nodes, they can still operate.
Traditional RDMBS offers full consistency, but has then to offer one of Availability or Partition-tolerance. The NOSQL databases relax the Consistency and offers Availability and Partition-tolerance [27].

To achieve horizontal scaling, distributed hash tables (DHT) are used [27]. A simple implementation of a DHT is to hash each key that is being stored into an integer value and use the modulo function to decide into which node to store the key. E.g. storing the key 2142 in a cluster of 4 nodes, 2142 % 4 will store the key in node 2.
[27], [26] and [40] investigates the availability of scalable database servers, mainly NOSQL databases by comparing their feature sets. No benchmarking or measuring has been conducted in any of these papers.

# 5 Methodology

The work on this thesis will consist of the following parts:

- Analysis of previous work regarding scalability
- Evaluation of architectural designs and techniques
- Server/game implementation
- Implement player simulations
- Analysis and verification using player simulations

At first, the aspects of scalability will be discussed. Papers, articles and presentations regarding previous mistakes and possibilities will be evaluated to be the foundation of the architecture for the implementation. Then the server will be implemented based on the motivations from the previous works. In the last step, player simulations will be used to verify the scalability. A player robot will be implemented to simulate real workload on the server(s). The analysis and test will be performed using multiple virtual machine instances to be able to test a variety of different number of clients and servers. The analysis will be based on monitoring the servers by logging response time, CPU load and number of concurrent users and if needed profiling the code. The main measurement of choice, will be the response time as it is the measurement that affects the users.
The implementation will carried out using an iterative process where the existing game and test suits will be ported to a server platform. To ease and shorten the testing process, in conjunction to the server part, a web base client will be developed for easy system and acceptance testing.

# 6 Planning

The first week will be used to structure a report template and gather relevant previous works. The next four weeks will be used for reading previous works, systematically analyzing pros and cons of architectural designs and techniques and start identifying suitable components for the game server such as database, web server etc. Then eight weeks will be used for implementing the server (most of the game logic exists and need to be ported to the chosen platform). Three weeks will be used for implementing the mock up client. Then the remaining four weeks will be used to evaluate the result, analyzing the scalability and refine the server, revisiting the literature compare the result and preparing the presentation. A Gantt chart is represented in Figure 3.

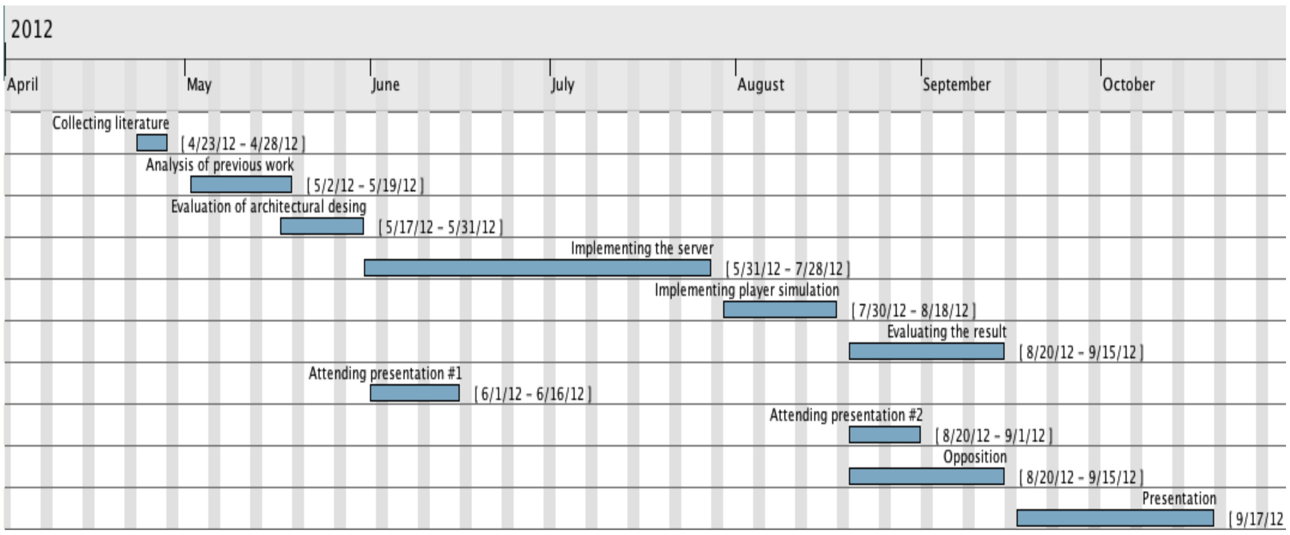


*Figure 3*

# 7 Realization

## 7.1 Preparatory research

Before implementation of the web server started, the definition of scalability had to be clear. As shown in the theory chapter, scalability can be defined in many ways, not only in the context of performance of applications, but also permeate the process of constructing and maintaining software to the surrounding organization. The main focus in the theory chapter was concerned with defining scalability of software, where the throughput can grow with increased resources. Further on, a few ways of measuring scalability were presented. This section in the theory chapter will be revisited later on in the realization as the need for an adequate measurement is needed to verify the scalability of the implementation.
About four weeks was dedicated to reading papers from IEEE, ACM and white papers that were relevant to this thesis. The content of these papers are presented in 4.1 and 4.2.

The next section in the theory chapter, 4.3 obtaining scalability, is the foundation of the next part of the realization where the technical decisions are made. Choosing the architecture and technical solutions. At this point, it was good to have a notion of the different characteristics of scalability to be able to reason about scalability.

The conclusion of the preparatory research is that a scalable application should use horizontal scaling. The four level tier suggested in [25], is an adequate foundation of the architectural design because it relies on horizontal scaling. This implies the need for a horizontally scalable framework for the application server tier, a data storage that scales horizontally and horizontal scalable cache servers. The load balancing tier that distributes the load among the application servers can be supplied from the IaaS vendor or if needed be investigated at a later point.

The first step of implementing the server was to port the game logic from the iPad game. The iPad game uses the design pattern MVC. The models and controllers had to be copied and rewritten to run on the web server tier.

A state of the game consists of the players, the board, the tile bag and a log of previous moves. The controller takes care of the player moves which are placing tiles, changing tiles and passing. The second step was to implement an RPC-API, through which the client communicates with the server. The server holds the game states and the client accesses the state via RPC. During the implementation of the server a client that runs in the browser was developed to provide better testing.

### 7.1.1 Choosing framework and programming language

As a first attempt Erlang was investigated as the language for implementing the application server. It is a language that is designed to scale and uses message passing to pass messages between processes. Processes can be spawned either locally on the same machine or on other machines seamlessly. Erlang is a functional language, which diverged from the language paradigm used for the original game implementation, objective C. Objective C is an imperative language and rewriting of the code to a functional language would not be straightforward. There were a few web libraries, I.e. Mochiweb [41] and a complete web server, jaws [42], which were promising suitable for the implementation. The first step was to get the web server and development tools up and running, then an RPC protocol. The yaws web server provided a library for building a JSON-RPC, and was easy to install and configure.

Then implementation of the game logic began, but was quite soon abandoned because of the lack of efficient array operations in conjunction with the language conversion difficulties. In the game, most of the operations on the model data for the board are array operations. As a result of the functional nature of the Erlang language, lists are represented as linked lists resulting in a $O(n)$ lookup time for a list of size $n$. A board in the game consists of $width \cdot height$ squares, resulting in a $O(width \cdot height)$ lookup operation for a square. In the imperative source language an array lookup is $O(1)$. When placing tiles on the board, each placed tile requires at least two lookups to verify that the placement is valid. One to see that there is no other tile placed on that particular square and one to make sure the tile is adjacent to another tile. Then there are more lookups involved when building the words and calculating the score.  Since there are plans to allow larger boards in the future, the array operation would be quite expensive. But the main cause of the language switch was the awareness of the time-consuming code rewritings.

### 7.1.2 Choosing another framework and programming language

The main problem for handling user requests as stated in [43] is blocking system calls, like read/writes to sockets and the file system. There are mainly two strategies to handle this. One is to, for each connection, spawn a thread that handles the connection and allow blocking I/O. For each connection, a new thread is spawned to allow concurrent blocking I/O. The other strategy is to use non-blocking, asynchronous I/O operations and an event loop. All the connections are handled in the same thread, and instead of waiting for the I/O operation to complete, the event loop continues to handle the next event in the loop. When the I/O operation is completed another event is triggered to the event loop to handle the result of the operation.

A comparison between state of the art event driven and thread per connection web servers using a single processor computer concludes that the event driven strategy performs 18% better than the thread per connection strategy. The comparison utilizes a large number of simultaneous connections and tries to emulate a real world workload. An analysis of the state of the art thread library used for the thread per connection server shows that the underlying implementation of the thread library utilizes an event loop and that the threading layer adds an extra overhead on top of that. In the measurements in the comparison between the web servers, the thread overhead is twice as large as the event overhead [43].

Given the result in [43], event driven frameworks were considered for the implementation, also another important aspect was that it should offer easy to use libraries suitable for implementing the server.

One framework fulfilling these criteria's is Node.js. Node.js is a quite new framework and have gained lot of attention from large software vendors, including Microsoft, Yahoo and LinkedIn [4]. The development of node.js started in 2009 and is written using the C++ event library Lib-UV and Google's JavaScript engine V8. The choice of JavaScript as implementation language is, according to the author Ryan Dahl, the event driven design of the language. Java script is used mainly in web browsers which are also run in an event loop and that there exists no eco system of blocking I/O libraries. This allowed the development of new asynchronous libraries and utilizing existing browser libraries which are already written asynchronously. Node JS offers about 14000 packages, with libraries and frameworks through the node packaged modules registry [44] including modules allowing to access most of the existing databases.

## 7.2 Implementation

### 7.2.1 Source control and development environment

For the development the JavaScript IDE Webstorm [45] was used, it has support for node.js and simplifies the development with easy accessible debugger. As for source management, Git is used and is hosted at Bitbucket [46], which offers free private repositories.

### 7.2.2 Porting the code to node.js

Rewriting the models and controllers was much more straightforward using an imperative language. Most of the functions were ported just by adjusting the syntax and write more rigorous tests to compensate for the lack of type checking in the JavaScript language. The test framework used for running and implementing the tests is nunitjs [47].

The next step was to define protocol for communication between the client and the server. A requirement decided for the protocol, is that it should be allowed by proxy servers and firewalls, thus utilize the standard http-protocol. The problem of using http as transport protocol for the RPC protocol is that http has no native support for bidirectional communication. For instance, if the player has to be notified by the server that the game state has been changed. It is not possible for the server to send a notification to the player. There were three ways of dealing with this problem:

- Let the client poll the server for changes periodically. The problem with this is that, depending on the interval of the poll-requests, it does not allow for real time communication since the client will not be aware of any changes until the client makes the request.

- Use long polling, the client sends an HTTP request to the server and the server keeps the request in case it needs to change a message. Usually there is a timeout for the requests of about 15 seconds before the client sends a new request.
- Use web sockets. Web sockets is a relatively new extension of the http-protocol which allow bidirectional communication. The client opens a socket for which client - server, server - client communication is established. The problem using web socket is that it was not supported by all web browsers, proxy servers and firewalls.

There were plans to implement a version of the game where a user have only a few seconds to make her ply. Thus, it is desirable to have the option of real time communication from the server and make long polling or web sockets the possible option. Web sockets are cheaper to keep alive than using long polling since long polling relies on repeated HTTP-requests. The optimal way to deal with this is to use web sockets when available and long polling where it is not, and abstract this into a library.

Instead of writing such a library, socket.io [48] was used. Socket.io offered a socket layer abstraction that uses Web Socket if available, otherwise falling back to other transport protocols such as Adobe® Flash® Socket, AJAX long polling, AJAX multipart streaming, Forever Iframe or JSONP Polling. Socket.io allows sending events bidirectional and binding functions to those events.

Binding event: (looks the same on both the server and the client)

```
socket.on('event', function (data) {
    doSomeThingWithData(data);
}
```

Sending an event:

```
socket.emit('event', data);
```

By using socket.io as a transport protocol, an RPC protocol was defined inspired by JSON-RPC [49] to allow the client to make function calls to the server to be able to receive game states and make plies. By abstraction layer on top of socket.io, it is possible to use another transport protocol library in the future.

An RPC is essentially like a function call in any programming language that takes arguments and returns a result. An example of a method call is defined below.

Method call:

```
{method: "methodName", parameters: [p1, p2, p3...], id:"call id" }
```

Function result:

```
{ error: error || null, result: resultData, id: "call id"}
//error is either null or undefined if the function call proceeded
```

```
// without any errors.
```

### 7.2.3 The first iteration

When having the basic game logic and communication protocol in place, the development of the web browser client started. The objective for the first iteration was to get everything that works on the iPad game also work on the server version and the ability to register users.

The tasks were implemented in this order.

- Be able to register as a user, with a password
- Be able to create a game against a random opponent
- Render the board, tiles, tile racket, players and active games
- Be able to drag and drop tiles
- Be able to verify words (dictionary)
- Be able to make all of the three game moves (place tiles, change tiles, pass)
- Notifying other players upon a move

#### 7.2.3.1 Be able to register as a user, with a password

For handling the server request for static files such as images, JavaScript and CSS files and handling sessions, the framework expressjs was used. Expressjs [50] added the possibility to use middle wares which can be used for authentications, routing etc. Using a framework for handling the request simplified setting up the client by serving the JavaScript files and allow for HTML templates for rendering HTML. In addition to the express framework, a middleware called everyauth [51] is added to handle user authentication and adding users.

Since no decision of which database to use had been made, an in memory key value store, Redis [8] was used which was abstracted through a data storage layer. The intention was to replace Redis with a horizontally scalable data storage later on. Besides key value storage, Redis also offers message queues for inter process communication. I.e. message passing between two node.js nodes. Communication between nodes was needed later on to be able to scale out node.js instances.

#### 7.2.3.2 Be able to create a game against a random opponent

The next requirement - the ability to start random games against random opponents required some planning to prevent concurrency collisions. A new random game can have the following variations: 2-4 player, random or standard board which gives 3*2 = 6 different kind of games. For the trivial case, with just 2 players and only on type of board, a queue of only length 1 is required. The algorithm is quite simple.

```
Try to dequeue the queue
if a user is already in the queue (we have a match)
 start the game
else
 put the user in the queue
```

This algorithm will work if we only have one instance that accesses the queue sequentially. Otherwise, we can have the problem where two instances tries to dequeue the queue at the same time, finding no user in the queue and then both of the instances will add a user to the queue, resulting in a missed match. There is also the case where there is one player in the queue, which the two instances find and both of the instances will start a new game from a match with the player in the queue. The solution to this problem is to use transactions or lock the queue when it is accessed, or to only let one process access the queue. Essentially all of the values stored in the key value store is exposed to this problem. The most severe collisions will occur upon when games are created. An assumption was made that there will be far less new games created per minute than game moves per minute and therefore the latter option is chosen to address the shared memory problem. It will also solve the problem that can occur when adding a game to a user's game list, since that operation requires first a read of the game list, adding a game and then writing the game list to the key value store again.

A new process called "player matcher" is constructed which is connected to the node instance using Redis. Redis offers the ability to store lists which can be used as queues by using the commands push and pop and also pop and block which blocks the pop command until any other client pushes something into the list. In this way, the queue can be used as a message queue. When a player wants to start a new game, the game server instance puts a new game request into the queue, with the parameter data.

The player matcher then pops the queue, hashes the parameters as a key, i.e. for a 2 player game with random board: 2_random. The player matcher then puts the key in a hash map and adds the player as a value to the hash map. In this way all the configurations of games can be handled the same way. If there is one player wanting to start a 3 player game with random board, the first player is put as a value into the hash map.
"3_random" -> [first player]
Then a second player does the same and the second player is added to the value of the key.
"3_random" -> [first player, second player]
The player matcher checks the lengths of the value.
If the number of players is fulfilled, the key is removed and a game is started for the players in the queue.

If this was a feasible solution would be revealed when testing the scalability using player simulations. The current solution was a quite isolated part and will be easy to exchange if needed.

#### 7.2.3.3 Render the board, tiles, tile racket, players and active games

For the client, the basic frame of the game view was rendered on the server using express and jade templates. For DOM manipulations on the client jQuery [50] and jQuery UI [51] were used. Besides jQuery, the utility library underscore [52] was used for list and collection manipulation. The view was rendered using the model received from the server. The complete game state was sent to the client excluding the other users' tiles. Figure 4 illustrates the view after the first iteration. The accordion to the left is a placeholder for where the chat and log will be placed on the client, but was not functional at this time.

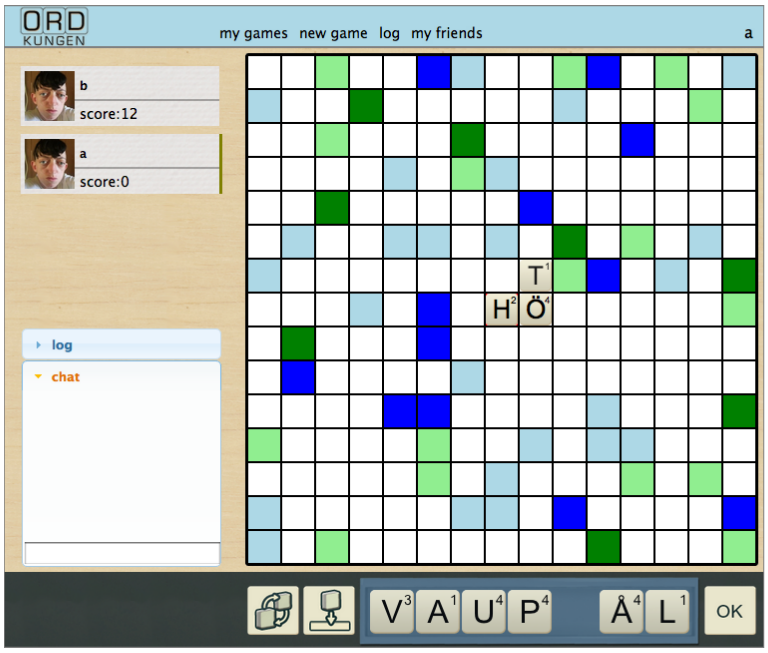


*Figure 4*

### 7.2.3.4 Be able to drag and drop tiles

For the ability to drag and drop the tiles, JQuery UI is used, which can extend HTML DOM elements with drag and drop functionality. e.g. `$('.tile').draggable();`

### 7.2.3.5 Be able to verify words (dictionary)

The word verification in the original game is done using a directed acyclic word graph, dawg as described in the paper world fastest scrabble program [53]. It is a compact representation of word graph and very fast for verifying words. At first the code from the client was reused and wrapped as a node C++ module. This implementation required a lot of maintenance and recompilation after each new node.js release. The dawg was rewritten using native JavaScript to simplify the maintenance.

### 7.2.3.6 Be able to make all of the three game moves

The prerequisites for making a move are, the user is part of the game, the game has not ended and that it is the players turn. The moves are exposed through an RPC handler which handles the incoming RPC's from the client.

**Placing tiles**

When placing tiles, it is first checked that the placement is valid. All tiles in a row or column, all tiles are adjacent to another tile, no tile is placed on top of another tile. Then all words are built from the placement, then the words are verified. If there are invalid words, those words are sent back to the client as invalid. Otherwise the scores are calculated. The player is handed new tiles, the game state changes player and a success response is sent to the player with information of which words were placed, the score and the new tiles.

**Changing tiles**
The client sends which tiles to be exchanged.
The server checks that there are enough tiles left in the bag (more than seven), if there are more than seven tiles left, an equal number of tiles are picked from the bag and dealt to the player before the players tiles are returned. The game state changes player and then a response is sent to the player with the response.

**Pass**
The pass only changes the player and increases a consecutive passes counter (which is reset upon placing or changing tile moves).

#### 7.2.3.7 Notifying other players upon a move

When a player makes a move, other players have to be notified. The implementation used the publication subscription feature in Redis. When a move is made, a message is sent, addressed to all players in the current game state except the playing player. The message is sent through a server message channel, which the server also subscribes to. All connected players are stored in a hash map on the server. When the server receives a message through the server message channel, it checks against the hash map to see if the players is connected. If the player is connected, the message is sent from the server to the user via the socket.io connection. Otherwise, the message is dropped. The use of publication subscription feature in Redis will make the notification to work, even if the players are connected to different servers.

#### 7.2.3.8 The implementation structure after the first iteration.

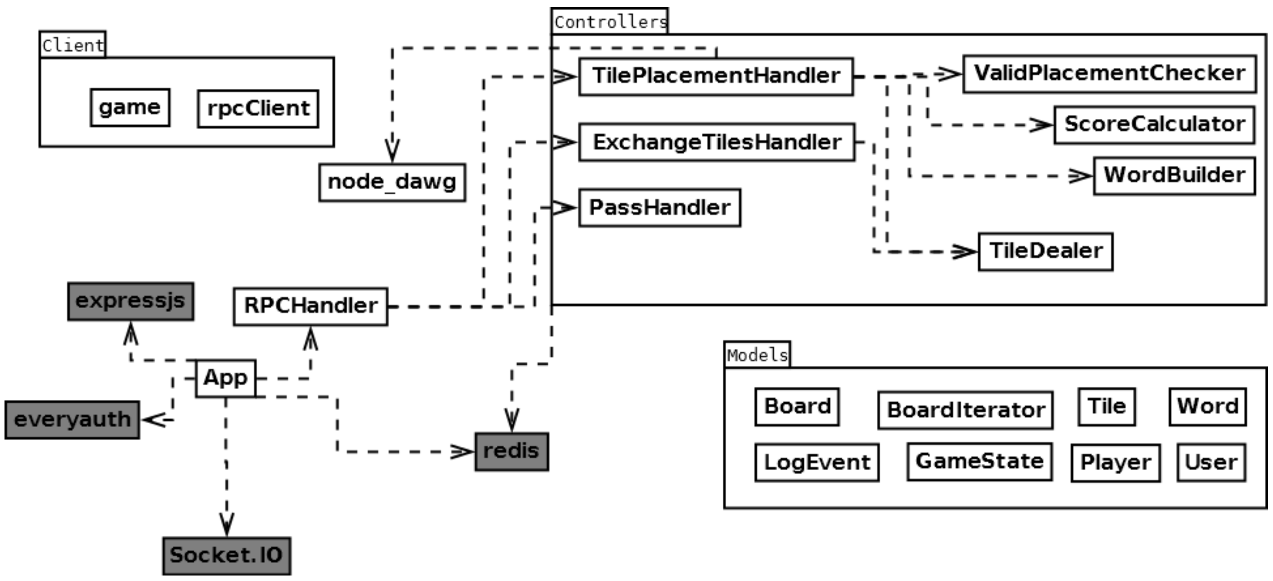
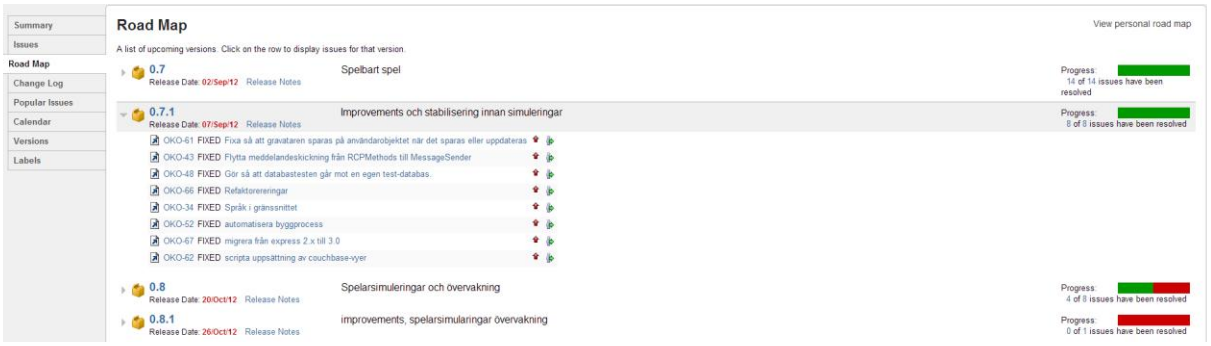


*Figure 5*

The first structure of the application, Figure 5. Each game move was handled by a separate node module, which were in the controller’s directory. These were used by the RPC handler to serve the client. The client connected through the socket.io library and got access to the exposed RPC methods. Everyauth was used to verify the user via an http form. The model scripts were used to wrap the mutation of the JavaScript objects. By this iteration, the client only consisted of two files plus the jQuery library.

