PROJECT CS - 1DT054
Uppsala University

"Treacherous Talks"

PRODUCT REPORT

Sunday 19th February, 2012

Dilshod Aliev
Jan Daniel Bothma
Stephan Brandauer
Andre Hilsendeger
Rahim Kadkhodamohammadi
Xinze Lin
Tiina Loukusa
Erik Timan
Sukumar Yethadka

Contents

Abstract

Treacherous Talks is an implementation of a board game ("Diplomacy") as a web service. Technical and functional requirements of the project were defined by our customer — Erlang Solutions [1].

The report explains the project and takes a look at the requirements. The technical details of the solutions chosen, are presented and their choices motivated. The features that make the project interesting, failure tolerance, scalability and multiple interfaces are highlighted. To conclude, some of our relative weaknesses are mentioned.

Chapter 1

Introduction

Diplomacy [2] is a board game, invented in the 1950s where the goal is to try to conquer Europe just before WW I. You come close to this goal by talking to the other players — by diplomacy — and making them your allies. And you achieve it by attacking them when they do not expect it.

The game is and was commonly played over distance — starting with playing by mail, then email and nowadays over pretty web pages with full-blown map visualization.

The requirements we were faced with asked for an implementation of Diplomacy as a web service while providing several interfaces to this service. Scalability and Failure Tolerance were of high priority.

Even though a board game is a fun thing to implement, we do think that the most interesting part of our project is the scalability- and fault tolerance-engineering.

For more details about the project methodology and how we tackled problems during the project, please refer to our course report.

Chapter 2

System Description

2.1 Requirements

We received a number of requirements from our customer Erlang Solutions that we were to include in the project.

2.1.1 Diplomacy server

The requirements on the backend were initially grouped as follows:

- Authentication — Only registered users should have access.
- Game master — To moderate games.
- Rule engine — The rule engine should evaluate orders and determine the results.
- Database — For storing game and user data.
- AI — Artificial intelligence based player for testing and playing games when real players are unavailable.

The authentication, rule engine and database features were directly implemented. The game master role morphed into two roles — the operator and the moderator. The AI feature is detailed further below in this section.

2.1.2 Three Interfaces

The requirement stated that we have to provide three interfaces that expose the same functionality to the user (the operator is forced to use the web interface). Those interfaces are:

1. **Web interface** — The game should be accessible via a modern web browser *and* it should use WebSocket to communicate with the browser.

2. **XMPP interface** — The game should be accessible via a chat client using text based orders.

3. **SMTP interface** — The game should be accessible via sending of emails using text based orders.

2.1.3 Scalability

The system should be scalable. Scalability in this context refers to horizontal scaling where the system should be able to handle more load by simply adding more nodes. No further explanations were provided regarding this feature, but since we saw it as a challenge we invested a lot of work into it.

2.1.4 Fail-Safety

The system should be highly available. We interpreted this by assuming that we should handle hardware failure of physical machines. As with scalability, we might have been able to get away with less work but this feature as well was too interesting to resist focusing on it.

2.1.5 WebSocket

WebSocket [3] is a web technology that enables two-way communication between a web browser and a web server over a TCP socket.

Its novelty is that it is not implemented by polling which makes it quite fast for updates pushed from the server to the client. WebSocket can be used by JavaScript to update only parts of the client page instead of reloading a page and is a great tool to shift computing to the client side.

The lack of tool support for WebSocket was an issue during the project, as detailed in Sec. 2.2.2.

2.1.6 AI

Writing a simple AI was a requirement which interested us as well. However, we were stuck in a tradeoff because time was limited: fail-safety+scalability vs. AI players. In combination with our customer, we agreed on focusing on the performance characteristics and leaving the AI players for future work.

2.2 Architecture

2.2.1 Overview

Our Architecture is divided in three main blocks, the *frontends*, the *controller*, and the *backends*. In Fig. 2.1 a running cluster as it could be configured is shown. *Server 1* shows a full configuration as it includes all three frontends,

a backend and Riak. Riak, our database is explained in Sec. 2.2.4. The operator, however, is free to choose to only run a specific selection of these and can combine them freely, for example as the *Server 2* in Fig. 2.1.

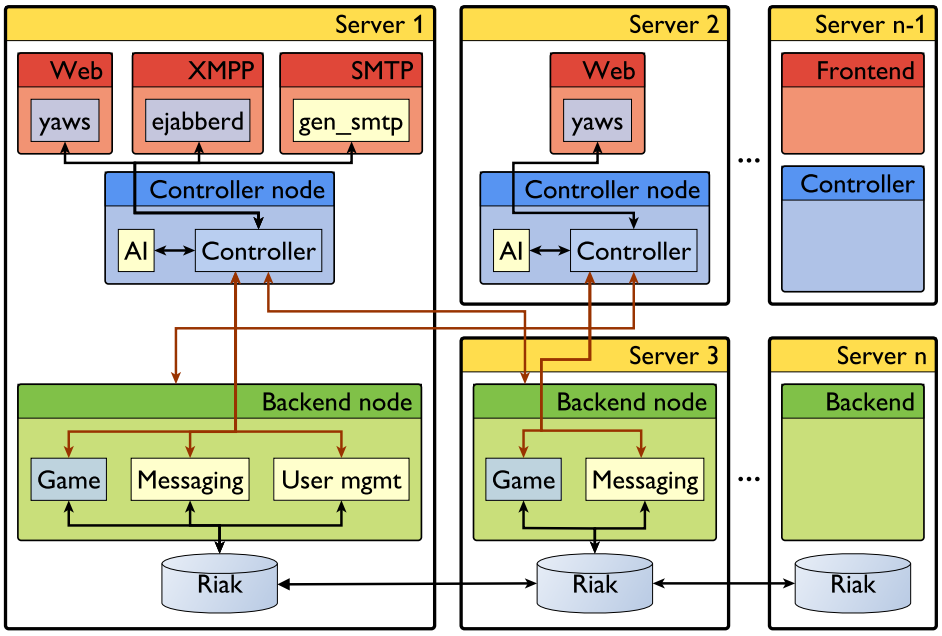

Figure 2.1: *A possible cluster.*

The responsibilities are not surprising: the frontends receive user input and transform the input into messages which are sent to the controller. The controller filters out illegal messages according to the session database and the user privileges before handing them off to the backends — which will respond with an answer.

2.2.1.1 Communication between applications

We follow the MVC [4] (Model View Controller) Pattern. The View contains all the supported User Interfaces (Web, SMTP, XMPP). The Controller is an application that talks to the Model. In the Model there are backend services (Game, Messaging and User Management) and the database. Some of the backend services talk to each other, while others work completely independent. Each box in Fig. 2.2 is an OTP [5] application, and each of them can be distributed onto multiple nodes and work simultaneously. The system is database-driven, that is, the backend services are stateless (see 2.2.1.4), and any request can be handled by any node where that service application is running. The communication between the applications is shown in Fig. 2.2.