### 7.2.4 Introducing an issue and project tracking software

After fulfilling the first iteration it was clear that it was hard to keep track of what to do next and how much time that has been spent on different tasks. It was also hard to plan the iterations. From now on, an issue tracking system was used to plan the iterations. Estimating how much time each task would take and when a task would be complete and log the amount of hours spent. For issue the issue tracking system, Jira [54] was used. This simplified planning the iterations and helped the writing of the report since all tasks were stored in the system and for which iteration they were implemented. Figure 6 shows a screenshot from the road map view in Jira.



Road Map

A list of upcoming versions. Click on the row to display issues for that version.

0.7 — Spelbart spel
Release Date: 02/Sep/12 Release Notes
Progress: 🟩 14 of 14 issues have been resolved

0.7.1 — Improvements och stabilisering innan simuleringar
Release Date: 07/Sep/12 Release Notes
Progress: 🟩 8 of 8 issues have been resolved

- OKO-61 FIXED Fixa så att gravataren sparas på användarobjektet när det sparas eller uppdateras
- OKO-43 FIXED Flytta meddelandeskickning från RCPMethods till MessageSender
- OKO-48 FIXED Gör så att databastesten går mot en egen test-databas.
- OKO-66 FIXED Refaktoreeringar
- OKO-34 FIXED Språk i gränssnittet
- OKO-52 FIXED automatisera byggprocess
- OKO-67 FIXED migrera från express 2.x till 3.0
- OKO-62 FIXED scripta uppsättning av couchbase-vyer

0.8 — Spelarsimuleringar och övervakning
Release Date: 20/Oct/12 Release Notes
Progress: 🟩🟥 4 of 8 issues have been resolved

0.8.1 — improvements, spelarsimularingar övervakning
Release Date: 26/Oct/12 Release Notes
Progress: 🟥 0 of 1 issues have been resolved

*Figure 6*

### 7.2.5 Second iteration (0.5)

The objective for this iteration was to prepare the game for real game play with real users. Verifying that the game really worked.
The planned new feature were:

- Chat (adds a chat per game)
- End game conditions (change game status when end game conditions are met)
- Event log (all plies logged in the game state)
- Hashed passwords (for user integrity)

#### 7.2.5.1 Chat

The chat messages were stored in a list, one list for each game. The requirements were: it should be clear who sent the message and other users should get the new message automatically if the users has the current game visible on the screen. Each chat message contained the chat message, a timestamp for when the message was sent and the user id of which user sent the message. The chat messages were persisted by game id as a list for simplicity.

#### 7.2.5.2 End game conditions

The end game conditions were ported from the original iPad game, except the game type with limited time. A game ends if there are no tiles left in the tile bag and a player has no tiles left or if there are three consecutive passes.

The game state now had 7 different statuses: Active, EndedByTime, EndedByTurns, EndedByTiles, EndedByPassing, EndedByPlayerResigned and Archived. At this point, only the statuses Active, EndedByTurns and EndedByTiles were used. The other ones were taken straightforward from the iPad implementation except the “player resign” and archived which would be implemented further on. The statuses were set as integers, active is 0 and all other statuses >= 1.

The end game condition verification was added to be run after each of the game moves. It updated the game status if end condition was met. This prevented further moves to be made in the game state because the game state was no longer active. If the status was larger than zero, the game had ended.

#### 7.2.5.3 Event log

In the iPad implementation each move or ply is added to a log, to be able to see what move has been made by which payer and the amount of score acquired.
For the pass and change of tiles move - the implementation is simple, since the move does not inflict the score. The place tile event has to consist of all the compiled words and their score. All the events are stored in the game state as a list in a chronological order.

An example of an event entry for placed words in JSON:

```
{
        type: 0,
        playerId: 1,
        score: 30,
        data: [{text: “word”, score: 30, multiplier: 3 }],
        bingo: false,
        timeStamp: timeStamp
 }
```

#### 7.2.5.4 Hashed passwords

Storing password in plain text would allow system administrators to see the user’s password in plain text, or even worse; if the system is compromised, a mischievous hacker could get access the users’ passwords. To prevent a mischievous person from reading the passwords, the user password should not be stored as plain text. A hash function is supposed to be an asymmetric function, a function that easily transform the password into a hash value. It makes it hard, at least within a reasonable time to do the reverse. The node-password-hash [55] library was used for hashing the users’ passwords.

### 7.2.6 Third iteration (0.6)

The main tasks and new features for the third iteration are

- Avatar image for the user
- Mark the lastly placed tiles
- Summarize the score upon a finished game
- Implement the ability for the user to change its settings
- Make it possible to have other users as friends
- Find a horizontally scalable database and port the database storage

- And some client specific new features that does not inflict the server implementation
- Show which tiles are placed within the current ply
- Display the amount of tiles left in the letter bag
- Show finished games in the view

Since the last features are quite trivial and only affects the client, further descriptions for these are omitted.

#### 7.2.6.1 Avatar Image for the user

Besides the basic user information name, city, country it is common to let the user have a picture as an avatar. As a first attempt, the images were stored on the server, using the user id as an identifier. I.e. user with id 1 has the avatar image 1.png. And with no image set, the user got a default image. The idea was to let the server serve these images and the implementation let the images be retrieved via http via the URL http://server/image/id.png. An addition to the retrieving part, an upload feature also had to be implemented.
But this idea was abandoned in favor of using the web service Gravatar [56], Gravatar offers all the part that was to be implemented. It allows users to reuse their avatar for many web applications. The user registers an account at Gravatar using their email address. Then any service can access the user's avatar via the Gravatar HTTP-API. The API works in the way of hashing the users email address using the MD5 hash algorithm. The avatar is fetched from Gravatar's web servers using the address http://www.gravatar.com/HASH.jpg?s=50, where s=50 defines the image size 50·50. Using an external avatar service reduces the load on the web server and simplifies the implementation; it also speeds up loading times bacause images are loaded from another server. Gravatar also allows for randomly generated default images if the user does not have a Gravatar account.

#### 7.2.6.2 Mark the lastly placed tiles

Since a game can be played without both of the players currently logged in to the server, it is not certain that the user is logged in to see which tiles are placed in the last ply. Hence, a property was added to the game state to indicate which tiles were placed in the last ply. The implementation was a list of tile ids that could be queried upon rendering the game state to mark the tiles using another color or bitmap.

#### 7.2.6.3 Summarize the score upon a finished game

When a game is finished, all the users get a reduction or addition of the total score sum depending on the number of tiles left in their tile racket. If a player finishes by placing all of its tiles, that player gets a bonus equal to all the other players' reduction.

This function was added for each RPC method that can result in a finished game, it also produces corresponding log events.

#### 7.2.6.4 Implement the ability for the user to change its settings

An RPC method was added to the server that allowed the users to change their names and settings. The corresponding interface on the client was also implemented.

#### 7.2.6.5 Make it possible to have other users as friends

As the game is a social game, users want a rematch or play against the same user several times. It is therefore desirable to be able to add users as friends, to be able to challenge them. The friend list was implemented as a list of user ids representing the users who are befriended.

#### 7.2.6.6 Find a horizontally scalable database and port the database storage

Until this iteration, key value store Redis has been used. Redis is not designed to scale horizontally. It can scale vertically by adding read slaves to the master server. For true horizontal scaling, the data has to be divided among the nodes in a database cluster. Indeed, multiple instances of Redis could be used with a DHT implemented in the application server. But then the problem of dynamically scale the number of data nodes remains to be solved since it would require redistribution of the keys upon adding or removing nodes.

*Choosing database*
All databases in [27], [26] and [40] were taken into consideration, when finding an open sourced, fast key value store with dynamic horizontal scaling capability. As the game data essentially just consists of game states for the game and user objects, it is good enough to store the data as key value pairs.
The choice came to Couchbase, which have changed name by the version 1.8 from Membase to Couchbase after a merger of the authors of Membase and CouchOne forming the company Couchbase. Couchbase is a horizontally scalable DHT key value store and in the parlance of the CAP theorem, Couchbase is CP. It is accessible using the widely spread memcached protocol [9], for which there exists at least 2 libraries for node.js, node-memcache and mc. The library used in this implementation is mc [57], as it supports the multi get command, which node-memcache does not (multi get enables requesting multiple keys in one command). Memcached is a widely used as the cache tier, and stores the data in RAM. Couchbase is an extension of memcached, which in addition to storing the data in memory also stores the data persistent to disk using Sqlite [9]. Frequently accessed keys are cached into memory, offering both cache and persistent data storage.
Couchbase is also packaged with a control panel accessible via http. The control panel offers the ability to add or remove new nodes to the database cluster and the cluster redistribute the keys automatically. There is also a REST-API which offers the ability to add or remove

nodes on demand. There is an open sourced community edition and an enterprise version. The enterprise edition offers support and "hot fixes" and is recommended for production use. The license offers the use of two enterprise edition servers in production for free, and the community edition has no such restriction of use.

*Implementing the model for the database*

Since Couchbase is in the CP region of the CAP theorem, it does not support transactions. This may lead to data model inconsistency if keys are related, i.e. all users have a list of game states in which they participate. Adding a game requires at least five data store operations for a game with two players, one for retrieving the new game state key, one for persisting the new game state, and two for each player (one for reading the players game list and one for writing the updated list).

All creations of game state, as mentioned before are handled by one single process, the player matcher process. This will at least avoid the problems that could occur when two processes updates the players' game lists concurrently. But it does not take care of the problem that could occur if the connection is lost after inserting a game state, before the update of the players' game lists. This potential inconsistency, where a game state exists, but is not included in any of the players' lists or in only one of the players list, will not lead to a server crash, but in worst case, inconvenience for the players. And thus, consistency is sacrificed to gain scalability.

All the keys are stored using a prefix for each type of key and the game state and player keys also have a counter to decide which suffix the key will have, Table 1. The Couchbase server offers an atomic increase command that allows for increasing an integer value atomically.

| Key prefix | Description |
|---|---|
| user_ | Used for the user object, stores name, email, city and country |
| userCounter | Integer value used as suffix for the user key. Incremented for each new user. |
| userMail_ | Used as prefix for the authentication object. Stores the user id and the hashed password. The suffix of the key is the users email address in lowercase. |
| chatLog_ | Contains a list of chat messages for a game, the suffix is equal to the game state id. |
| gameState_ | Used for storing the game state. Contains the board, players, rules, letter bag, valid letter for the dictionary, the game state revision, the list of log events, the game state status, a timestamp for when the game state was last altered and consecutive passes. |
| nextGameStateId | Integer value used as suffix for the user key. Incremented for each new game state. |

| | |
|---|---|
| activeGames_ | Used for storing the list of the games in which a user is participating. A list of integers where each integer represents a game state id. The suffix is equal to the corresponding user id. |
| friends_ | Contains a list of user ids which are friends of a user. The suffix is the id of that user. |

*Table 1*

### 7.2.7 Fourth iteration (0.7)

The main goal of the fourth iteration is to get the game playable in the web client.

- Make it possible to make custom board designs
- Make it possible to send a challenge to a friend
- Implement user rating
- Search for other users
- Implement time limit for a ply
- Make it possible to resign
- Optimize the storage of the game state

And some client specific new features that does not inflict the server implementation

- Show connection status in the client
- Disable buttons in the client when other players turn
- Implement modeless notification in the client

In addition the planned feature tasks in this iteration, a lot of time was spent on refactoring. On both the client side and the server side.

- Splitting the client JavaScript file into “modules”.
- Using JavaScript view template for rendering html instead of building strings.
- Using less instead of CSS and dividing the CSS file into files per view area.
- Using server side asynchronous library for simplifying error handling and syntax
- Upgrading to Couchbase 2.0

#### 7.2.7.1 Make it possible to make custom board designs

The original iPad implementation offered the ability to make and use user-designed boards. This was implemented on the client as the ability to draw board squares on a predefined 15x15 matrix. On the server side, a list of maximum of three boards is stored using the key “boards_userId”. The boards would later on be used when sending challenges to other users. Figure 7 shows the client implementation of the board editor.

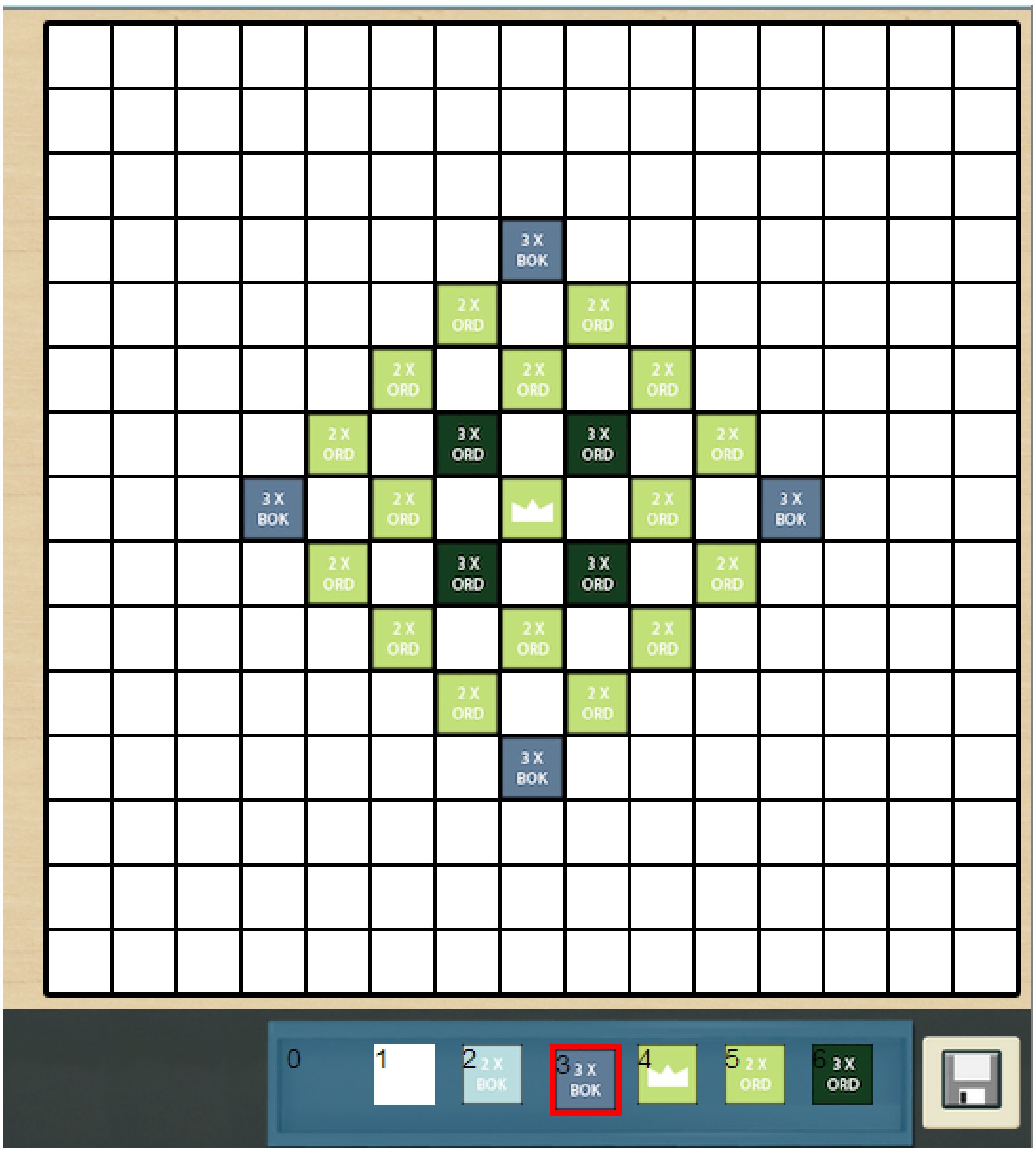


*Figure 7*

### 7.2.7.2 Upgrading to Couchbase 2.0

As this thesis was written there was a new version of Couchbase under development, version 2.0. Instead of using Sqlite as persistent storage, Couchbase uses couchdb, which adds the features of querying the stored data using “views” and map-reduce functions. This new version was scheduled for release in fall 2012 and was released as developers’ preview. The version 2.0 became the choice of the implementation since it offers the map reduce functionality.

### 7.2.7.3 Make it possible to send a challenge to a friend

This feature will allow a user to challenge another user. There are a lot of options to consider when sending a challenge.

- Which board (random, standard or own design)
- Which players to challenge
- Which dictionary to use (for word verification)

The ability to create a new random game and challenge a friend was merged into one single dialog. See Figure 8 for the new game dialog layout.

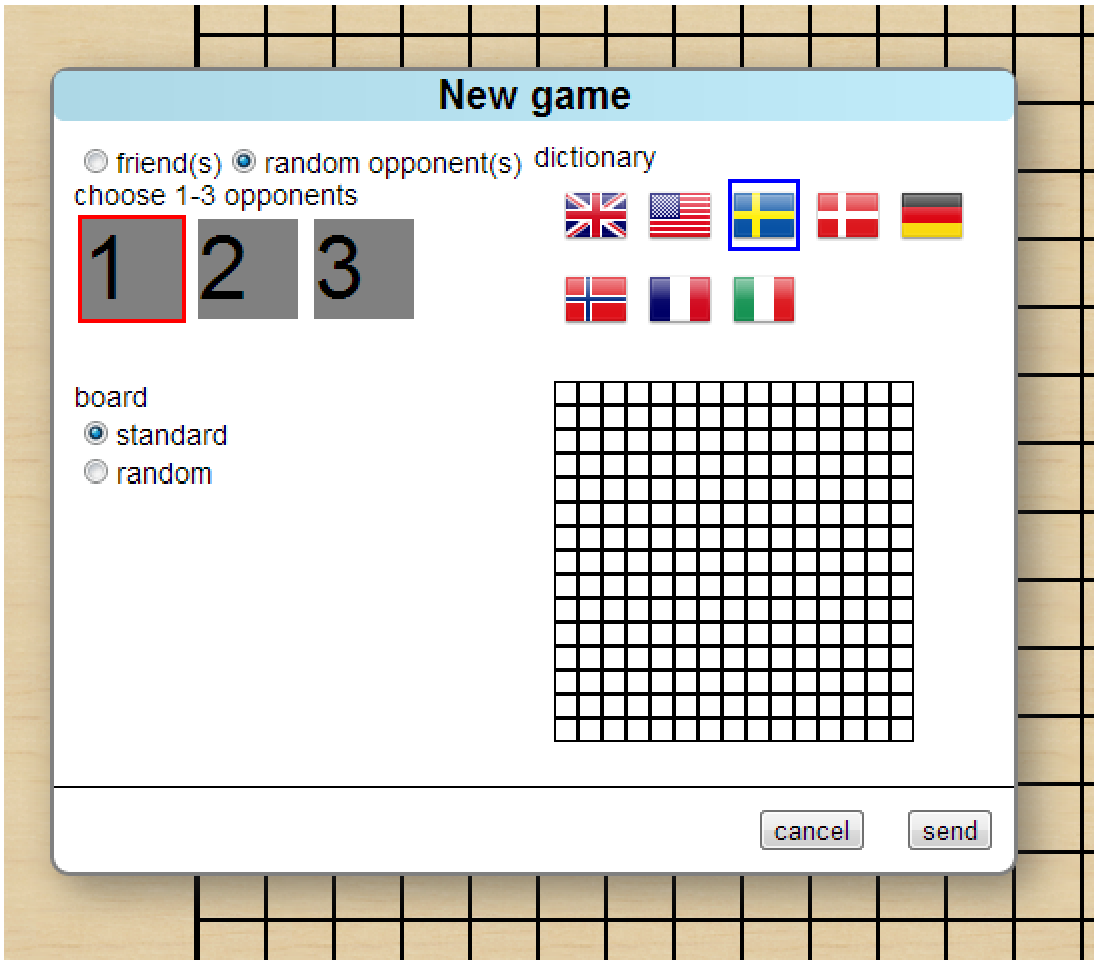


*Figure 8*

During implementation of this feature the complexity of handling lists of challenges for each user became too intricate. This is what led to upgrading the database to version 2.0. The main new feature in the 2.0 version is the possibility to use a map function on the documents stored. See the section about upgrading to Couchbase 2.0 below for more information about map-reduce and Couchbase views.

In this case, the map function is used for each challenge object to emit a key for each challenged user. Instead of maintaining a list of received challenges for each player, the database can be queried for the challenges relevant for a specific user.

#### 7.2.7.4 Implement user rating

User rating is a way of defining how good a player is. The implementation of a good user rating function is out of the scope for this thesis, so a predefined function is implemented which is borrowed from Betapet [58]. All user starts with a rating of 1000, then upon winning or losing a game, the rating will change. Since the implemented rating function will not be a part of the final version, the details are omitted. But it assumed the algorithms would produce about the same workload.

When new rating is calculated, the delta rating and new rating is added as one log event per user for the actual game state.

#### 7.2.7.5 Search for other users

It is not possible to search for data within a key value store; this once more motivates the upgrade to Couchbase 2.0. A view is created for the users.
In the user documents the e-mail and names are emitted as keys. The name is spliced by white spaces so that it is possible to find a user by only searching for surname or last name. To be able to perform better searches a better index is needed, either by emitting all prefixes of the name or using another indexing system. But for this first implementation, it is good enough to be able to search for email and name since the number of users is small.

#### 7.2.7.6 Implement time limit for a ply

Since the game is running even though the users are not present and logged in. It is hard to tell if a user has given up the game and do not care about it anymore. Thus it is required to have a time limit for each ply so that a game cannot continue forever. Once again the views in Couchbase 2.0 is used. A view containing all active games was created and the timestamp for the last move was emitted as a key. This made it possible to sort the games, order by last alteration date.

The code for this was placed in the player matcher process; which was renamed to game manager. The algorithm for this picks the oldest game (the game which has been idle for the longest time), if it has 72 hours has passed, the current player automatically plays a pass. If 72 hours has not passed, the process sleeps for the time until 72 hours for the oldest game has expired and the algorithm repeats.

#### 7.2.7.7 Make it possible to resign

A new game move was implemented. When a user resigns, the game will get status “user resigned” and the game ends. The final scores were calculated for each player and the game was ended and new player rating was calculated.

#### 7.2.7.8 Optimize the storage of the game state

Measuring the length of a serialized game state at the end of the game indicates that the size of a game state can grow to about 15 KB. A lot of space was consumed by the property names, due to their length – for example a tile, {letter:‘L’,score:2,id:2} has the length of 25 bytes. This can be reduced to {l:’L’,s:2,id:2}, which has the length of 16 bytes. This applied for all serialized objects.

The board was modeled as a matrix of squares, where each square was defined as {type: <integer>} and with a placed tile {type: <integer>, tiles: {letter: 'L', score: 2}}. The board was spliced into two properties, one containing the square type, an matrix of integers and a hash map representing the placed tiles i.e. a tile placed at square position 2,3: {'2,3': {l:'L', s:2}}

By reducing the length of the property names to a single character for the most frequent objects (tiles log events) and changing the board model, the size of a game state was reduced to less than 10 KB. The downside of reducing the length of the property is that it can be harder to understand what the property represents. But for a tile the 's' and 'l' are quite obvious.

### 7.2.7.9 Refactoring the client

Since there is only one single global scope in JavaScript, no namespaces and no classes, everything defined is visible in the global scope. One way to hide variables is to use the scope of a function to hide the variable within a function scope and return an object that has access to the function scope. Exemplified below:

```
var obj = function(){
   var _privateVariable = "private";
   var getPrivate = function(){
       return _privateVariable;
   }
   return {
       getPrivate: getPrivate
   }
}();
```

The right hand side of the expression is a function that is invoked just after it is defined. The variable within the function scope is only accessible using the getPrivate-function returned by the function and assigned to obj. The getPrivate-function is accessed through `obj.getPrivate();`

Namespaces can be defined within the global scope by defining a variable, in the game the variable `oko` is used, `oko = {};`

And in each file, representing a module, for instance the chat modules, is defined as:

```
oko.chat = function(){
    var _private1;
    var _private2;
    ...
    var f1 = function(){...};
    …
    return {
        f1: f1,
        ...
    }
}();
```

The client is divided into the following modules, listed in Table 2.

| | |
|---|---|
| main.js | the main file which starts the game |
| oko.buttons.js | The module that handles all the buttons in the |

| | |
|---|---|
| | view |
| oko.challenges.js | Handles challenge, responding and viewing |
| oko.chat.js | Handles the chat, views the chat messages and controls the input |
| oko.editBoard.js | Handles the board editor |
| oko.friends.js | Handles viewing, adding and removing friends. |
| oko.gameState.js | Handles the game states, and renders them |
| oko.games.js | Handles the list of active games |
| oko.localized_en.js | Language file for the english language |
| oko.localized_sv.js | Language file for the swedish language |
| oko.log.js | Renders the log events for a game state |
| oko.menu.js | Handles the menu events |
| oko.moves.js | Handles the moves, place tiles, exchange tiles, pass and resign |
| oko.newGameDialog.js | Handles the events from and renders the new game dialog |
| oko.rpcclient.js | Handles all communications to the server |
| oko.tileChooser.js | Renders the tile chooser dialog for blank tiles |
| oko.time.js | Contains functions for converting unix time to formatted string, also adjusts for time difference between the client and server by comparing local time on the client and on the server. |
| oko.userSearch.js | Handles the search input and renders the search result when searching for users |
| oko.users.js | Handles and caches the user objects so that they do not have to be fetched from the server |
| oko.view.board.js | Handles rendering of the board |
| oko.view.dialog.js | Handles showing and hiding dialogs |
| oko.view.notification.js | Renders game notifications (modeless) |
| oko.view.tileRack.js | Handles the tile rack, placing and rendering tiles. |
| oko.view.tiles.js | Render tiles |
| oko.view.userDialog.js | renders the user dialog, which shows information about users. |

*Table 2*

#### 7.2.7.10 Using JavaScript view template for rendering html instead of building strings.

When inserting new DOM elements into an HTML DOM tree, they can be constructed by HTML strings. For example:

```
var el = '<div class="userInfo"> <div class="name">' + user.name +'</div></div>'
```

This can be quite cumbersome to maintain because when building complex element it is hard to distinguish what is data and what is the view.

By using a view template, the data and the view can be separated. The choice of viewTemplate engine led to the use of handlebars, due to the syntax highlighting support in webstorm. The corresponding handlebar template:

```
<div class=”userInfo”>
  <div class=”name”>
    {{name}}
  </div>
</div>
```

The view template is then compiled, creating a function. The template is rendered by calling `Handlebar.templates.nameOfTemplate(user);` where the user object contains the property name.

This simplifies altering the view with a clear distinction between the data model and the view itself. The templates can be precompiled and served as static content by the web server.

#### 7.2.7.11 Using less instead of CSS and dividing the CSS file into files per view area

Less [61] - “write less CSS” - is a library for writing LESS rules when creating CSS rules. It also offers the usage of variable and mix-ins that simplifies the maintenance of the style sheets. The less files are compiled into CSS files, which are served to the client. The CSS rules were rewritten using LESS and improved maintainability of the layout rules.

#### 7.2.7.12 Using server side asynchronous library for simplifying error handling and syntax

The pattern for writing asynchronous functions in node.js is using the last argument as a callback function. I.e.

```
async(p1, p1, function(err, res){
    if(err) throw error;
});
```

The first argument of the callback function is the error object and the second the result of the function call. If there are a lot of asynchronous function calls performed that has to be in sequence, it gets a bit messy, since each of the function call has to handle error and it is hard to keep track of which scope is which,
i.e.

```
async1(p1, function(err, res){
    if(err) //handle error
    async2(res, function(err, res){
        if(err)) //handle error
        async3(res, function(err, res){
            if(err) ) //handle error
            async4(res, function(err, res){
                if(err) ) //handle error
                async5(res, function(err, res){
                    if(err) ) //handle error
                    async6(res, function(err, res){
                        if(err) ) //handle error
                        callback(null, res)
                    });
                });
            });
        });
    });
});
```

This is where a library for handling asynchronous calls is of use. There are several libraries available in the Npm registry which all offers the same functions with a difference in their syntax. The choice came to choosing a library called Seq, which offers both sequential calls, parallel calls and mapping; also the ability of filtering the result between each call.

The example from above written using Seq

```
Seq()
.seq(async1(p1, Seq)
.seq(function(res){
async2(res, this);
})
.seq(function(res){
async3(res, this);
})
.seq(function(res){
async4(res, this);
})
.seq(function(res){
async5(res, this);
})
.seq(function(res){
async6(res, this);
})
.seq(function(res){
        callack(null, res);
})
.catch(function(error){
// handle the error
}
```

The error handling can be consolidated into one place, at the end of the chain, and the amount of code required to do sequential asynchronous calls is reduce.

#### 7.2.7.13 Upgrading to couchbase 2.0

*New persistent model*

With the release of Couchbase 2.0, the persistent storage is changed from using Sqlite to using the persistent storage of Couchdb. Couchdb is build using the Erlang OTP platform and is designed for lock free concurrency. Couchdb offers full ACID compliancy per document. All document updates are serialized and written to the end of the file when persisting to disk, therefor, no committed data is ever overwritten. This keeps the documents in the database in a consistent state. Every document is versioned with an incremental sequence id that increases with each document update. The read operations uses a Multi-version Concurrency Control (MVCC) model, which permits the reader to always see a consistent snapshot of the database, even if a write operation is performed concurrently on the same document being read. The use of the append only-strategy makes the database crash tolerant, because a crash does not produce an inconsistent state. But the use of append only, will result in waste of space since many versions of each document is stored at the same time. This is taken care of by compacting the database. The compaction creates a new file only containing the latest versions of the documents and then deletes the old file [59]. The compaction can be scheduled or triggered by a fragmentation ratio.

*Views*

The use of apache Couchdb as a persistent storage in couchbase 2.0 also adds the option of indexing the documents into views. A view takes unstructured documents, extract desired properties and information from them and index that information. A view can then be queried to get document containing certain properties or information. To be able to index documents, they have to be stored in the JSON format. The map-reduce functions and are written in java script [60].

E.g. it is desirable to get all the game states for which a certain user participates. Each game state contains a list of players where each player has a user id. With CouchDb, it is possible to create a view for all the active games and use the user ids as keys when writing a map function. The map function emits all the players' user ids for each game. Then the view can be queried, using a user id to get all the active game states for that user id.

```
function(gameState) {
    if(gameState.status == 0){
        for(var i = 0; i < gameState.players.length; i++) {
        emit(gameState.players[i].id, gameState.id)
        }
    }
}
```

Code from the game implementation

The indexing can take long time to perform for large data sets, but once it has been generated, the view indexing is incremental for each document update or document creation [60].

## 7.2.8 Fifth iteration (0.7.1)

The main goal of the fifth iteration was to finalize and stabilize the server code to prepare the game for the simulation phase.

- Localization
- Script for deployment of Couchbase views
- Automate less-compilation, JavaScript minification and view template compilation
- Remaining code refactoring
- Upgrade libraries to latest versions

### 7.2.8.1 Localization

The game was to be released in multiple languages. Two new files were created for the client: oko.localized_en.js (English) and oko.localized_sv.js (Swedish). Depending on the language requested either the en or the sv file is loaded. The files consisted of the “namespace” oko.localized, which contains string properties that are used instead of string literals in the code. E.g. oko.localized.newGame = “New game” in oko.localized_en.js and oko.localized.newGame = “Nytt spel” in oko.localised_sv.js. There were also two corresponding files created for the server, incorporating the server side localization.

#### 7.2.8.2 Script for deployment of Couchbase views

Couchbase offers an administrative user interface for writing the views, but also offers adding views via a rest-API. Adding the views using the user interface was time consuming, but necessary with each deployment. The library used for accessing the views, baseview did not contain any functions for adding and deleted views. This had to be implemented and was also contributed as open source via the open source community to the library. It offered the opportunity to contribute to the “open source community” and try the workflow on GitHub [61]. A node script was written that could be run to add all the views at once.

#### 7.2.8.3 Automate less-compilation, JavaScript minification and view template compilation

When developing and testing, the less files and the view templates were compiled dynamically for each request. For production these files are static and needs to be generated only once. It is also possible to reduce the size of these files by removing white spaces and shorten variable names. This is called minification. Scripts for generating and minifying the static files were created.

#### 7.2.8.4 Remaining code refactoring

Some refactoring was made to make the code easier to read and some parts are extracted into separate modules.

#### 7.2.8.5 Upgrade libraries to latest versions

The final step was to make sure the code was stable for the latest version of all the libraries used. Some rewriting was required due to some library version incompatibility. For instance, the expressjs framework was upgraded from version 2.6 to 3.0, and broke the integration with socket.io, which had to be fixed.

### 7.2.9 Sixth iteration (0.8)

The main goal of the sixth iteration was to implement server supervision, player simulation and automatic scaling. The iteration was divided into the following tasks:

- Implementing the simulations
- Adding measurements
- Setting up the systems / deployments

#### 7.2.9.1 Implementing player simulations

The goal of the player simulations was to emulate real players using player robots; generating workload on the server. They were to log in to the server, create new games and play against each other.

Parameters to consider when creating the robots and running the simulations.

- How often should a player make a move?
- What is a proper move distribution, the ratio of the different kind of moves?
- How many simultaneous games should a player have?
- When placing a word, how many tiles should be placed and where to place them?
- Is it necessary to place real words?

The players were using node.js, just as the server. The players connect to the server using the socket.io client library.

The chosen algorithms for the players are

```
1. Login user
2. If login fails, register user, repeat from 1.
3. Request the games for the user from the server if no games are loaded or reload
is requested.
4. If there are less than the chosen amount of games, send a random game request.
5. If there are no games for the players turn, wait until either other players make
a turn or a new game is created.
6. Make a move
7. Sleep until next move and repeat from 3.
```

In conjunction to the main algorithm there is an event loop that listens to events, sent from the server.

- If an opponent makes a move, a game is added to the queue for the games of which it is the players turn.
- If a new game is created, request a game reload.

**The login procedure**

The first hurdle was to get the players to be able to login to the server. The authentication relies on utilizing a session cookie received after posting the login credentials to the server. A web browser sends the session cookie with every server request, including the socket.io handshake request. When using the socket.io client library, it was not possible to send the session cookie with the handshake since it was not possible to add the cookie to the HTTP header. This would have been the normal behavior of a web browser.

The problem was solved by sending the cookie as an URL parameter, as part of the URL schema. Thus, the socket.io authentication procedure had to be rewritten to make it possible for the robots to login.

One instance of the player simulations application can run multiple instances where each player is identified with a number that is used as a suffix for the player name and email address used for registering the player. I.e. user 1 gets the username user1, password user1, and email user1@simulations.oko. The configuration of a simulation instance consists of the player number interval and time between its game moves. The time between game moves consists of two variables: one for minimum time and one for a random part of the delay.

**Making the moves**

There are essentially three moves that the player can make: place tiles, exchange tiles and pass. The most important move is the place tiles move since it is most frequently used.

*Tile placements*

Two options were considered when implementing the place tiles move: 1) Using a real dictionary and make real moves or 2) allow all letter combinations. The first option would add to the complexity of the simulations as well as the amount of work required for generating the moves. The second options would require altering the server to allow for a word validation that allows every word.

On the server side, the difference between real words and all letter combinations would be negligible because of the usage of the dawg. Also, the word lookup is a small fraction of the work required when verifying tile placements.

The algorithm for creating a tile placement is as follows:

```
1. Decide the maximum length of the word, a uniformed random number between one and
the number of tiles in the players tile rack.
2. Decide whether to place horizontally or vertically randomly, p = 0.5 for
horizontal and p = 0.5 for vertical.
3. Decide where to start the placement by choosing a random placed tile where the
probability of choosing a tile is uniformed.
4. Create a random offset for a maximum number of tiles placed in front of the tile
(uniformly 0 to the number of tiles to be placed).
5. Try to place the tiles adjacent to the chosen tile in the chosen direction.
If it is not possible to place all tiles, the maximal number of possible tiles will
be placed.
If no tiles can be placed with the given tile and direction, the simulation will
produce a pass move instead.
```

This will only render valid placements. But since the amount of calculations for valid and invalid moves are roughly the same, the case of generating invalid placement is not considered.

*Exchange tiles*

The algorithm for exchanging tiles is as follows:

```
1. Decide how many tiles to exchange a uniform random number between one and the
number of tiles in the tile rack.
2. Exchange the tiles.
```

*Pass*

The implementation of the pass algorithm was just to send a pass request to the server.

### 7.2.9.2 Implementing measurements

At the client side it was possible to measure the time elapsed for making a move by recording the time $t_{before}$ before the move request was sent to the server and the time $t_{after}$ when the response was received. The time elapsed is $t_{after} - t_{before}$. It was also possible to calculate the number of moves per second all the players in a simulation instance performed.

These measurements were implemented and recorded by a configurable interval, i.e. every minute or every fifth minute. The average was calculated by summarizing all of the response times and divide by the total elapsed time.

### 7.2.9.3 Testing the simulations

At first, when testing the simulations for stability verification, it was noted that, after running for a while with a constant number of players, the response times increased and eventually the server ran out of memory. The frequency of the database operations increased without increasing the number of players. The cause to this increase was located to the game list implementation. Each player had a list of games for which the player was participating. When a game was created, it was added to each participating. But there were no process responsible for removing games from the lists when they were finished. Each time a player requests the list of active games, the server had to fetch all game states associated with that list to create a list of game states. When the lists grew longer and longer, more works was needed to render these lists, of mostly finished games.

Removing the finished games from the game lists could have solved this problem. But this would have needed to be an atomic operation: fetching the list, removing the game id and saving the list. Another solution was to utilize the map function in the database, letting the database map the players for each game state. This would solve the consistency problem with the players' game lists and the game states.

The database offers an "eventually consistent" model for the views. Thus there is no guarantee that the views of the game states are consistent with the game states in database at all times. But as default behavior the database updates the view every fifth second or when 5000 changes have been made. The query could also be set to not allow a stale view, requiring that all documents are indexed before the response is rendered. This behavior will result in a poorer performance and is not recommended [60].

The transition to view queries solved the problem with increasing number of database operations. It also reduced the complexity of handling the users' game lists.

Then another problem was discovered: the rate of the game moves decreased over time. The source of this problem was identified to the players were not notified correctly when their opponents had made their move. This problem was solved by reloading the game list if the player had no current games where there is the player's turn or had not received a game update within 10 move iterations.

After these issues were solved the simulations was running for 24 hours and was able to maintain a stable move frequency, a stable memory allocation and a stable number of database operations per second.

### 7.2.10 Measurements

The simulations were used to measure the response time from the server as the number of players increased. The number of players was increase by 500 at each measurement and the time used for measuring was five minutes. The measurement implementation is described in 7.2.9.2 (Implementing measurements). At the first measurement iteration, the scalability was tested by measuring the different response times at different settings. The measurement method was evolved during the measurement process to incorporate the methods described in 4.2 (measuring scalability).

At first the server setup consisted of one application server instance and one database server instance.
The instances were run as virtual machine using the Ipeer elastic cloud. The database instance were configured with 2 CPU cores, 10GB of persistent storage and 1GB of RAM running Ubuntu 12.04 64-bit. The application server and simulation instances were configured with one CPU core, 1.5GB memory and Ubuntu 12.04 32 bit. The database running on the database server was Couchbase 2.0 beta 64 bit and the Node.js version on the application server and player simulation was 0.8.14. Figure 9 illustrates the setup.

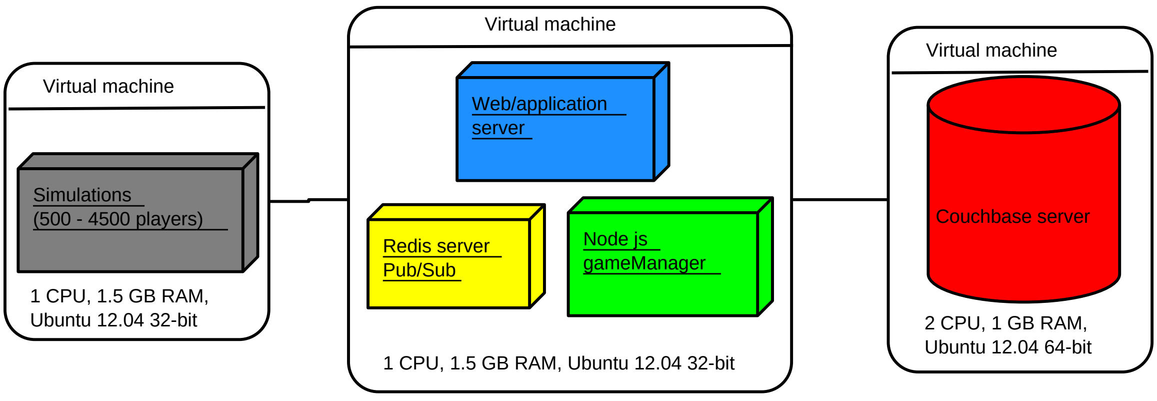


*Figure 9. The setup for measuring scalability using one web server instance*

#### 7.2.10.1 The view query problem

After running the simulations, a problem with the retrieval of the game state was discovered, the view questions took more than 10 seconds with more than 3000 simultaneous players. This problem was discovered when logging in to the game during player simulations running.

A separate measurements for receiving the game state was implemented to measure how the view query time increases with the increased number of players, this is defined as the game fetch time. The response time for the moves is simple defined as the response time.

The simulation setup was able to reach 4500 simultaneous players and crashed before reaching 5000 players because the process ran out of memory.

The game began to be slow and sluggish with more than 3000 players and the time to perform view queries dramatically increase between 3000 and 3500 players. The result is presented in Table 3, Figure 10 and Figure 11.

| Number of players | Game moves /second | avg. response time | max response time | min response time | Avg. game fetch | min game fetch | max game fetch |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 500 | 14.41 | 43.5 | 251 | 4 | 26.3 | 11 | 65 |
| 1000 | 34.2 | 46.6 | 243 | 3 | 38.3 | 11 | 113 |
| 1500 | 54.4 | 64 | 665 | 4 | 52.5 | 13 | 393 |
| 2000 | 73.85 | 200.7 | 1255 | 4 | 203.5 | 15 | 1247 |
| 2500 | 92.7 | 456.6 | 1580 | 12 | 456.6 | 22 | 1594 |
| 3000 | 104.2 | 1396 | 2891 | 154 | 3635 | 464 | 8602 |
| 3500 | 107.2 | 1434.6 | 2971 | 281 | 55374 | 40865 | 69141 |
| 4000 | 103.1 | 2065.4 | 4179 | 402 | 119703 | 110894 | 125209 |
| 4500 | 96.8 | 2737.8 | 4905 | 965 | 196234 | 192117 | 201732 |

*Table 3. The result of measuring scalability using one web server instance*

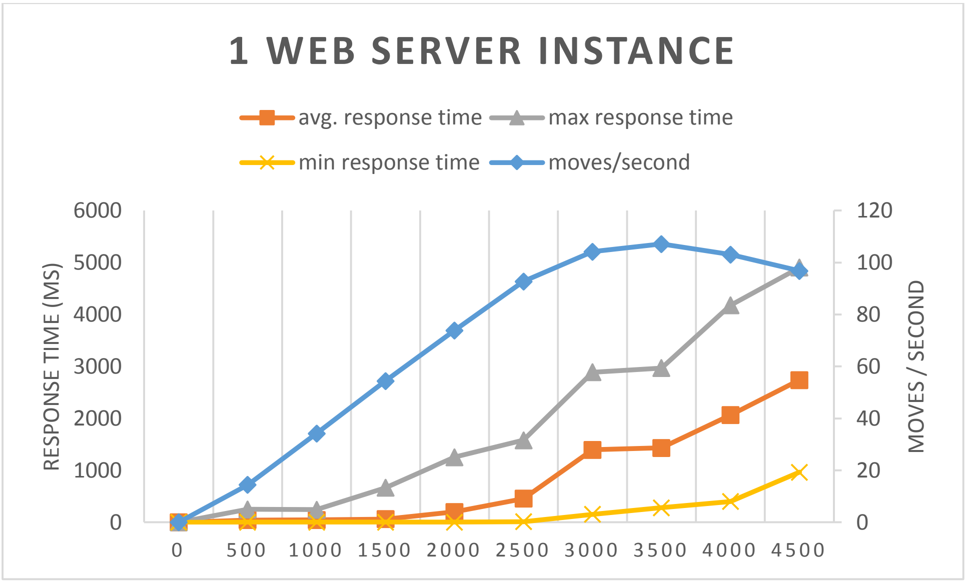


*Figure 10. The response time for game moves with one web server instance*

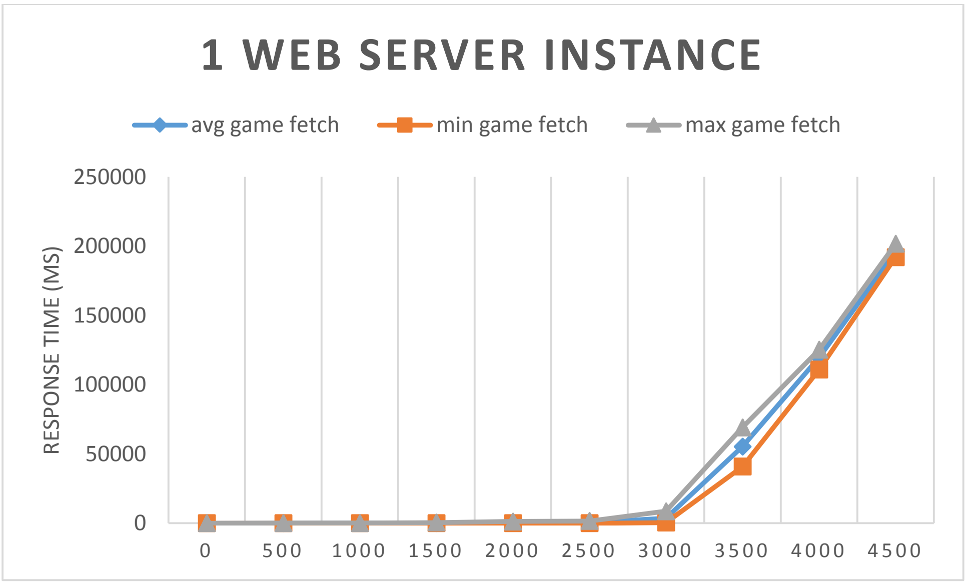


*Figure 11. The response time for fetching player games using one web server instance*

The reason for the drastically increased time for the view queries may be have been caused by a number of factors. Two of them stated below:

1) It may have been because the CPU on the web server instance was saturated and the operations for creating HTTP requests for the view engine got starved.
2) It may be because of the load on the database server, because of map function that renders the view

The first reason could be identified by adding an extra CPU (1a) to the web server instance or adding a second web server instance (1b). An extra CPU would add extra computational power to system calls for the operating system, despite the single threaded event loop nature of node.js.

If it is the second cause, adding an extra CPU or application server instance would not result in better performance.

#### 7.2.10.1.1 Measurements using an application instance with 2 CPUs (1a)

Figure 12 illustrates the measurement setup, using two CPUs for the web server instance.

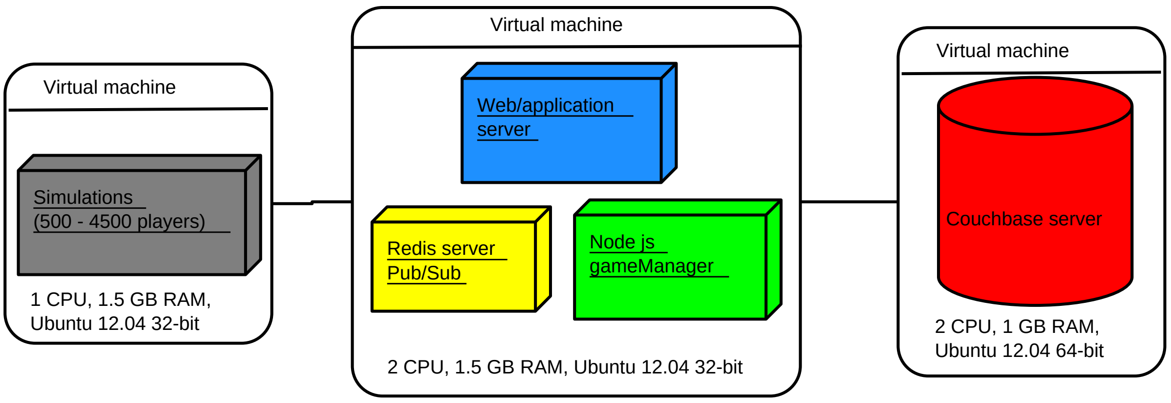

Virtual machine
Simulations (500 - 4500 players)
1 CPU, 1.5 GB RAM, Ubuntu 12.04 32-bit
Virtual machine
Web/application server
Redis server Pub/Sub
Node js gameManager
2 CPU, 1.5 GB RAM, Ubuntu 12.04 32-bit
Virtual machine
Couchbase server
2 CPU, 1 GB RAM, Ubuntu 12.04 64-bit

*Figure 12. The setup of the measurements using a web server instance with two CPUs*

There was an improvement of the response times for the view queries of about half of the response time. But with more than 3500 simultaneous players, the response time was still unacceptable. The response time reduction did confirm the theory of that the limitation lies within the web server instance. The measurements were performed for a player count of 2500 to 4500 players. The result is presented in Table 4, Figure 13 and Figure 14

| Number of players | moves/second | avg. response time | max response time | min response time | avg game fetch | min game fetch | max game fetch |
|---|---|---|---|---|---|---|---|
| 2500 | 96.67 | 205.92 | 7 | 1 | 192.86 | 11 | 1117 |
| 3000 | 114.98 | 450.96 | 2484 | 5 | 424.21 | 15 | 2674 |
| 3500 | 129.45 | 953.49 | 2955 | 111 | 1024.86 | 27 | 3152 |
| 4000 | 124.43 | 5934.94 | 53390 | 9 | 7494.35 | 21 | 23515 |
| 4500 | 131.96 | 3776.1 | 13329 | 316 | 44669.16 | 13340 | 70719 |

*Table 4. The result of the measurements using one web server instance with two CPUs*

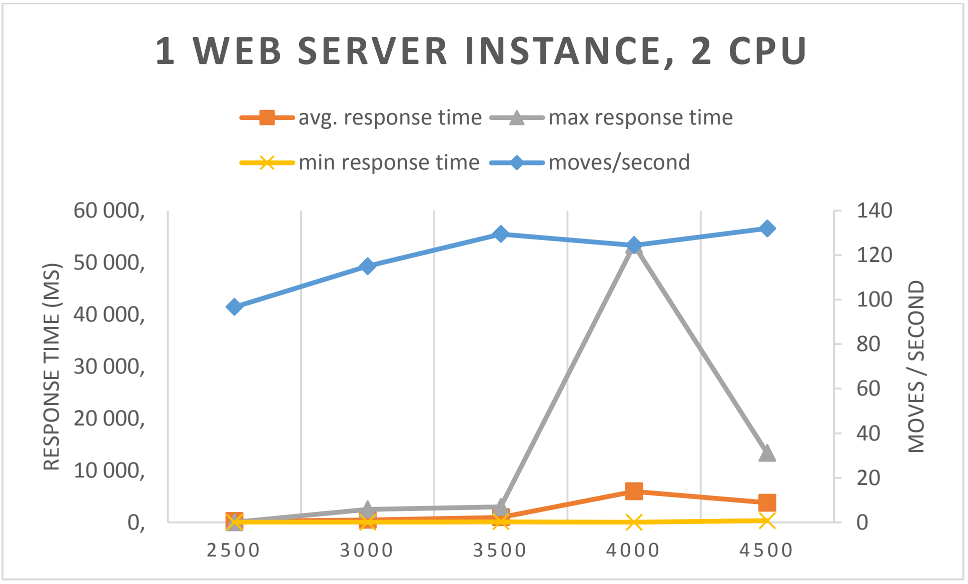


*Figure 13. The game move response times using one web server with two CPUs*

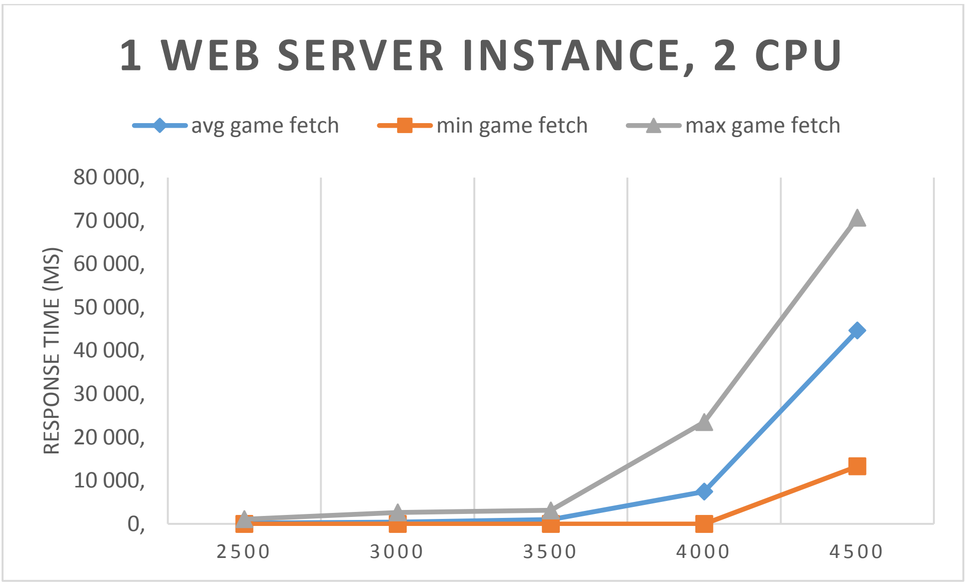


*Figure 14. The response time of fetching the player's games using one web server instance with two CPUs*

#### 7.2.10.1.2 Measurements 2 web server instances

Figure 15 illustrates the setup for the measurement with two web server instances.

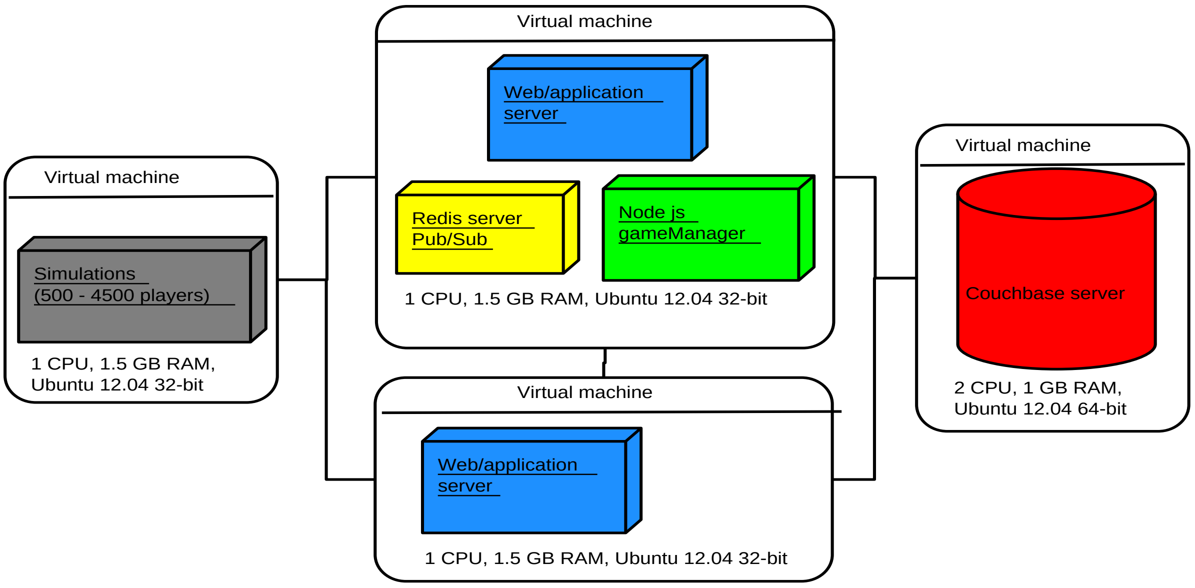

Virtual machine
Web/application server
Redis server Pub/Sub
Node js gameManager
1 CPU, 1.5 GB RAM, Ubuntu 12.04 32-bit
Virtual machine
Simulations (500 - 4500 players)
1 CPU, 1.5 GB RAM, Ubuntu 12.04 32-bit
Virtual machine
Couchbase server
2 CPU, 1 GB RAM, Ubuntu 12.04 64-bit
Virtual machine
Web/application server
1 CPU, 1.5 GB RAM, Ubuntu 12.04 32-bit

*Figure 15. The setup when using two web server instances to identify the view query problem*

When using 2 web server instances, distributing the connections evenly between the web server instances, the problem with the prolonged view query response times was solved. The average response times were below one second, confirming that the problem was not caused by the database. The measurements are presented in Table 5, Figure 16 and Figure 17

| Number of players | moves/second | avg. response time | max response time | min response time | avg game update | min game update | max game update |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 500 | 13.98 | 44.34 | 108 | 2 | 17.09 | 9 | 60 |
| 1000 | 31.68 | 44.34 | 156 | 2 | 23.75 | 8 | 65 |
| 1500 | 53.43 | 45.94 | 335 | 2 | 33.78 | 9 | 273 |
| 2000 | 69.28 | 75.22 | 477 | 3 | 66.41 | 9 | 375 |
| 2500 | 92.57 | 91.59 | 616 | 3 | 85.3 | 9 | 561 |
| 3000 | 112.5 | 123.17 | 890 | 4 | 116.54 | 10 | 1227 |
| 3500 | 131.67 | 223.77 | 3738 | 3 | 224.76 | 10 | 3329 |
| 4000 | 145.55 | 780.39 | 11388 | 4 | 551.3 | 9 | 4148 |
| 4500 | 163.84 | 767.33 | 3307 | 3 | 537.46 | 14 | 2790 |

*Table 5. The result of the measurements when using two web server instances.*

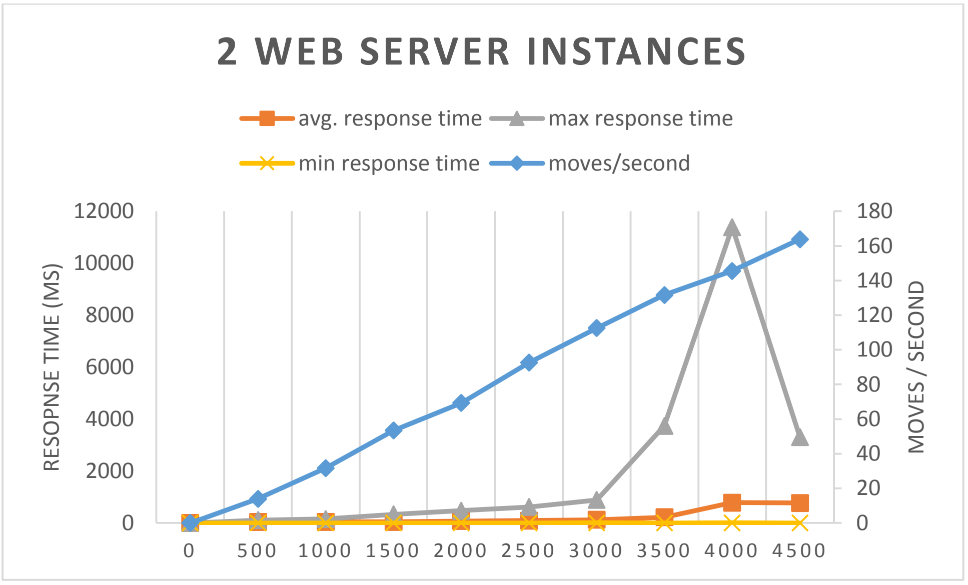


*Figure 16. The response times of the game moves when using two web server instances*

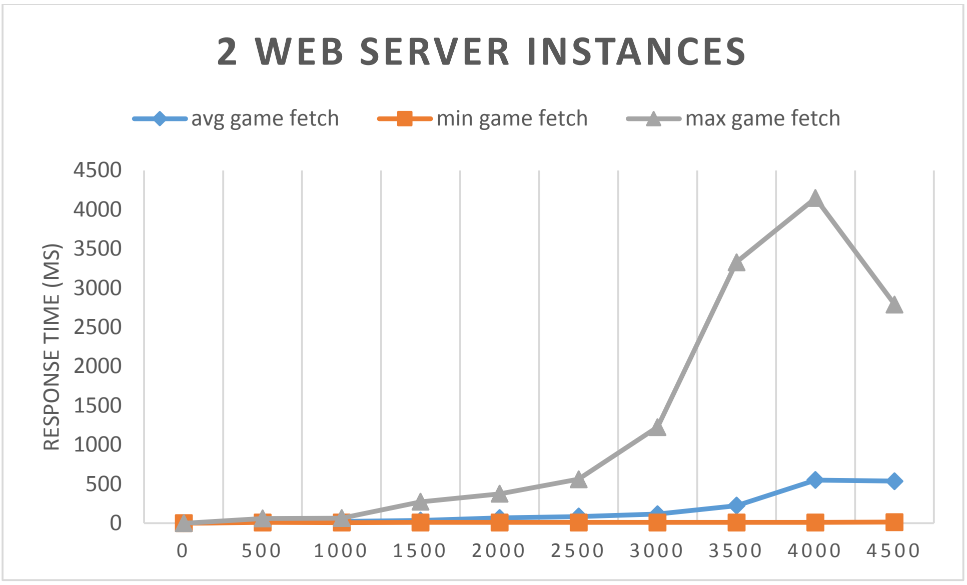


*Figure 17. The response times of the player's game fetch operation*

Comparing the result of using just one web server instance to using one webserver instance with 2 CPUs and using 2 web server instances shows that the query response times are improved when scaling the web server instance as shown in Figure 18.

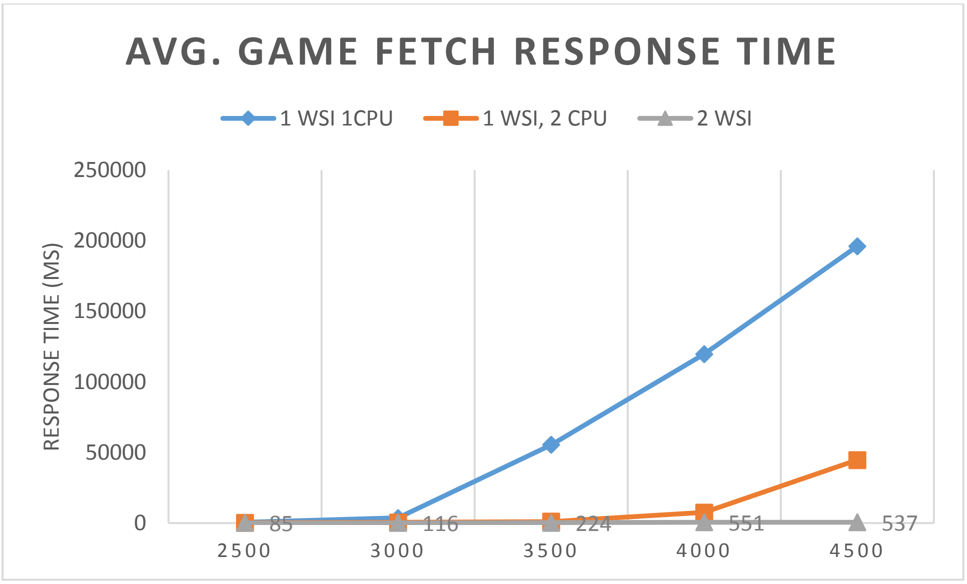


*Figure 18. The player's game fetch operation using one or two web server instances (WSI)*

Another problem was discovered when performing these measurements, discussed in the next section.

### 7.2.10.2 Memory consumption problem

During the measurements, the response times peaked at over 10 seconds at 4000 simultaneous players, but at 4500 players, the response peaked at about 3 seconds. This indicated some problems with the stability of the application or the platform.

To monitor the application during a longer session, profiling was added to the server. Measurements chosen where number of connected users, memory consumption, CPU utilization, game moves per second and response time (time spent from request reaches the server until the response leaves the server). Running during one hour revealed that there are occasions when there are prolonged response times of several seconds. The setup was the same as the first measurements, illustrated in Figure 9. Figure 19 shows a prolonged response time at about 46 minutes run, the maximum response time peaks at 7 seconds.

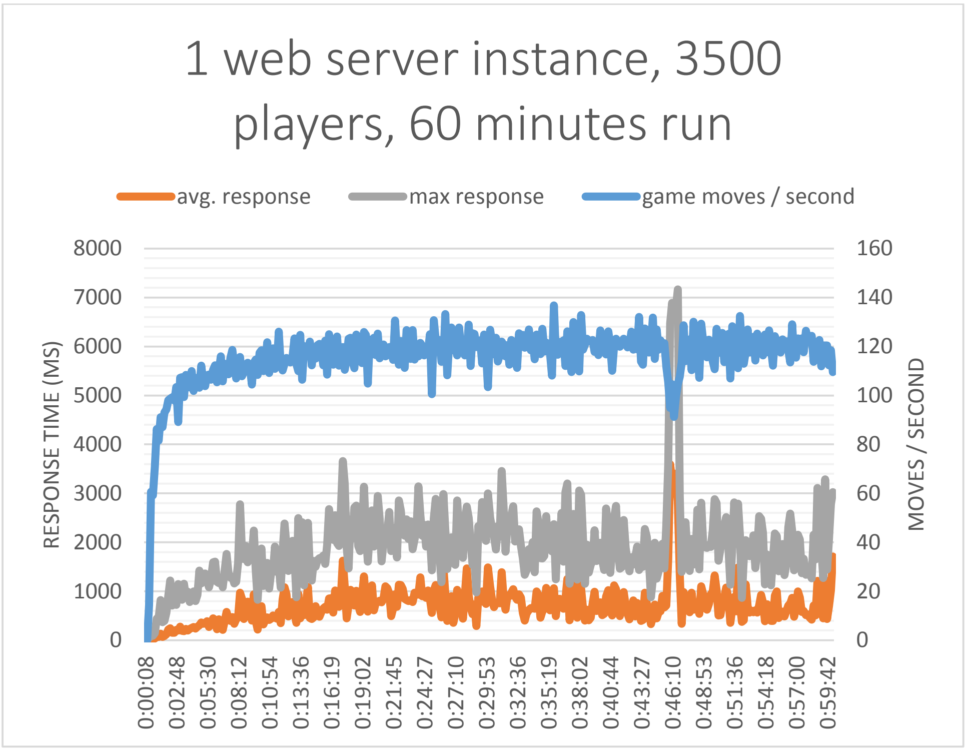


*Figure 19. The game move response and moves per second during one hour.*

The memory measurements were conducted using the process.memory function in the node.js API. The RSS (Resident set size) part shows the allocation of memory that resides in RAM. The memory allocation reported as RSS is not necessarily the amount of memory used by the node.js process, as it also includes memory that has not yet been garbage collected. There are no obvious changes in the memory consumption at 46 minutes where the increased response time occur, besides a slight drop of the memory consumption, see Figure 20.

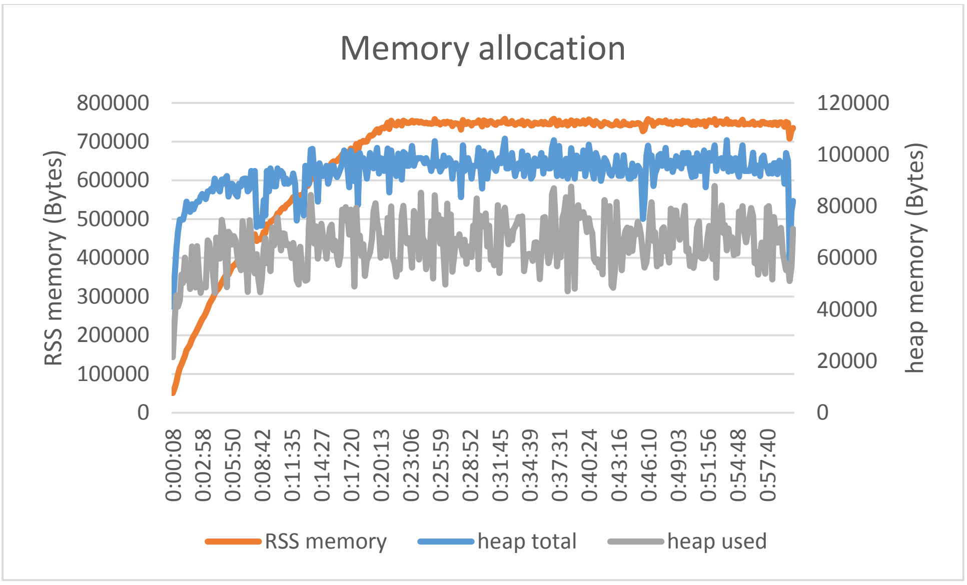


*Figure 20. Shows the memory allocation reported by Node.js during one hour of operation*

When looking at the CPU load, there is also a slight CPU increase at the time of the increased response time. Otherwise, there is no obvious relation between the response time and CPU load, see Figure 21.

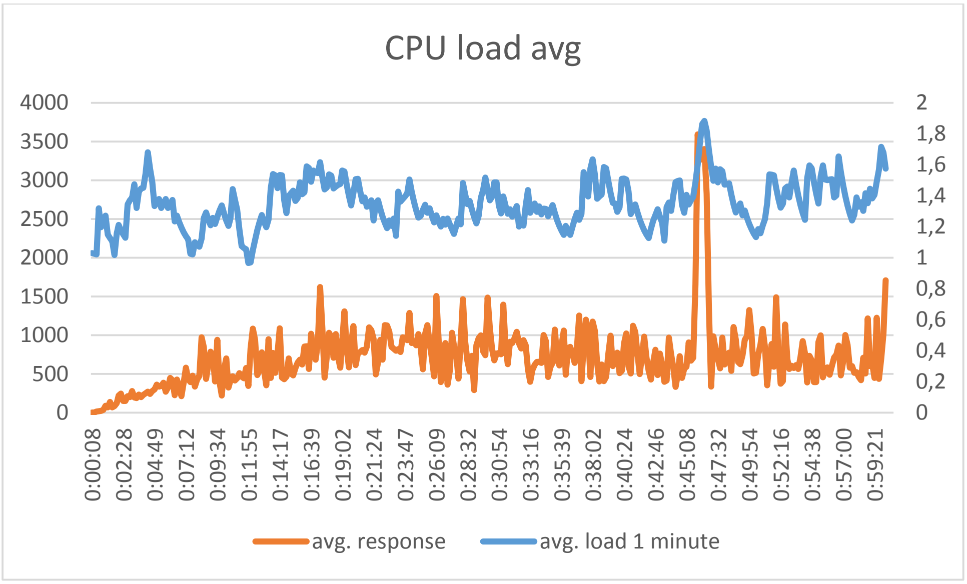


*Figure 21. The CPU utilization during one hour of game play.*

There were two main theories about what caused these hiccups:

1) Garbage collection. If the application uses a lot of memory, the garbage collection will run more frequently and as the less memory is free, the longer time the garbage collection will take. As shown by the measurements, the RSS memory consumption seems to be at a constant maximum.

2) Cache misses in the database. If a lot of active data does not fit the ram on the database server, the data will have to be read from disk and result in longer response times.

To investigate if garbage collection was the cause of the problem, the application was run with garbage collection traced enabled (using the `--trace-gc` parameter when starting node.js). It  was run for 10 minutes to measure the time taken for a garbage collection event. The trace, Figure 22, revealed no occasions where one single garbage collection alone took more than 260 ms (when garbage collection was triggered because of out of memory). One occasion at 260 ms alone would not prolong the response with several seconds, but many occasions in combination with cache miss of the database cluster could sum up to seconds of response time.

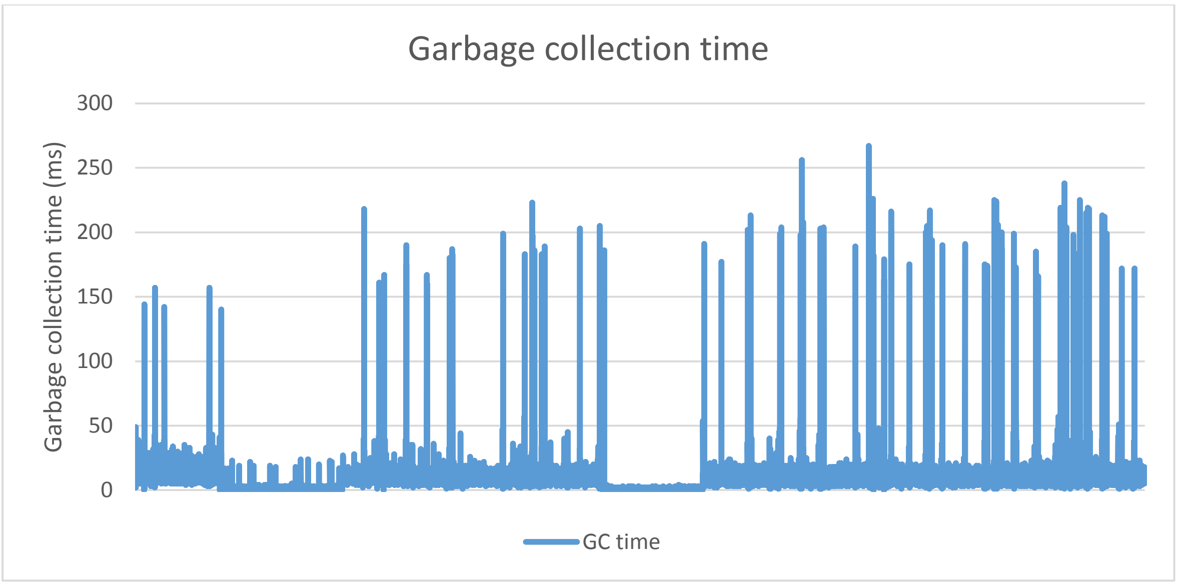


*Figure 22. Shows the garbage collection times during 10 minutes of game play.*

To further analyze the application, Nodetime [62] was used to understand where the application spends time. Nodetime is an online node.js profiler that is installed by adding a module to the node application. The measurements are uploaded to a Nodetime web server and can be analyzed using a web interface. It measures database response times and profiles the execution paths. It was noted that a lot of time was spent on serializing/deserializing JSON, as part of the database connector about 30% of the time.

But when further tests were conducted, another error was discovered within the database client MC. The client was prone to stop working after a while on heavy load. It also lacked of a viable error handling, no errors events were triggered upon error. This was regarded as a severe error and the database client was changed to memcached [63], which made it possible to catch the errors upon their occurrence. When using the new database client, the source of the instability was located at the condition when the dedicated RAM for the bucket in the cluster was full and all data had not been persisted. When the dedicated RAM is full, the database ejects older keys to give room for new keys. It only ejects keys that have been persisted to disk. If there is no dedicated RAM left in couchbase, and the database is not able to eject any keys, it returns a temporary error. This is probably the case what happened when the client MC stopped responding as no errors were handled. With the new client, the errors are propagated to the web server instance.  A separate test was written as a node.js application to fill the bucket and see if the errors were properly handled. It turned out that the errors were reported when they occurred, but some requests were dropped, without any reply. This is most likely the same case as with the MC client. The solution to the problem is to have more dedicated RAM to each node in the database cluster so that it will be able to accept data without any temporary failures.

When the database client issue was solved, the work continued trying to fix the hiccup problem. The task was to reduce the size of the game states, to reduce the time spent serializing/deserializing JSON. It would also reduce the amount of data sent between the database cluster and the web server instance, as well as the memory footprint in both the database and the web server instance, hence, perhaps solving the problem.

The size of the game states was reduced by lowering the number of properties in the serialized game state by utilizing lists instead of properties. The structure of a tile was {"id":77,"l":"A","s":1}, 23 Bytes, this could be reduced into a list construction of three elements: [77, A, 1], 8 bytes. There are 100 tiles in a game state which is 2300 bytes, the new construct reduced the size of the tiles by (23-8)*100 = 1500 bytes. Then, the log entries were separated from the game states. Until now, the log entries were stored in the game state, allocating up to 7 KB. None of the game state alteration really needed the list of log entries. The log entries were stored as bundles of 10, using the append function (one key contains 10 entries). This would save the overhead of the key sizes.

Before the size reduction, the average size of a game state was 3.5KB, the minimal size 3 KB and the maximal measured size was 10.5KB. After the reduction the average size of a game state was: 2.4 KB, the minimal size 2.1KB and the maximum size 2.8KB. The new maximum size of a state was reduced by over 70%.

After these optimizations of the game state, new measurements were performed. And no prolonged response times were measured. The server was able to handle 1000 additional players before it ran out of memory. The measurement shows a more stable curve when increasing the number of players. There is still a problem with the view queries when going beyond 3500 players, but the average response times are good enough for 5000 simultaneous players. See Table 6, Figure 23 and Figure 24

| number of users | game moves per second | max Response | min Response | avg Response | get games per second | Game fetches max | Game fetches min | Game fetches max |
|---|---|---|---|---|---|---|---|---|
| 500 | 16 | 166 | 2 | 9 | 4 | 235 | 4 | 14 |
| 1000 | 32 | 425 | 2 | 9 | 7 | 219 | 3 | 15 |
| 1500 | 51 | 647 | 2 | 17 | 10 | 369 | 4 | 22 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 2000 | 70 | 529 | 2 | 61 | 11 | 714 | 4 | 109 |
| 2500 | 91 | 1052 | 2 | 102 | 7 | 704 | 4 | 106 |
| 3000 | 110 | 1214 | 2 | 149 | 7 | 1152 | 6 | 148 |
| 3500 | 128 | 1436 | 2 | 291 | 7 | 1178 | 5 | 249 |
| 4000 | 146 | 3483 | 4 | 603 | 6 | 3790 | 19 | 539 |
| 4500 | 161 | 4342 | 2 | 949 | 8 | 4718 | 16 | 911 |
| 5000 | 172 | 5235 | 221 | 1594 | 9 | 7827 | 72 | 2271 |
| 5500 | 164 | 8394 | 459 | 2419 | 10 | 19788 | 154 | 7098 |

*Table 6. The result of the measurements after the game state size reduction*

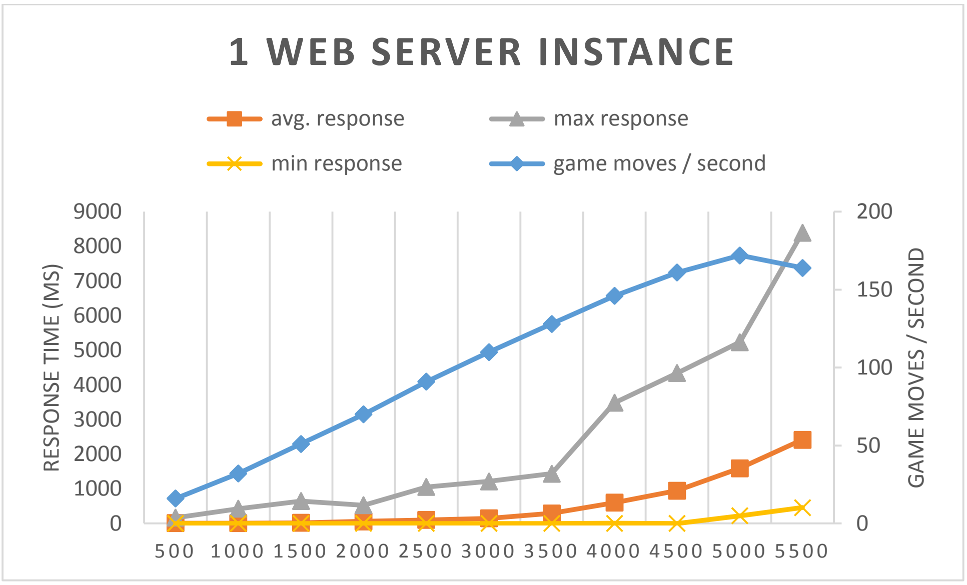


*Figure 23. The response times using one web server instance after the game state size reduction*

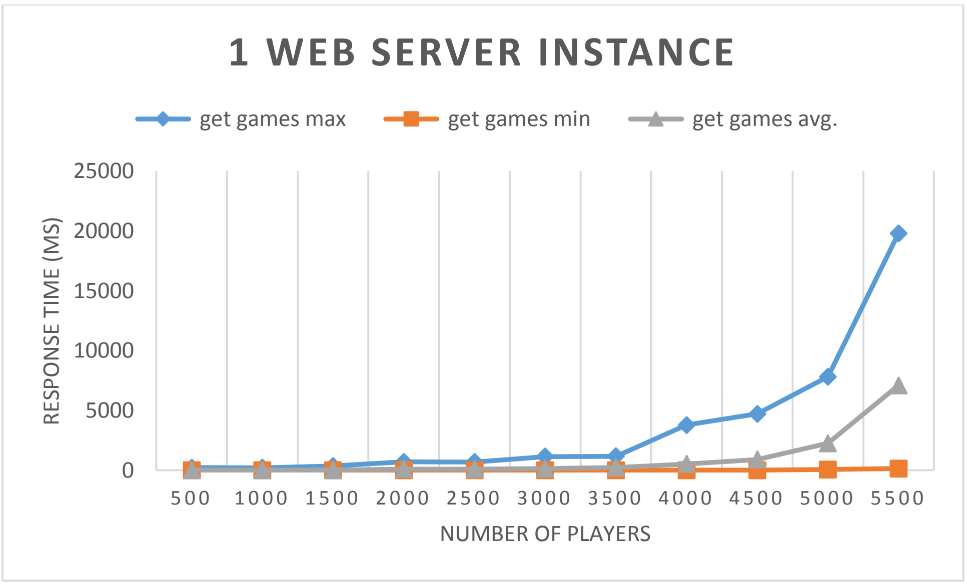


*Figure 24. The response for the player's game fetches using one web server instance after the game state size reduction*

Comparing the throughputs in game moves per second, the throughput increases at the same rate until 3000 players, then the old implementation reaches the maximum throughput. The new implementation hits the maximum throughput at 5000 players, with about the same response times as the old implementation at 3000 players, Figure 25.

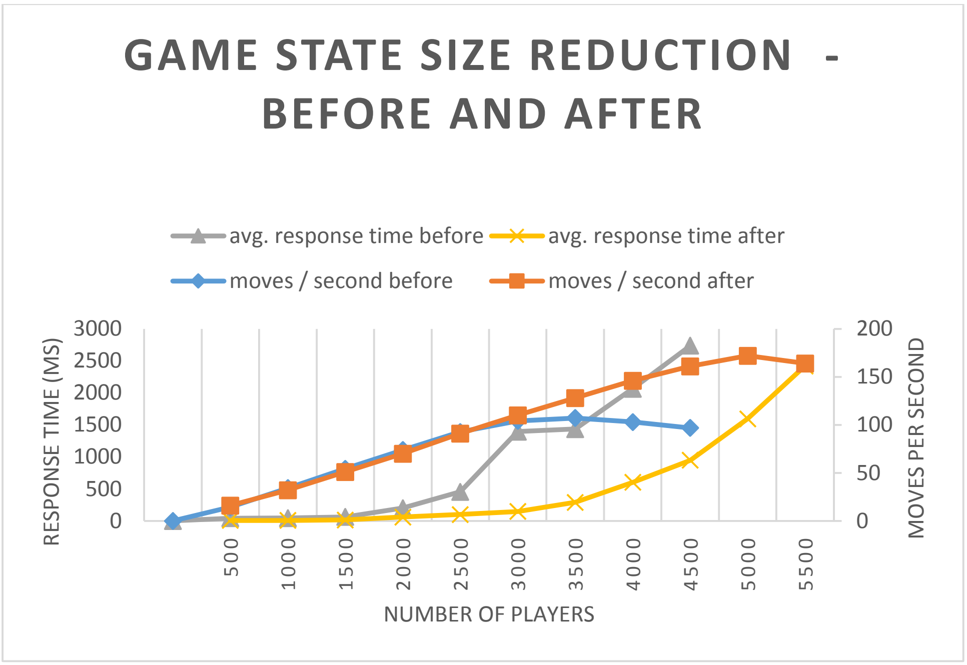


*Figure 25. Response times of before and after the game state size reduction.*

The view queries are also improved, the new implementation tops at an average response time of 7 seconds at 5500 players. With the old implementation, such response time were produces at somewhere between 3000 and 3500 players, see Figure 26.

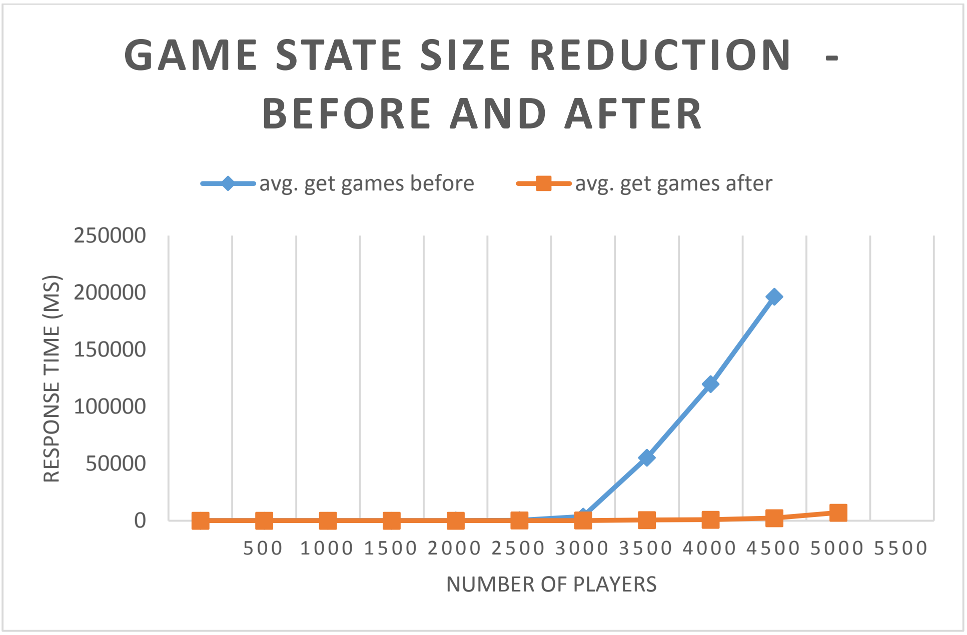


*Figure 26. The player's game fetch response time, before and after the game state size reduction*

### 7.2.10.3 Measuring the application tier scalability

When the application was considered stable, the same measurements were conducted while scaling out the number of web server instances. The measurement were performed with 1 to 5 application servers, but only using one instance for the simulating 500-5000 players. Figure 27 illustrates the setup.

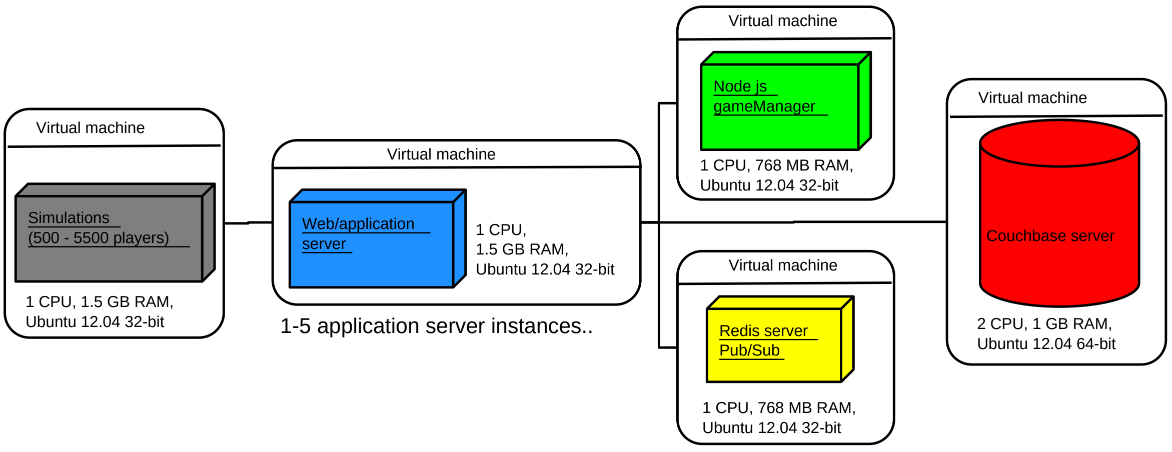

Virtual machine
Simulations (500 - 5500 players)
1 CPU, 1.5 GB RAM, Ubuntu 12.04 32-bit
Virtual machine
Web/application server
1 CPU, 1.5 GB RAM, Ubuntu 12.04 32-bit
1-5 application server instances..
Virtual machine
Node js gameManager
1 CPU, 768 MB RAM, Ubuntu 12.04 32-bit
Virtual machine
Redis server Pub/Sub
1 CPU, 768 MB RAM, Ubuntu 12.04 32-bit
Virtual machine
Couchbase server
2 CPU, 1 GB RAM, Ubuntu 12.04 64-bit

*Figure 27. Scaling out the number of web server instances*

There were no great improvement in the throughput. There is a slight improvement when going from one instance to two instances. But no improvement at all when going beyond two, see figure Figure 28.

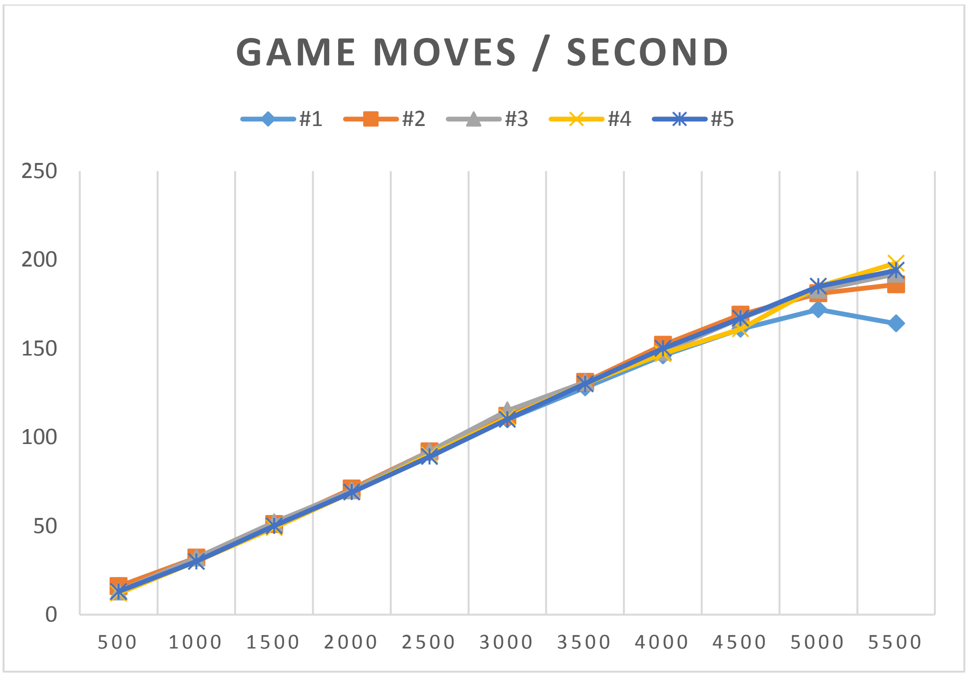


*Figure 28. The number of game moves per second when scaling out the number of web server instances at different number of simultaneous players.*

The response times show the same result as in the throughput, see Figure 29. This indicates that the there is a bottleneck in the setup. The processor load of the instance running the player simulations were measured to 100% indicating that the simulations itself was the bottleneck. The simulators had to be spread among more than one instance to be able to measure the performance of a scaled cluster of application servers.

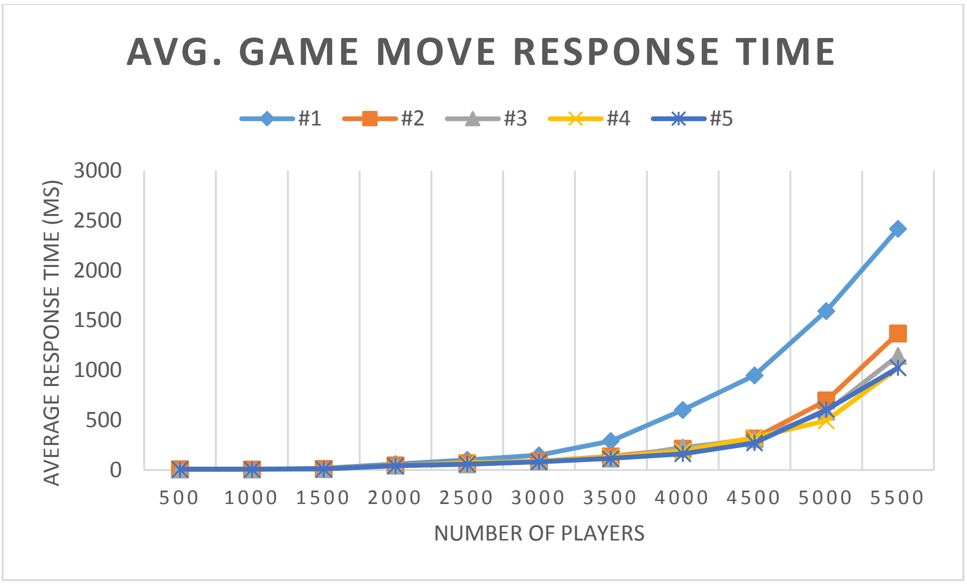


*Figure 29. The average response time when scaling out the number of web server instances.*

#### 7.2.10.4 Identifying the simulation thresholds

To test the limit of for how many players a single player simulation instance could simulate, a test with 4 web server instances, a fixed number of players (5500) and varying number of player simulation instances. The number of players was evenly distributed among the player simulation instances as well as the web server instances. Figure 30 illustrates the configuration.

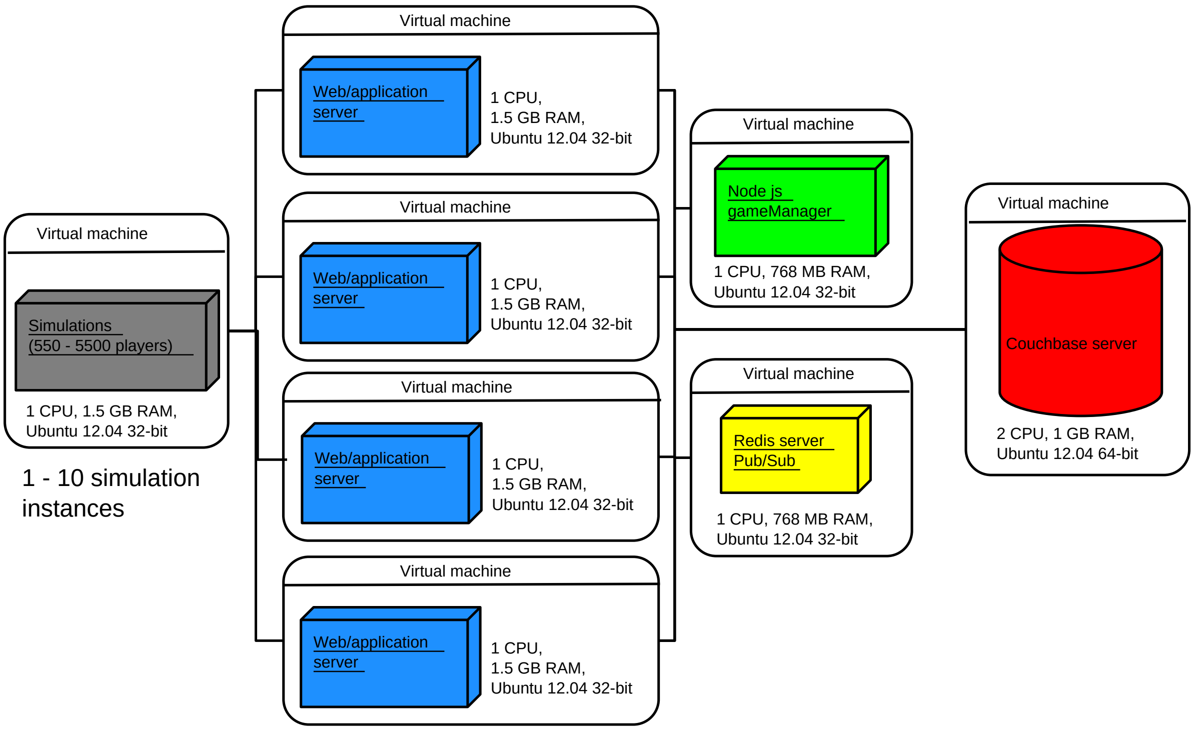

Virtual machine
Web/application server
1 CPU, 1.5 GB RAM, Ubuntu 12.04 32-bit
Virtual machine
Web/application server
1 CPU, 1.5 GB RAM, Ubuntu 12.04 32-bit
Virtual machine
Web/application server
1 CPU, 1.5 GB RAM, Ubuntu 12.04 32-bit
Virtual machine
Web/application server
1 CPU, 1.5 GB RAM, Ubuntu 12.04 32-bit
Virtual machine
Simulations (550 - 5500 players)
1 CPU, 1.5 GB RAM, Ubuntu 12.04 32-bit
1 - 10 simulation instances
Virtual machine
Node js gameManager
1 CPU, 768 MB RAM, Ubuntu 12.04 32-bit
Virtual machine
Redis server Pub/Sub
1 CPU, 768 MB RAM, Ubuntu 12.04 32-bit
Virtual machine
Couchbase server
2 CPU, 1 GB RAM, Ubuntu 12.04 64-bit

*Figure 30. Scaling down the number of players per player simulation instance.*

The result is presented in Table 7.

| Number of players | moves/second | max response time | avg response time | get games avg |
|---|---|---|---|---|
| 5500 | 164 | 8394 | 2419 | 7098 |
| 2750 | 210 | 1618 | 145 | 150 |
| 1833 | 179 | 1163 | 62 | 75 |
| 1375 | 212 | 1138 | 35 | 56 |
| 1100 | 223 | 1773 | 42 | 73 |
| 916 | 212 | 1184 | 33 | 54 |
| 687 | 210 | 1211 | 39 | 64 |
| 550 | 204 | 1911 | 37 | 67 |

*Table 7. The result of the downscaling of the number of players per player simulation instance*

It was not possible to scale the number of players per instance linearly since the number of players were divided among the simulation instances. The measurements of the player simulations reveals that the simulation instance itself adds to the response times. To achieve an average response time below 100 ms, there cannot be more than 2000 players per simulation instance. And the response time becomes asymptotic at about 40 ms, with less than 1500-1400 players per instance, Figure 31.

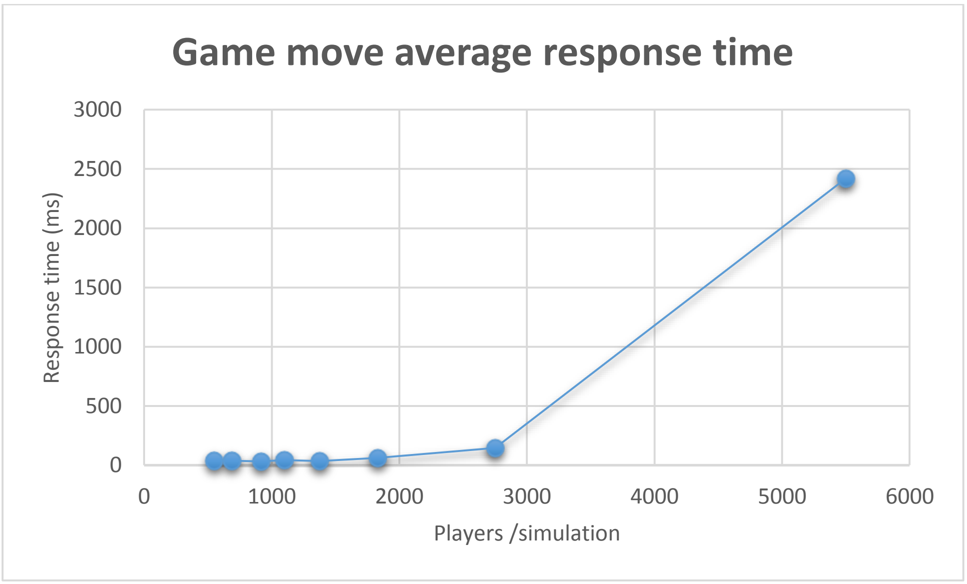
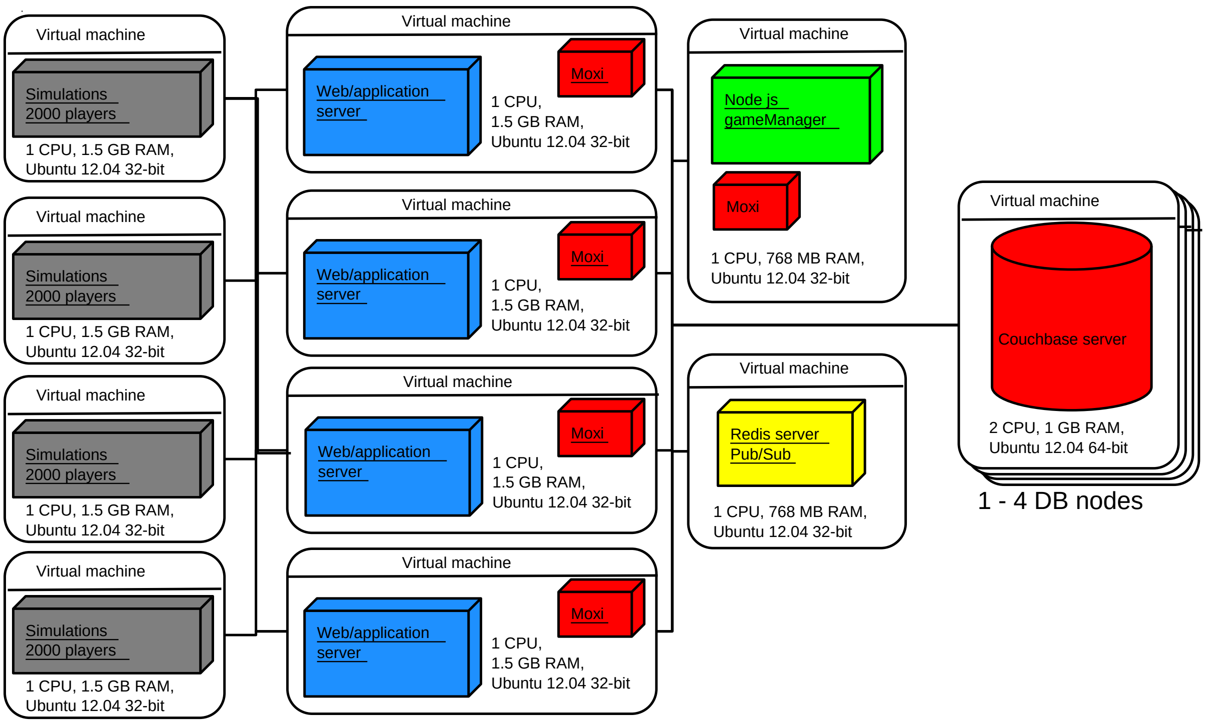


*Figure 31. The average response time when scaling down the number of players per player simulation instance*

### 7.2.10.5 Measuring the database scalability

To measure the database scalability (scaling the number of database nodes), the number of players was set to 8000 divided among 4 instances running 2000 players each. 4 application servers were connected to the database cluster with a variable number of nodes, varying between 1 and 4. The setup is illustrated by Figure 32.



Virtual machine
Simulations 2000 players
1 CPU, 1.5 GB RAM, Ubuntu 12.04 32-bit
Web/application server
Moxi
1 CPU, 1.5 GB RAM, Ubuntu 12.04 32-bit
Node js gameManager
1 CPU, 768 MB RAM, Ubuntu 12.04 32-bit
Redis server Pub/Sub
Couchbase server
2 CPU, 1 GB RAM, Ubuntu 12.04 64-bit
1 - 4 DB nodes

*Figure 32. The setup when scaling out the number of nodes in the database cluster*

To connect to the database cluster each Node.js process on the application servers were connected through a proxy, Moxi 1.8 [64], which allowed the server to keep using the memcached protocol as the proxy handled the key distribution among the nodes in the cluster. The bucket was configured without any replicas to achieve linear capacity when adding a node.

Using one node, the average response time is measured to 555 ms, with 8000 players, see Table 8. When adding a second node to the cluster, the average response time jumped to 3300 ms. That is, with 2 nodes, the average response time was about 6 times higher. The response time decreases when adding a third and a forth node, but is still higher than using only one node.

| # nodes | game moves / second | max response time | min response | average response time | Get games max | Get games min | Get games avg. |
|---|---|---|---|---|---|---|---|
| 1 | 296 | 14096 | 3 | 555 | 2246 | 40 | 428 |
| 2 | 222 | 35329 | 14 | 3040 | 2568 | 42 | 692 |
| 3 | 241 | 31993 | 18 | 2349 | 3601 | 201 | 691 |
| 4 | 212 | 25927 | 6 | 1258 | 4023 | 21 | 824 |

*Table 8. The result of the measurements when scaling out the number of nodes in the database cluster.*

Figure 33 shows how the average response time and game moves is affected by the number of nodes in the database cluster.

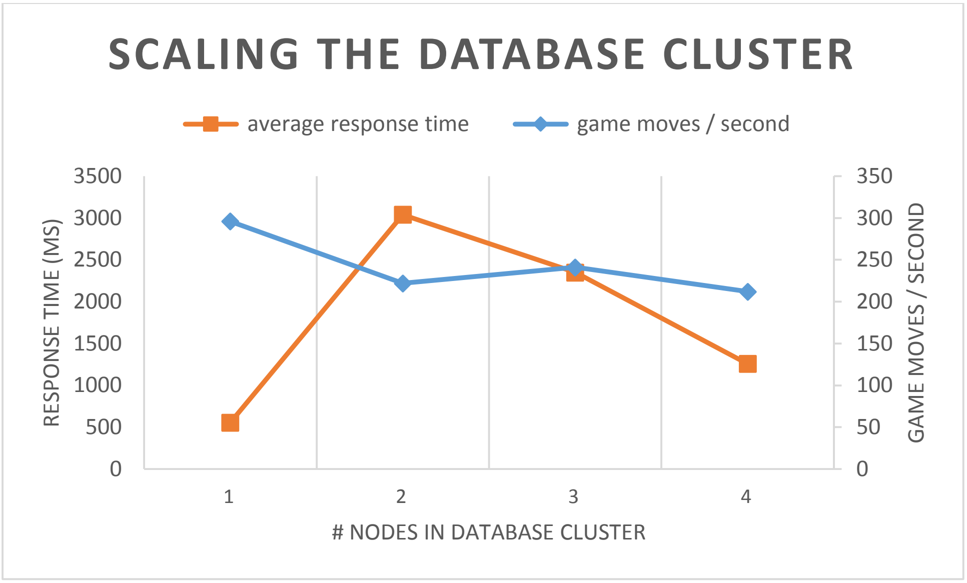


*Figure 33. The average response time when scaling out the number of nodes in the database cluster*

In Table 6 on page 64, the result of using only one web server instance was presented. An average response of 61 ms. was measuered with 2000 players.  With horizontal scalability, we would expect to see the same result when using four times the machines and four times the players (2000 players per machine). But instead, 1258 ms was measured, which is 20 time higher than expected. It was also twice the response time compared to using one database node at the same level of simulated players (8000).

For a comparison of what results to expect when scaling out the number of nodes, the Altoros presentation from CouchConf San Francisco 2012 [65] demonstrates an average write response time of below 2 ms at 20000 writes per second. A read response time below 5 ms at 20000 reads per second with data size of 1.5-2.5K. Roughly about the same data size as in the game states. Each game move renders  4.1 database operations (1 read serving the client, 1 read retrieving the game state when making the move and one update when writing the altered game state and one append for the log messages (1/10 of the time one extra write) = 2 reads, 1.1 writes and 1 append = 4.1). At 250-300 moves per second this is about 1000-1200 database operation per second. This is 1/20 of the cluster load compared to Altoros test in which the response times were averaged at below 5 ms. At a load of 1500 operations per second, their benchmark produced response times below 2 ms.

The difference in the configuration of the database nodes compared to Altoros, was 2 CPUs, 1 GB of memory and one disk compared to 4 CPUs, 15 GB of memory and 4 drives. In our case, the total size of the data never exceeded the 1 GB, which allowed the data to fit in the cache of the nodes. This would prevent the memory size to be a bottleneck.

Looking at the CPU consumption of the Moxi proxy, it was about 5-10 %, the CPU usage of the application server about 50% and memory consumption at a total of 70%. There were no indications that the problem was neither CPU bound nor memory bound. This indicates that the scalability limitation lies somewhere else. It could be in the application, a proxy overhead, a configuration failure of the Moxi proxy, a combination of the memcached client and the Moxi proxy or the underlying infrastructure the cluster was run on.

To test the database, without having to run the implementations and application servers, a simple node.js program was constructed, that only measured response time of set and get operations. These types of operations were the most common. The tests were run using two virtual machines with two node.js instances producing the database request operations get and set.
The trend is that the response time increases when going from one node to two and three nodes, see Figure 34. Then it decreases when going from three nodes to four nodes. This pattern was similar to the pattern that emerged when running the full scale simulations, scaling out the databases.

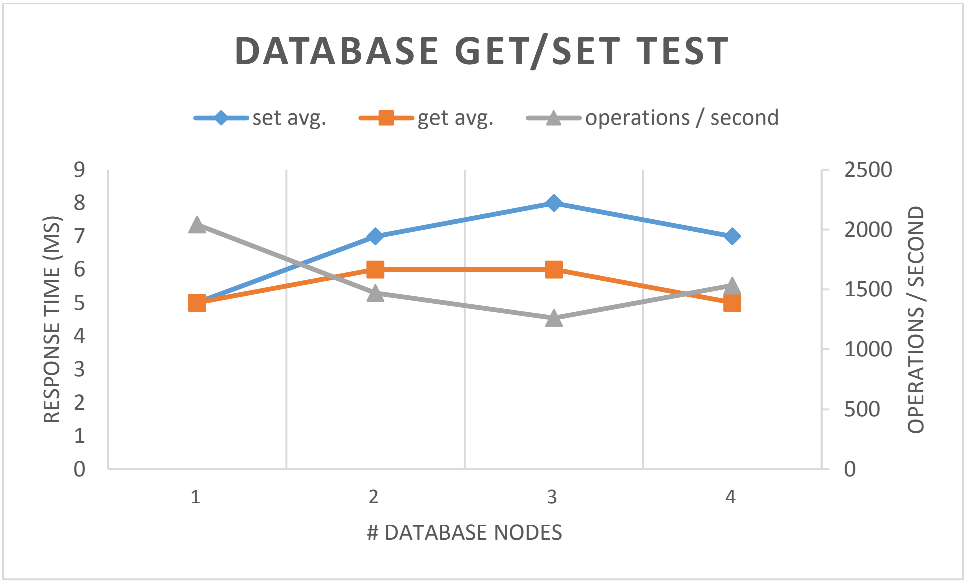


*Figure 34. The result of the database test measurements*

At this point the time spent for the project had already gone overdue and as a last attempt to address the problem, the database client was once again replaced. Couchbase recommends using a cluster aware client, instead of using the Moxi proxy. There was an experimental proxy aware client under development for node.js build on top of the c library libcouchbase. In a last attempt to see if the problems lies within the Moxi client, the database layer in the application server was refactored to make it easier to change the database client. Support for the Couchbase, cluster aware client Couchbase 0.10 was implemented in conjunction with the refactoring. Test were performed, but could not last long enough to produce any viable result since the Couchbase client stopped producing any responses after running for about 30 seconds. The queries arrived to the database and back, but no callbacks were triggered upon the responses by the client. It was possible to get it more stable and running for a longer time when the number of players were reduced to 1000. But this was not useful for performing measurements of scale. The cluster aware client was in an early stage of development and will hopefully be improved in the future.

At this stage there were no time left for investigating the cluster scalability problem. The next possible steps will be discussed in 9.4.

# 8 Result

Firstly the result of the architectural design is presented, then followed by the implementation details of the server, game manager, web browser client and the player simulations. The results of the measurements are presented at the end of the chapter.

## 8.1 Architecture

The architecture is designed to scale horizontally where capacity can be added by adding web server instances and database nodes running on separate logical machines. Figure 35 illustrates an installation using a hypervisor cloud. A hypervisor is a virtual machine manager that can create and run virtual machines. The architecture is the theoretical result of the literature study, but can also be used as an implementation. This architecture was used when performing the measurements, with an exception of the load balancer.

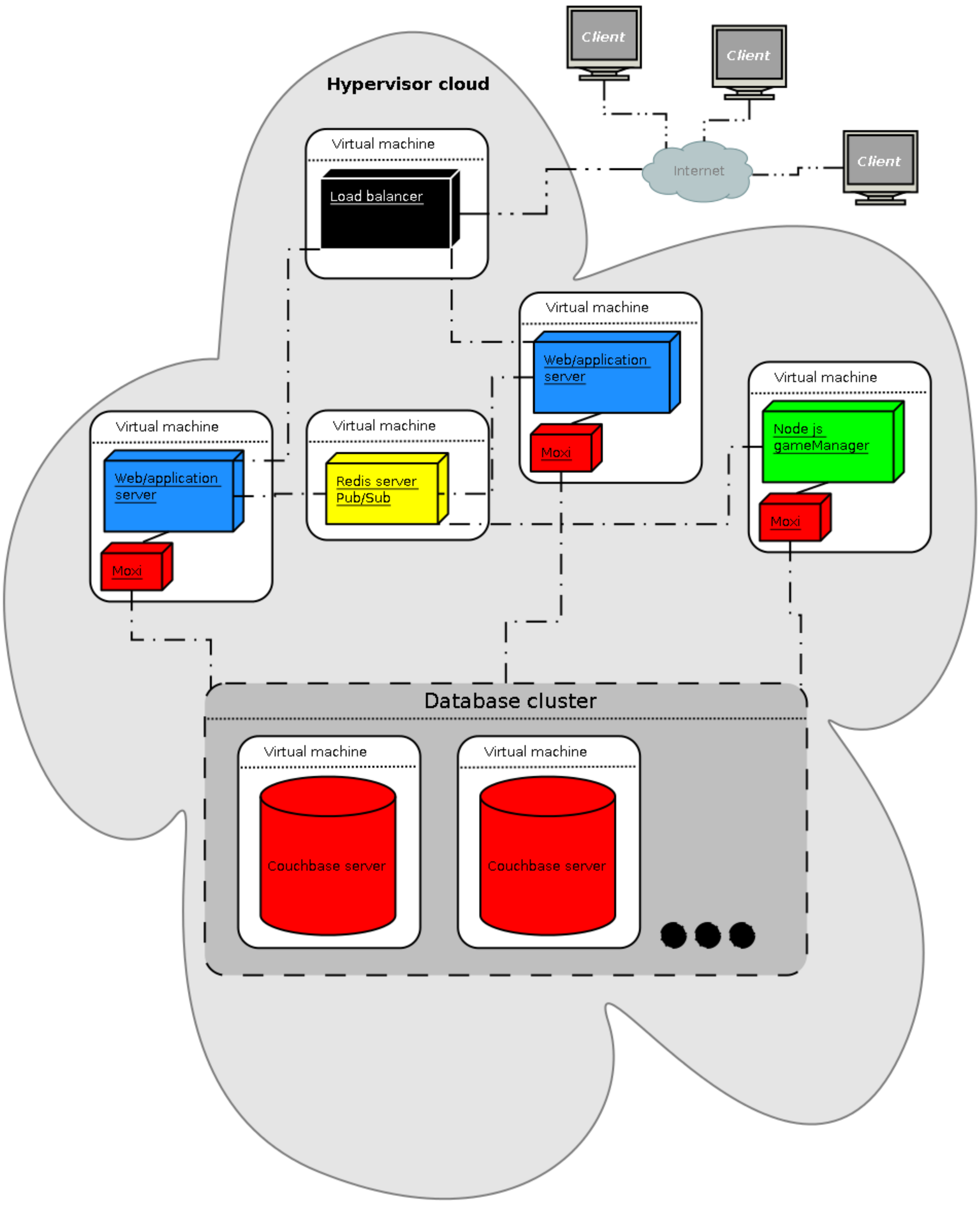


*Figure 35. The architecture of the complete system when deployed for maximum scalability.*

The hypervisor typically runs on separate hardware and abstracts the notion of where the virtual instances are physically located. The game clients are connected through http/web sockets via internet through a load balancer which direct the incoming connection to the web server instance that has lowest number of connected users. The load balancer could also use a simple round robin delegation. The use of web sockets is preferred when using a load balancer since it is a persistent connection to the server. Otherwise, if using http long polling, where there is a new connection at a predetermined interval, the load balancer could relay the connection to a new web server each poll. There are techniques to handle this issue such a sticky connections which inspect the package content to preserve a client - server connection.

The application servers are interconnected utilizing the publication subscribe feature in Redis server. This interconnection is used for sharing events among the application servers; such as game state change events or chat message sent events. On such events a message is sent through the publication channel and all of the subscribing servers receives the message. The message is matched to its connected users and relays the event to the player client.

The game manager (gameManager) takes care of creating new games and keeping track of inactive games by their timestamps. When a player requests a new game, a message is sent through the pub/sub system to the game manager which matches players, creates a new game and notifies the involved players.

Each web server instance communicates with the database cluster through a local proxy, which translates the memcached protocol to the cluster aware Couchbase protocol. The database cluster can be scaled by adding additional database nodes, machines running the Couchbase server application. When a new node has been added to the cluster, it can be rebalanced to utilize the new node and redistribute the keys evenly among the nodes. This can be done without taking the cluster offline. When the node joins the cluster it gets the same settings as the other nodes, therefore its specification should be similar or identical to the other nodes.

If compared to Figure 2, there are a few additions in the final architecture such as the pub sub interconnection between the web servers and the gameManager which allows for concurrency safe game management. The cache tier is integrated into the database cluster which reduces the complexity of accessing the data by only have to send the data requests to one source, removing the need of taking care of cache misses and cache updates.

For a smaller setup, all of the applications can run within a single machine. This would reduce the cost of additional machines for the game manager, load balancer, Redis server and database cluster. The load balancer can be omitted if only one web server instance is used.

## 8.2 Implementation

### 8.2.1 Server

The architecture of the server is represented in Figure 36.

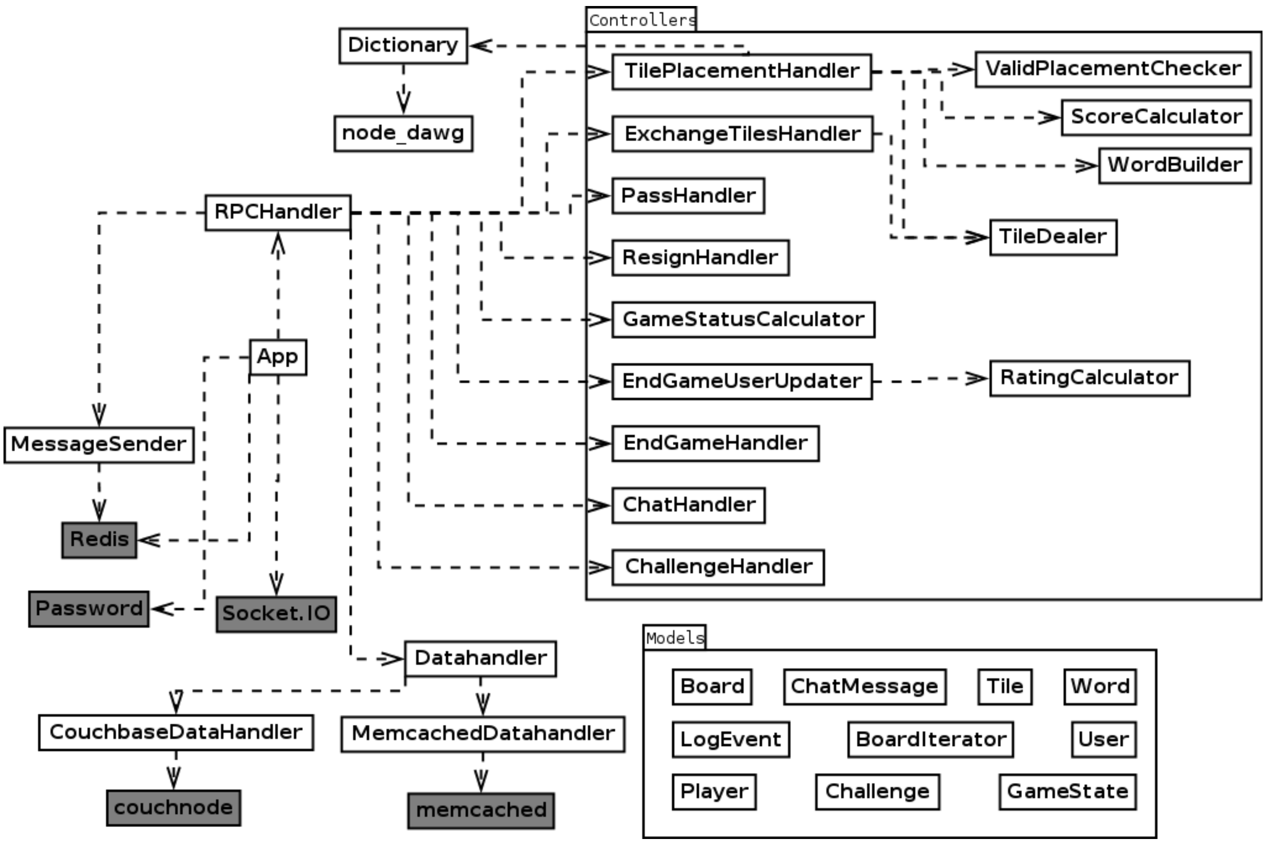


*Figure 36. Overview of the implementation of the web server.*

The server is started by executing the app.js module which initiates the web and all related modules. Before the server is started, the listening port, Redis server address and memcached address has to be configured.

The web server is using the express.js framework to serve the static and dynamic client files to the server and everyauth to take care of the authentication. Once the client is authenticated it uses the authentication cookie to reconnect through socket.io. Socket.io exposes the RPCHandler methods which are implemented by the handler modules in the controllers-directory. See Table 9 for the exposed RPC methods. The rest of the modules in the controller package handles all of the game logic, user handling and chat functionality.

| |
|---|
| addChallenge |
| addFriend |
| deleteFriend |
| exchangeTiles |
| getBoard |
| getChallenge |
| getChallenges |
| getChatLog |
| getFriendIds |

| |
|---|
| getGames |
| getGameState |
| getLatestChatMessages |
| getLogEvent |
| getLogEvents |
| getServerTime |
| getUser |
| getUsers |
| newRandomGame |
| pass |
| placeTiles |
| resign |
| respondToChallenge |
| saveBoard |
| searchForUser |
| sendChallenge |
| sendChatMessage |
| updateUserSettings |

*Table 9. The exposed RPC methods.*

The modules in the Models “package” are used for creating and mutating the models. These classes simplifies refactoring the implementation structure of the models since all creation and mutation goes through these modules.

The data handler abstract retrieving and updating the game states, users, chat messages etc. There are two underlying data access implementations, of which the memcached is the most mature, but eventually, the couchnode library might be the choice as it is cluster aware and backed by couchbase, eliminating the need of the moxi proxy.

The inter process communication is handled by the messageSender module. The app subscribes to a redis pub sub channel which the message sender sends the messages to.

### 8.2.2 GameManager

The architecture of the Game manager is described in Figure 37.

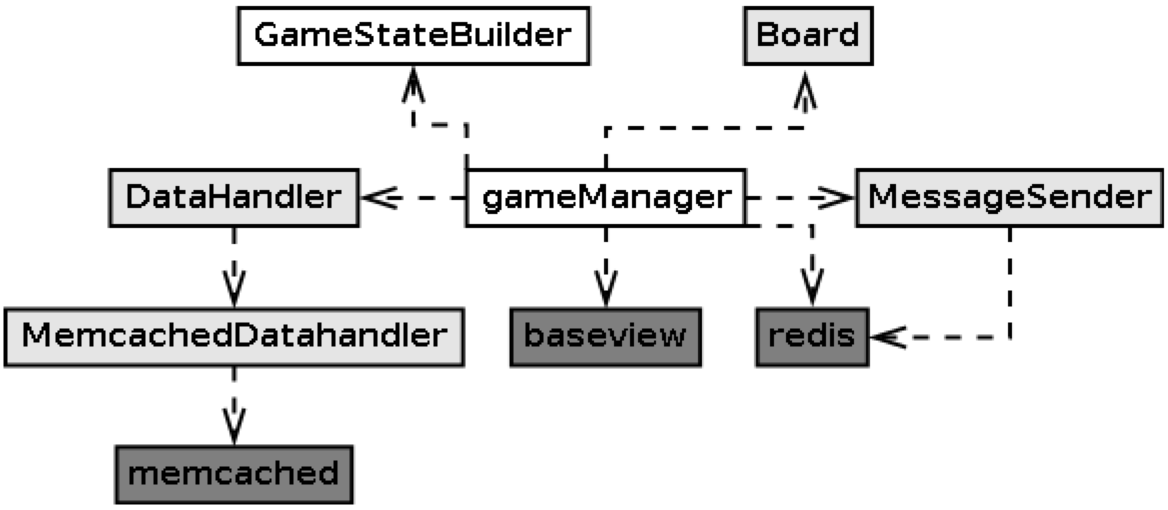


*Figure 37. An overview of the game manager implementation*

The game manager is needed due to the need of atomic operations for matching players, creating and managing game states. The modules DataHandler, Board and MessageSender are shared with the web server and the MessageSender is used to notify when a new game has been created or an inactive game state has been taken care of.

### 8.2.3 Client

The architecture, and files of the web browser implementation of the client are represented in Figure 38.

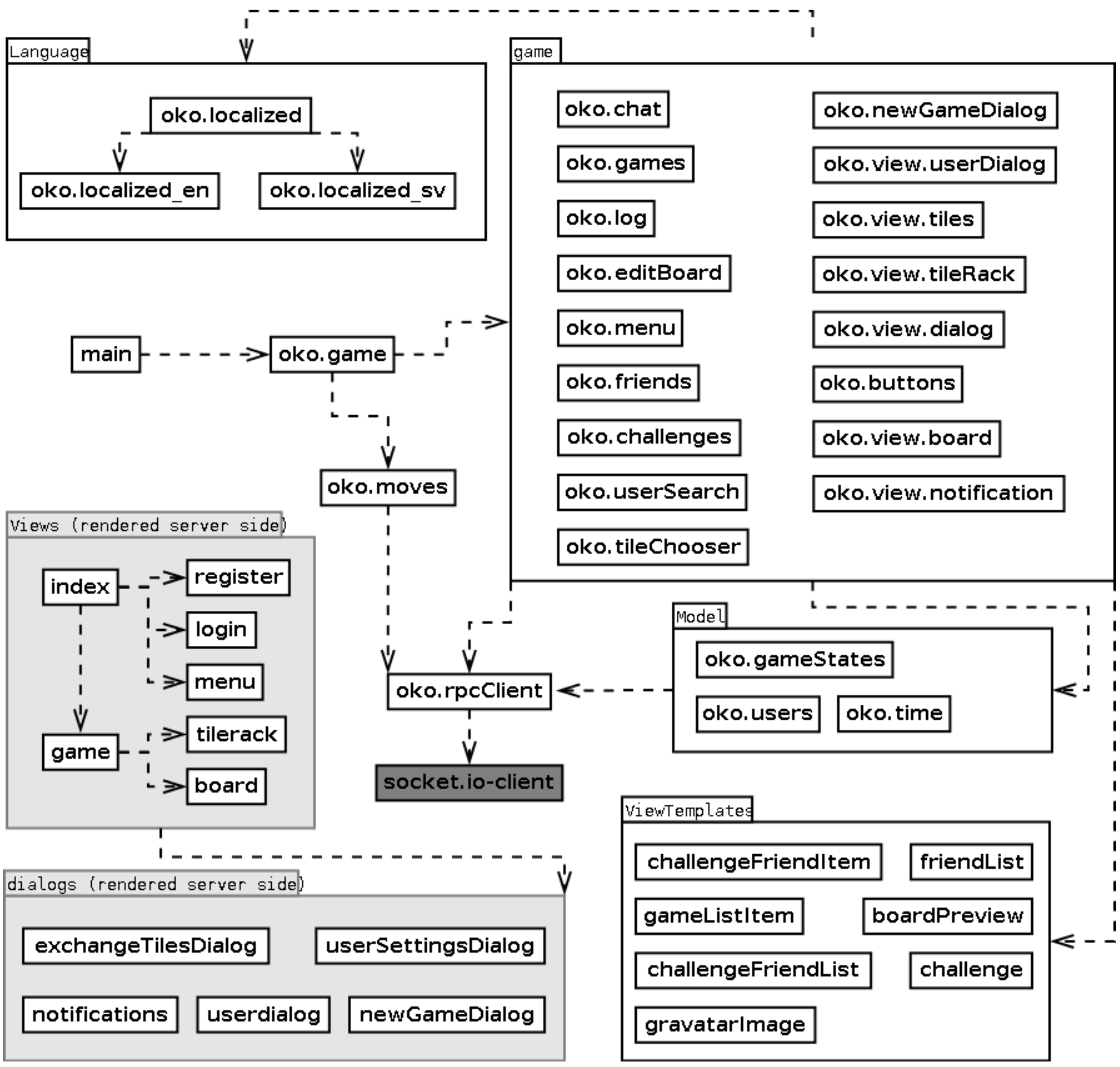


*Figure 38. An overview of the web client implementation.*

The client retrieves all static and dynamic content from the server, such as JavaScript, images and html. The main.js initiates the game and necessary libraries. Since there is only one scope, no module system like in Node.js, the game specific code is defined by the namespace “oko”. The client side of the game would be the view package if using the MVC pattern and all logic (controller) is handled by the server. When performing a move or loading a game state, the “rpcClient” is used send the specific RPC request to the server. The RPC client also registers a handler for incoming messages from the server, such as game state update events and when new chat messages are available to the player.

The files in the “model package” in the figure are used to cache the game models and “oko.time“ is used to synchronize the server time to be able to convert timestamps to local time.

Figure 39 shows the login screen of the game where it is possible to sign up or login.

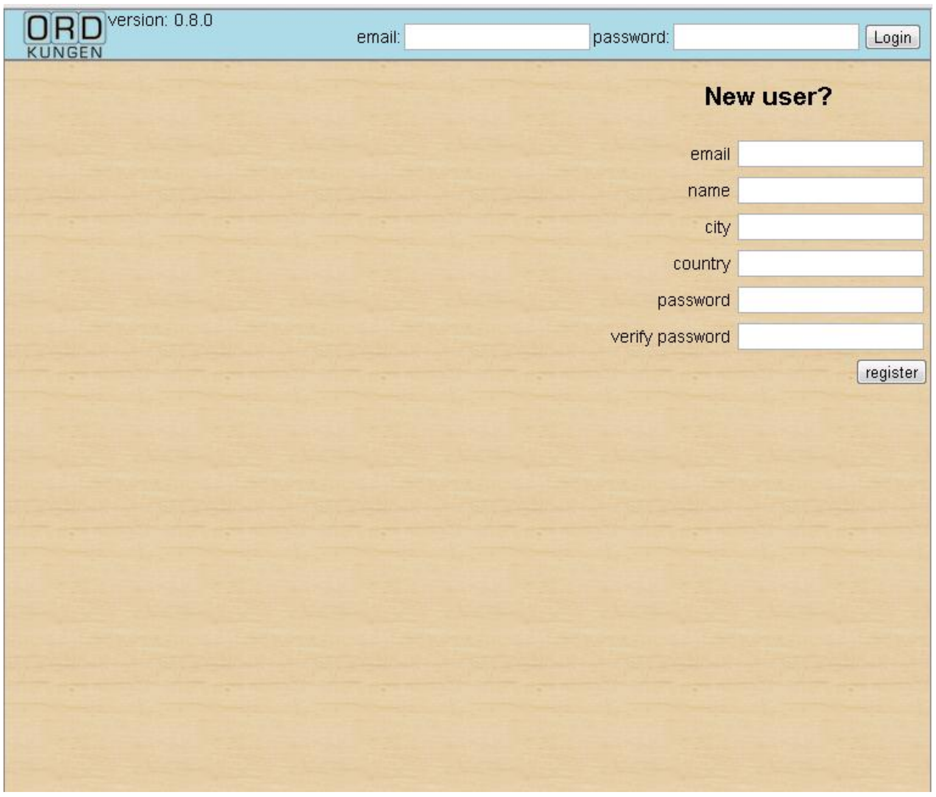
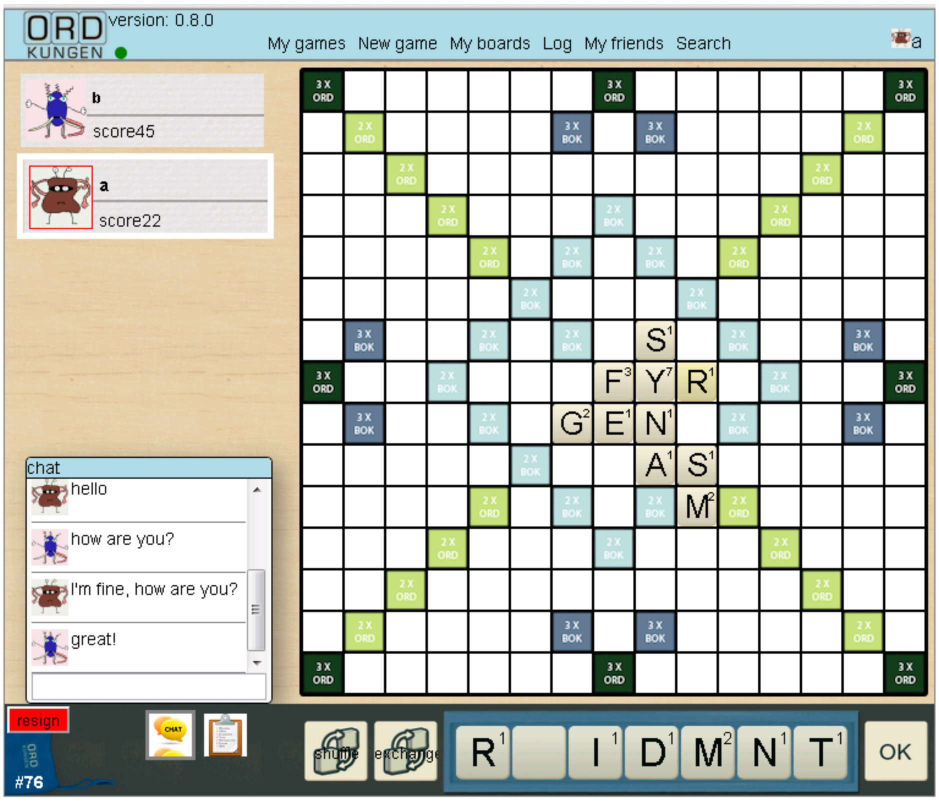


*Figure 39. The login view of the web client.*

Figure 40 shows the result of the game play. The menu on the top can be used to create new games, manage the user-designed boards, friends etc. On the bottom, there are buttons for opening the chat and the log. The buttons around the tile racket are used for tile operations; shuffle, exchange and move confirmation. The graphic is not yet completed, but the game functionality is there and the game is fully playable.



b
score45

a
score22

chat
hello
how are you?
I'm fine, how are you?
great!

Figure 40. Typical game play of the web client.

## 8.2.4 Player simulations

An architectural overview of the implementation of the player simulation and its files is shown in Figure 41.

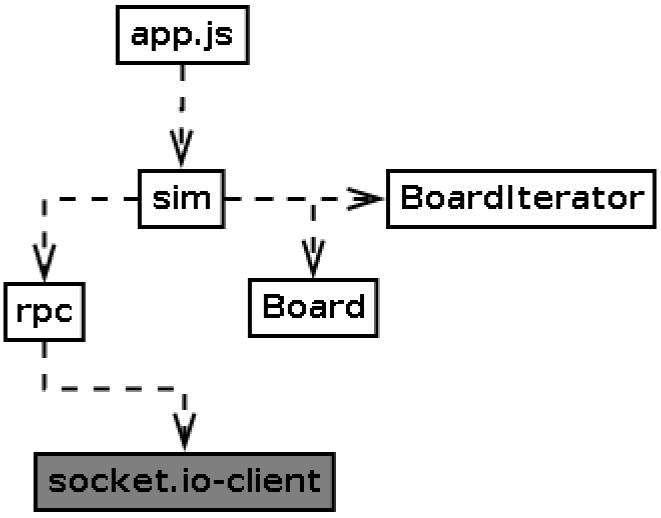


Figure 41. Overview of the player simulation implementation

The player simulations simulates clients logging in to the server and playing the game. The simulations are configured by entering a number of players the instance will simulate and the web servers the players will connect to. The number of players is represented as an interval of concurrent players the application will create, each player will get the number as a suffix to their name. The time between the moves is configured as minimum time plus a random time defined in ms. In the simulations these settings were 20 seconds plus 0 - 10 seconds. The random time has a uniformed distribution, giving an expected value of 20+5 = 25, varying between 20 and 30 seconds.

When starting the application, n instances of simulation will be created where n is the number of players in the configured interval (i.e. 0-1000). When a simulation instance is created, the player logs in (or registers if unsuccessful) and begins to play.

If it has less than 10 active games, a new random game request is sent. Then it tries to make a move. If it is not possible to make a move (it has no active), the player will sleep for 5 seconds and try again. The player also listens for events from the web server, and gets notified when an opponent finishes its ply.

The player makes the following moves:

- pass (p = 0.02)
- exchange tiles (p = 0.08)
- place tiles (p = 0.9)

The player simulator records all response times when performing RPCs to the server. The average values are calculated by a pre-defined interval.

## 8.3 Scalability tests

### 8.3.1 Application tier

Due to the unexpected behavior of the database cluster scaling it was not possible to perform any test measure the application tier limit. It would be necessary to eliminate the bottleneck created by the database tier to have the application throttled by the application tier. The test performed was at a maximum of 5500 players with a cluster of one database node, scaling out from one web server to five.

Figure 42 shows that the response time does not decrease linearly with the number of web servers. This is probably due to the limitations of the simulation instance and the database cluster. But it still shows that response time decreases and the number of moves per second increases by adding additional web servers. But going from 4 to 5 web servers instances does not have an impact on the performance, indicating that the throttling is not within the application tier at these numbers.

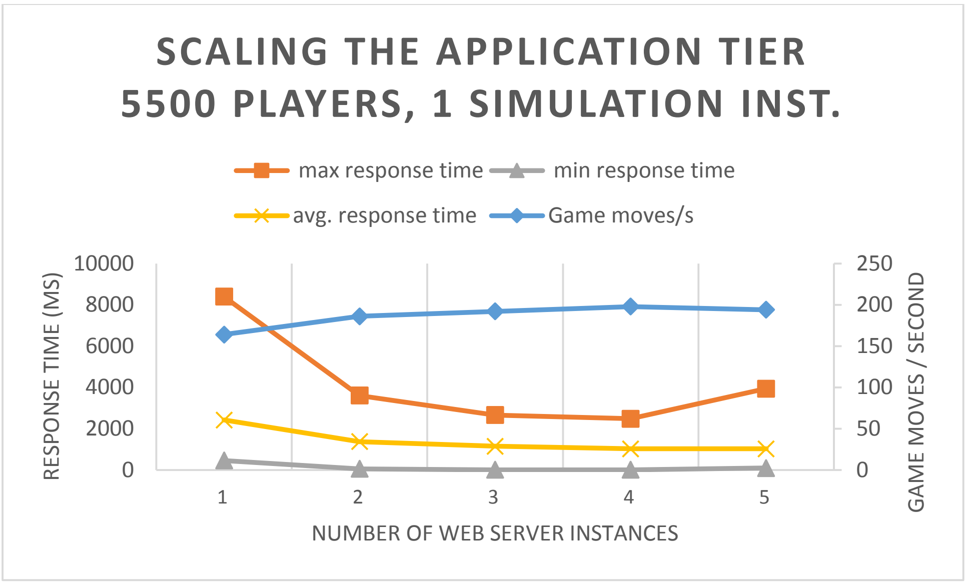


*Figure 42. The response time at 5500 players when scaling out the number of web server instances*

### 8.3.2 Simulations

Figure 43 indicates that when running too many players within one instance, the instance itself will throttle the measurements. To get an average response time beneath 100 ms, a simulation instance should not carry more than about 2000 players. But then even at 200 players, the simulation instance itself will inflict the measurements, but at an acceptable level. Ideally, there should be only one player per simulation instance, but for practical reason this is not feasible due to the overhead of setting up the test environment and the additional cost that would be added. It would be at least 2000 times more expensive to run one player per instance than 2000 player per instance.

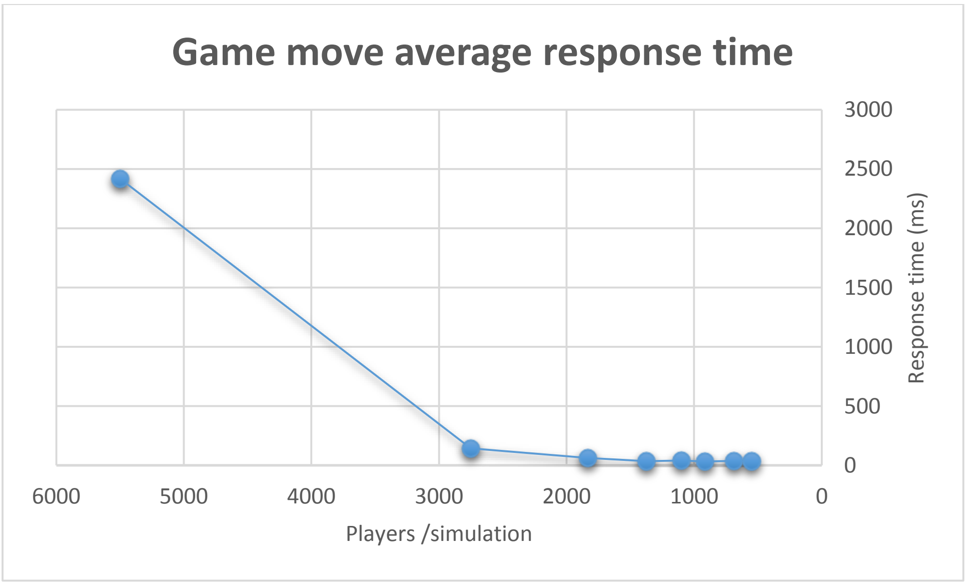


*Figure 43. How the number of players per player simulation instance affects the response time*

### 8.3.3 Database tier

When scaling out the number of nodes in the database cluster, the storage space increases linearly with the number of node. But, as the chart in Figure 44 shows, the response times increases despite that the data is equally partitioned among the nodes.

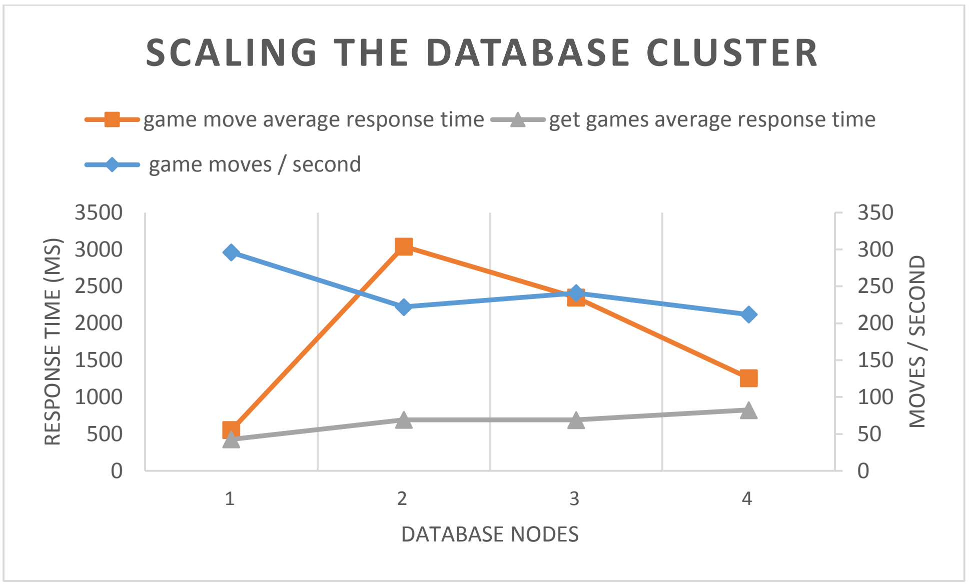


*Figure 44. How the response time is affected when scaling out the number of nodes in the database cluster*

# 9 Discussion

## 9.1 Result

### 9.1.1 Implementation

The architecture of the complete application follows the suggestion of [25], and the number of web server instances can be increased to handle a larger load. At the same time, there are some parts that will not scale very well, but has been regarded as scaling to an acceptable level. The game manager is one of these parts which is used for creating new games and taking care of expiring games. Although no measurements were made to state the number of game creations per second the system could manage, we can make a simple calculation of the proportion of creations versus moves. If there are 20 moves per game and one creation per 20 moves, assuming the same amount of resources is required for validating a move as creating a game, the problem would arise first when scaling out to 20 web server instances. This number of players was never achieved during the measurements, but when reaching that scale, the game manager could indeed become a bottleneck. The other part which was not addressed is the inter process communication. The current implementation sends a message to all web server instances upon each move that is made. As a first improvement to the number of messages each web server instance receives would be to partition the players among the web server instances by redirecting them to dedicated servers, something which could be done using the same hashing algorithm used by the data cluster. By this strategy, only messages targeting a particular partition of the players would be sent to the web server instance corresponding to that partition.

Based on the reported literature in the theory section, the architecture of the application could be regarded as scalable from a theoretical perspective, but there are no measurements to argue the claim because the issue with the database cluster scaling. The only measurements that gives an indication that the system is scalable is the measurement presented in 8.3.1, when going from one web server instance to two and reducing the response times.

To really verify the effect of scaling the number of web server instance, the capacity of the database cluster would have to be infinite, or at least not a throttling bottleneck. There are indications that some parts of the system are scalable, but not the composition of the system parts.

The most unexpected result was when scaling out the number of nodes in the database cluster. Even the separate database tests gave the same result with less performance when scaling out the number of database nodes. This could very well be caused by the usage of an immature and novel framework, combined with immature libraries. The only thing that could really be said about scaling out the number of nodes in the database is that the theoretical storage capacity increases linearly with the number of nodes connected to the cluster. Given the result of the Altoros presentation, with 20000+ requests per second compared to the maximum of about 2000 requests per second achieved within this thesis, the performance should at least not decrease when adding database nodes.

When addressing the choice of a novel framework such as the node.js platform, there were no major problems with the core framework. At first, the usage of JavaScript, an untyped language raised concerns about all runtime errors that a type checker would prevent in a typed language. These errors were discovered early in the development because of their severity and once fixed, they did not reoccur. The number of type fault did not reach its expectation. The positive aspects of using JavaScript is that the amount of code required for doing the same thing is less compared to i.e. java, as there are no classes. The event driven design suites JavaScript due to the existence of closures and ability to pass functions as variables. This is very useful for registering a function as a callback for a function call, the callback only need to be passed as an argument.

Another positive aspect of node.js was the momentum of the community embracing node.js with a lot of libraries covering almost any need. The downside was that there were often more than one alternative and it was hard to tell which one was the most mature. Otherwise, the quality of the libraries varied with some being well maintained and others being in effect discontinued.

If another framework would have been chosen, the implementation time would certainly have increased. But in turn, the time spent troubleshooting may have been decreased if a mature framework would have been chosen.

### 9.1.2 Measurements

The intention of the player simulations and measurements was to attest the grade of scalability of the application. The main unit of the measurements was response times because that what is of importance to the end user. The number of moves per second was a unit produced by the simulation instances, thus it was depended on the response times since the less time spent waiting for response, the more time spent making moves. The two measurements corresponds to one another. It was planned to use some of the other measurement method discussed in the literature chapter, but as for a start, the simple measurement of response time was adequate.

Measurements revealed problems with the game state sizes after using a profiling software. By reducing the size of the game states, the system was able to handle a greater number of players and also produced more predictable results. This problem had not have been discovered without running the simulations.

There was a noted variance of the measurements, two runs of the same simulation did not always produce the same result. A proper way of handling this uncertainty would be to measure the variance, state a confidence interval and do a proper number of repeated measurements to ensure the measurement is within a proper confidence level. In [34], they solve the problem of the varying performance of the infrastructure by introducing a performance variance parameter. This would have required automated tests because the amount of time required to perform repetitive tests.

If the scalability problem with the database cluster would be solved, simulations with more than 8000 players should be performed to measure the application tier scalability. If a system with two web server instances can serve twice as many players as one web server instance giving the same response times. The experiments should scale up to at least 4 or 8 web server instances to be able to state the scalability trend. This would have given a proper answer to if the application would fulfill the purpose.

It would also have been possible to do more measurements despite the database problem. It was clear that one database could accommodate 8000 players, but not clear if this was an upper bound. At the given time, the aim was to go for even larger test, with a setup of 50000 – 100000 simultaneous players. The database was not part of the implementation of the thesis, thus the focus lied on measuring scalability of the web servers. In retrospect, it would have been a better choice to complete the measurements with only one database to state the limit.

## 9.2 Methodology

There were a lot of time dedicated for reading previous literature discussing the scalability problem. Most of the papers define methods for defining and measuring scalability of an existing application and less about how to build a scalable application. The focus of the literature should have been averted towards the field of performance engineering and experience from the industry by reading white papers and real use case stories of successful implementation of scalable applications. These particular type of resources added the most value to the thesis as the entire application built upon those. In the field of performance engineering, the player simulations are used to continuously measuring the performance of the system. This turned out to be a good strategy, having measurements to lean against, ensuring improvements really are improvements. The use of these measurements could be further evolved would be to include them in the test suite. Just as using unit tests to test the application logic, the performance tests can be used to ensure the application performance. But once again, the tests have to be more reliable and predictable and produce the same result each time they are run. Otherwise, it is hard to tell if a modification is an improvement if the margins are small. The improvement could likewise be the cause of the variance of the underlying infrastructure or randomness of the tests.

The goal of implementing the complete game was a bit optimistic for the given time schedule. The time of the implementation was about 100% overdue, the time spent was not completely tracked and recorded, but it prolonged the time spent on the thesis and postponed the measurements. If the implementation had been more delimited, less time would have been spent on the implementation and more time would have been left for measuring and analyzing. By an earlier adoption of measurements before choosing and implementing a technology, scalability issues would be discovered earlier, reducing the complexity of solving them. The scalability issues regarding the database arose when the application architecture was regarded as finished and there were no time left for properly dealing with the issues.

Implementing the complete game was what added to the novelty of the thesis, compared to the literature in the theory chapter, namely building a complete web application with scalability in mind from scratch. The choice of using player simulations to mimic a real world scenario was also different compared to the other literature, which addressed more limited parts of systems, such as the web server or a search engine. The simulations also added to the complexity of controlling the experiments since there are random elements for producing the game moves. It made the simulations hard to test, debug, and predict. Even though the simulations were playing at random, the variable results of the measurements are most probably due to the use of a cloud platform with shared resources.

The use of an issue system was at help during the implementation, it provided a clear development path with scheduled milestones, and helped to keep focus on one task at a time. Without an issue system, it would have been hard keep track of what has been implemented when. This was at help when writing the realization chapter. The tasks planned could easily be translated into separated implementation sections. Combined with the SCM (source control management) it also added the ability to step back in time, to see the implementation at a given point in time during the implementation process as each commit in the SCM was linked to an issue in Jira.

The simulations had to be run manually and when changing the number of players in the player simulation instance, the simulation had to be restarted. The new settings were loaded and all the players had to login again. A lot of time was spent on waiting for the players to log in, especially when the simulation grew large. Each player worked independently and started playing as soon as it logged in. The more saturated the system became, the longer time it took to log in. The measurement could not start until all the players were logged in because the number of players was one of the parameters when performing the measurements. Given the time frame it was uncertain if the effort for building better tools would reduce the total time spent simulating or if the tools was regarded as good enough. In retrospect, better tools would have opened up for automated tests that could run a range of parameters during the night. Also when running more than one simulation instance, the results had to be aggregated manually. Given the number of hours spent on the manual tests, these hours would have been better spent automating the tests. But on the other hand, implementation is always a risk as it is hard to estimate the time needed. The safe path is to stick with a lesser tool and spend a lot of time using them but it also gives the lowest potential reward compared to developing a better tool. Despite the risk, I would recommend dedicating more time for building better tools to unburden the person using them, especially for application with repetitive use of the tools. This thesis was probably at the margin where better tools would pay off.

When setting up the larger tests with several machines it was possible to first install one machine and then take a backup of that machine, convert the backup to a machine template and make many instances of the template. This was also manual work and a test with 4 web server instances, 4 database instances and 4 simulation instances required setting up 3 templates and 9 additional machine using those templates. Setting up such an environment would take about 60 minutes given the ready templates. Each machine had to be additionally configured once started and the database node would have to be joined to the same cluster. The templates could be saved and reused for later tests. After the simulation sessions were completed the machines would have to be destroyed manually. There exists an API for automating machine instantiating using templates as well as automatically instantiating new machine based on CPU load thresholds. None of these automation possibilities were tested or used during the work of this thesis but would make it possible to automate the scaling out process as well. This ability should be explored to simplify the scaling process both for released deployment and simulation testing. An automated version for the released version and a version with ability to set different parameters through a control panel for testing and simulations. Such a tool could also be used for aggregating the status of the machines, i.e. CPU utilization, memory consumption and response latencies.

The database cluster incorporates an API for joining and parting nodes to the cluster as well as rebalancing. Making it possible to control the scale of the cluster through the application, instead of manual configuration.

The overall planning of the project did not fit into the time frame. The only part that did not exceed too much was the literature study. All other parts exceeded by at least 50 - 100% of the time. The scope of the project would be more suitable for two persons than one. Also, at about 100% of the planned time spent, I began work full time and continuing the thesis during my spare time. This slowed down the progress of the thesis and prolonged it by about 5 month, the time spent performing the simulations, stability improvements and scalability improvements of the system. The last 5 month could probably be condensed into 1-2 month of working full time, giving a total overdue of about 4-8 weeks (between 20% and 40%). This appoints to the difficulty of planning software development since the source of the overdue is associated with the implementation and its unpredicted practical difficulties.

This is most certainly why most of the scientific papers only addresses delimited and controlled scalability problems or keeps to a theoretical level instead of implementing large complex systems.

In the end, building a scalable application is more about the development process than the choice of a good architecture. The process has to allow changes to the architecture or implementation as bottlenecks are discovered. No system is the other alike and therefore different system will have different bottlenecks.

## 9.3 Generalization

The implementation process and methodology could be generalized to fit any related problem such as turned base games and stateless web applications. The implementation itself is built with isolated parts for handling the game logic and those parts can be easily replaced to fit other games. The limitations of the architecture is that a game state can only be modified by a single player at a time, there is no mechanism for handling concurrent game state modification implemented. This would be hard to implements in the current architecture to fit applications such as collaboration software or real time games where user concurrently make changes to the game state. In such cases, the game state would have to be moved to the web server instances and the users have to be routed to the particular web server instance which holds the game state. The architecture presupposes stateless web servers which all can access the same state from the database cluster.

The measurements could be generalized to fit any web application regardless the architectural implementation. The specific implementation of the player simulation can be modified in the same manner as the server application to fit other turned based games where the players logs in and makes move upon their turn.

The methodology of using simulations as a method of verifying scalability could be applied to any project where scalability is required.

## 9.4 Future work

The most important issue to address is the database cluster issue and this could be done by:

- Contact the vendor to see if they have a solution
- Wait for the cluster aware client and hope it solves to problem
- Evaluate other alternatives, such as other databases

The concluding measurements should test how the system behaves when scaling out the number of web server instances with no bottlenecks in the player simulation instances or in the database cluster.

Further suggestions to better utilize the simulations is to investigate acceptable thresholds in terms of response times. Otherwise, the thresholds are only arbitrary assumptions. To be able to use the simulations as a verification of unchanged performance upon new features, the simulations should be automated to be run by a continuous integration server.

For a varying number of users, the application should adapt automatically by the current workload through automatic scaling out / scaling in.

For generalization of the software, it could be refactored for easier swap of the game logic to serve other games.

For the continuation of the scalability investigation, the choice of framework should be compared to other architecturally different framework such as java EE or MVC.Net. It would also be of interest to compare the performance of different database, i.e. how SQL databases compared to key value stores when storing game states.

# 10 Conclusion

The purpose of the thesis was to investigate how to build a scalable distributed web application, a social game, using modern tools and frameworks within a case study.

The work has consisted of a literature study as an attempt to get a clear definition of scalability. A clear, general definition could not be derived, but has to be defined within its context. Further on during the literature study, techniques and architectures were discussed and evaluated. A four tier architecture, discussed in 4.3 (Figure 2) was chosen as a foundation for the architecture for the implementation. After a quick evaluation of the Erlang framework, a decision to move to an imperative language was made. The framework Node.js, with JavaScript as language was chosen for the implementation. The game was implemented by porting tests and game logic from the original Objective C implementation for the iPad version. The development proceeded in iterative process backed up by an issue system to keep track of the tasks and iterations. Besides implementing the server logic, a client was developed for the web browser to be able to fully test the game functionality. As the database tier and caching tier Couchbase was chosen as database due to its combination of storage, cache and horizontal scalability.  When all essential game functionality had been implemented, the work proceeded by implementing the player simulations, which were able to log in and play the game. The simulations were used to test the performance of the server application using different parameters such as the number of players and number of web server instances. A memory problem was discovered which resulted in unpredictable behavior with occasionally unreasonable long response times. The size of the game states were reduced and the server application became stable. Other simulations were performed to discover the thresholds in the different tiers: the web server instances, player simulation instances and database cluster. When examining the scalability of the database cluster, another unexpected problem was encountered which resulted in a negative scaling effect when adding additional database nodes. The source of the problem was not located, and the simulations could not proceed to desired levels. Thus, the simulations could not be used to properly test the scalability of the system.

The conceptual design attests to a scalable design, supported by the literature in the literature chapter apart from some issues discussed in 9.1.1. The implementation includes all necessary basic functionality, but its scalability could not be fully verified by any concluding simulation measurements. Although, the measurements shows that two web server instance produces better performance than one, but further scaling does not add to the performance. With more time given and further work, the database issue might have been solved and the remaining tests could be conducted. The method of using simulations to test the application scalability and performance could be applied to any software project where scalability is a requirement. The work of the thesis shows the importance of using a good method to achieve scalability, rather than choosing a scalable architecture when building a web application.

The time planned for the implementation was well underestimated and the total time of the thesis was overdue. And despite the extra time added, all problems were not solved. As discussed in 9.2, a simplified version might have been a better choice due to the time limit. But, such a simplified version would not correspond to a real implementation.

# 11 References

[1] Miniwatts Marketing Group, "Internet world stats," Miniwatts Marketing Group, 2012. [Online]. Available: http://internetworldstats.com/stats.htm. [Accessed 25 march 2012].

[2] Wikimedia Foundation, Inc., "Facebook - Wikipedia," Wikimedia Foundation, Inc., 2012. [Online]. Available: http://en.wikipedia.org/wiki/Facebook. [Accessed 25 march 2012].

[3] B. Johnson, Writer, *Talk at 2010 USENIX Annual Technical Conference Keynote: Lessons of scale at facebook.* [Performance]. Youtube, 2010.

[4] Joyent, Inc, "Node.js," Joyent, Inc, 2012. [Online]. Available: http://www.nodejs.org. [Accessed 25 March 2012].

[5] erlang.org, "Erlang the programming language," erlang.org, 2012. [Online]. Available: http://www.erlang.org. [Accessed 2 april 2012].

[6] VMware, "RabbitMQ - Messageing that just works," VMware, 2012. [Online]. Available: http://www.rabbitmq.com/. [Accessed 2 April 2012].

[7] iMatix Corporation, "The intellegent Transport Layer - ZeroMQ," iMatix Corporation, 2012. [Online]. Available: http://www.zeromq.org/. [Accessed 2 April 2012].

[8] Redis, "Redis," Citrusbyte, 2012. [Online]. Available: http://redis.io/. [Accessed April 2012].

[9] COUCHBASE, "Couchbase | Simple, Fast, Elastic NoSQL Database," COUCHBASE, 2012. [Online]. Available: http://www.couchbase.com. [Accessed 2 april 2012].

[10] Amazon Elastic Compute Cloud (Amazon EC2), "Amazon Elastic Compute Cloud (Amazon EC2)," Amazon Web Services, Inc, 2012. [Online]. Available: http://aws.amazon.com/ec2/. [Accessed 2 april 2012].

[11] Ipeer AB, "Elastic Cloud Server - Virtuell server i molnet med stor flexiblitet," Ipeer AB, 2012. [Online]. Available: http://www.ipeer.se/elastic-cloud-server.php. [Accessed 2 april 2012].

[12] Otocolobus AB, "Rex verbi for iPad," Otocolobus AB, 2012. [Online]. Available: http://www.rexverbi.com/. [Accessed 2 April 2012].

[13] T. Hoff, "High Scalability: How FarmVille Scales To Harvest 75 Million Players A Month," POSSIBILITY OUTPOST, 8 February 2012. [Online]. Available: http://highscalability.com/blog/2010/2/8/how-farmville-scales-to-harvest-75-million-players-a-month.html. [Accessed 15 February 2013].

[14] S. Patterson, "Words With Friends Loses 2 Million Players," iEntry Network, 10 May 2012. [Online]. Available: http://www.webpronews.com/words-with-friends-loses-2-million-users-2012-05. [Accessed 15 February 2013].

[15] V. Thakkar, Writer, *Scaling Words With Friends' sudden enormous success.* [Performance]. GDC Vault, 2012.

[16] T. Lossen, "Berlin Buzz Words, Redis to the resque (Vimeo)," Berlin Buzz words, 6

June 2011. [Online]. Available: http://vimeo.com/26863362. [Accessed 5 April 2012].

[17] T. Lossen, "Wooga - Redis to the Rescue! – Why wooga Replaced MySQL with Redis," Wooga, 19 April 2011. [Online]. Available: http://www.wooga.com/2011/04/redis-to-the-rescue-why-wooga-replaced-mysql-with-redis/. [Accessed 5 April 2012].

[18] Collins, "Collins English Dictionay - Complete & Unbridged," Collins, 2012. [Online]. Available: http://dictionary.reference.com/browse/scalability. [Accessed 8 May 2012].

[19] M. D. Hill, "What is Scalability?," *ACM SIGARCH Computer Architecture News, December,* pp. 18-21, 1990.

[20] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," *AFIPS spring joint computer conference*, vol. 30, pp. 483-485, 1967.

[21] E. Luke, "Defining and measuring scalability," *Proceedings of Scalable Parallel Libraries Conference,* pp. 183-186, 1994.

[22] A. B. Bondi, "Characteristics of scalability and their impact on performance," *Proceedings of the second international workshop on Software and performance - WOSP '00,* pp. 195-203, 2000.

[23] M. Michael, J. E. Moreira, D. Shiloach and R. W. Wisniewski, "Scale-up x Scale-out: A Case Study using Nutch/Lucene," *2007 IEEE International Parallel and Distributed Processing Symposium*, pp. 1-8, 2007.

[24] Gigaspaces, "Scale Up vs . Scale Out, whitepaper," Gigaspaces, 2011.

[25] B. Adler, *Building Scalable Applications In the Cloud Reference Architecture: Reference Architecture Best Practices, whitepaper,* RightScale Inc., 2011.

[26] R. Cattell, "Scalable SQL and NoSQL Data Stores," vol. 39, no. 4, 2010.

[27] J. Pokorny, "NoSQL Databases : a step to database scalability in Web environment," pp. 278-283.

[28] P. Jogalekar and M. Woodside, "Evaluating the Scalability of Distributed Systems," *IEEE Transactions on parallel and distributed systems*, vol. 11, no. 6, pp. 589-603, 2000.

[29] J. L. Gustafson, "Reevaluating amdahlal's law," vol. 31, no. 5, pp. 532-533, 1988.

[30] A. Y. Grama, A. Gupta and V. Kumar, "Isoefficiency: Measuring the Scalability of Parallel Algorithms and Architectures," *IEEE Parallel & Distributed Technology,* pp. 12-21, 1993.

[31] D. Rover, "Scalability of parallel algorithm-machine combinations," *IEEE Transactions on Parallel and Distributed Systems,* vol. 5, no. 6, pp. 599-613, #jun# 1994.

[32] X.-h. Sun, Y. Chen and M. Wu, "Scalability of Heterogeneous Computing," *2005 International Conference on Parallel Processing (ICPP'05),* pp. 557-564, 2005.

[33] Microsoft, "Windows Azure," Microsoft, 2012. [Online]. Available: http://www.windowsazure.com/. [Accessed 12 April 2012].

[34] W.-T. Tsai, Y. Huang and Q. Shao, "Testing the scalability of SaaS applications," *2011 IEEE International Conference on Service-Oriented Computing and Applications*

*(SOCA),* pp. 1-4, #dec# 2011.

[35] M. L. Abbott and M. T. Fisher, The Art of Scalability, First edit ed., M. Taub, T. MacDonald, S. Qiu, J. Fuller, A. Popick and K. Brooks, Eds., Boston: Addison-Wesley Educational Publishers Inc, 2010.

[36] C. U. Smith, D. Ph, L. G. Williams and R. Lane, "Best Practices for Software Performance Engineering," 2003.

[37] L. Duboc, E. Letier, D. S. Rosenblum and T. Wicks, "A Case Study in Eliciting Scalability Requirements," *2008 16th IEEE International Requirements Engineering Conference,* pp. 247-252, #sep# 2008.

[38] B. Veal and A. Foong, "Performance scalability of a multi-core web server," *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems - ANCS '07,* p. 57, 2007.

[39] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *ACM SIGACT News,* vol. 33, no. 2, p. 51, #jun# 2002.

[40] J. Han, E. Haihong, G. Le and J. Du, "Survey on nosql database," *Pervasive Computing and ldots,* pp. 363-366, 2011.

[41] Mochi Media, Inc., "mochi/mochiweb," 2012. [Online]. Available: https://github.com/mochi/mochiweb. [Accessed 8 May 2012].

[42] yaws, "Yaws Yet another web server," 2012. [Online]. Available: http://yaws.hyber.org/. [Accessed 7 May 2012].

[43] D. Pariag, T. Brecht, A. Harji, P. Buhr, A. Shukla and D. R. Cheriton, "Comparing the performance of web server architectures," *ACM SIGOPS Operating Systems Review,* vol. 41, no. 3, p. 231, #jun# 2007.

[44] Joyent, Inc, "Node Packaged Modules," 2012. [Online]. Available: https://npmjs.org/. [Accessed 7 June 2012].

[45] JetBrains, "The best JavaScript IDE with HTML Editor for Web development :: JetBrains WebStorm," 2012. [Online]. Available: http://www.jetbrains.com/webstorm/. [Accessed 5 June 2012].

[46] Atlassian, "Free source code hosting for Git and Mercurial by Bitbucket," 2012. [Online]. Available: https://bitbucket.org/. [Accessed 2 may 2012].

[47] fmontanari, "fmontanari / nunitjs," 2012. [Online]. Available: https://github.com/fmontanari/nunitjs. [Accessed 15 june 2012].

[48] G. Rauch, "Socket.io," Learnboost, 2012. [Online]. Available: http://socket.io. [Accessed June 2012].

[49] M. Morley, "JSON-RPC," MPCM Technologies LLC, 2012. [Online]. Available: http://www.jsonrpc.org. [Accessed April 2012].

[50] expressjs, "Express - node.js web application framwork," [Online]. Available: http://expressjs.com/. [Accessed 31 March 2013].

[51] B. Noguchi, "everyauth," [Online]. Available: https://github.com/bnoguchi/everyauth. [Accessed 31 march 2013].

[52] The jQuery Foundation, "jQuery," 2012. [Online]. Available: http://jquery.com/. [Accessed 15 June 2012].

[53] The jQuery Foundation., "jQuery UI," 2012. [Online]. Available: http://jqueryui.com/. [Accessed 15 June 2012].

[54] DocumentCloud, "Underscore.js," 2012. [Online]. Available: http://underscorejs.org/. [Accessed 15 June 2012].

[55] W. A. Andrew and G. J. Jacobson, "The world's fastest scrabble program," *Communications of the ACM*, vol. 31, no. 5, pp. 572 - 578, 1988.

[56] Atlassian, "Issue & Project Tracking Software | Atlassian JIRA," Atlassian, 2012. [Online]. Available: http://atlassian.com/software/jira/overview. [Accessed 12 July 2012].

[57] D. Wood, "davidwood / node-password-hash," 2012. [Online]. Available: https://github.com/davidwood/node-password-hash/. [Accessed 15 July 2012].

[58] Gravatar, "Gravatar - Globally Recognized Avatars," 2012. [Online]. Available: http://sv.gravatar.com/. [Accessed 3 july 2012].

[59] overclocked, "mc - The Memcache Client for Node.js," 2012. [Online]. Available: http://overclocked.com/mc/. [Accessed 3 July 2012].

[60] Betapet, "Betapet - Rating," Betapet, 2012. [Online]. Available: http://www.betapet.se/rating/. [Accessed 17 july 2012].

[61] A. Sellier, "Less - The dynamic stylesheet language," Alexis Sellier, [Online]. Available: http://lesscss.org/. [Accessed 31 March 2013].

[62] T. A. S. Foundation, "Technical Overview: Couchdb," 2012. [Online]. Available: http://wiki.apache.org/couchdb/Technical Overview. [Accessed November 2012].

[63] Inc, Couchbase, "View basics: couchbase 2.0 manual," 2012. [Online]. Available: http://www.couchbase.com/docs/couchbase-manual-2.0/couchbase-views-basics.html. [Accessed November 2012].

[64] GitHub Inc, "GitHub," 2012. [Online]. Available: https://github.com/. [Accessed 6 june 2012].

[65] Nodetime, "Nodetime - Performance Profiler and Monitor for Node.js Applications, Node.js APM," Nodetime, 2012. [Online]. Available: http://nodetime.com/. [Accessed 5 November 2012].

[66] T. Eggert, "elbart/node-memcache," 2012. [Online]. Available: https://github.com/elbart/node-memcache. [Accessed 2 November 2012].

[67] Couchbase, Inc, "Moxi proxy 1.8," Couchbase, Inc, 2012. [Online]. Available: http://www.couchbase.com/docs/moxi-manual-1.8/. [Accessed 23 November 2012].

[68] Renat Khasanshyn, "Benchmarking Couchbase," Altoros Systems, 2012. [Online]. Available: http://www.couchbase.com/presentations/benchmarking-couchbase. [Accessed 30 December 2012].

[69] X.-h. Sun and J. Zhu, "Performance Considerations of Shared Virtual Memory Machines," *IEEE Transactions on parallel and distributed systems,* vol. 6, no. 11, pp.

1185-1194, 1995.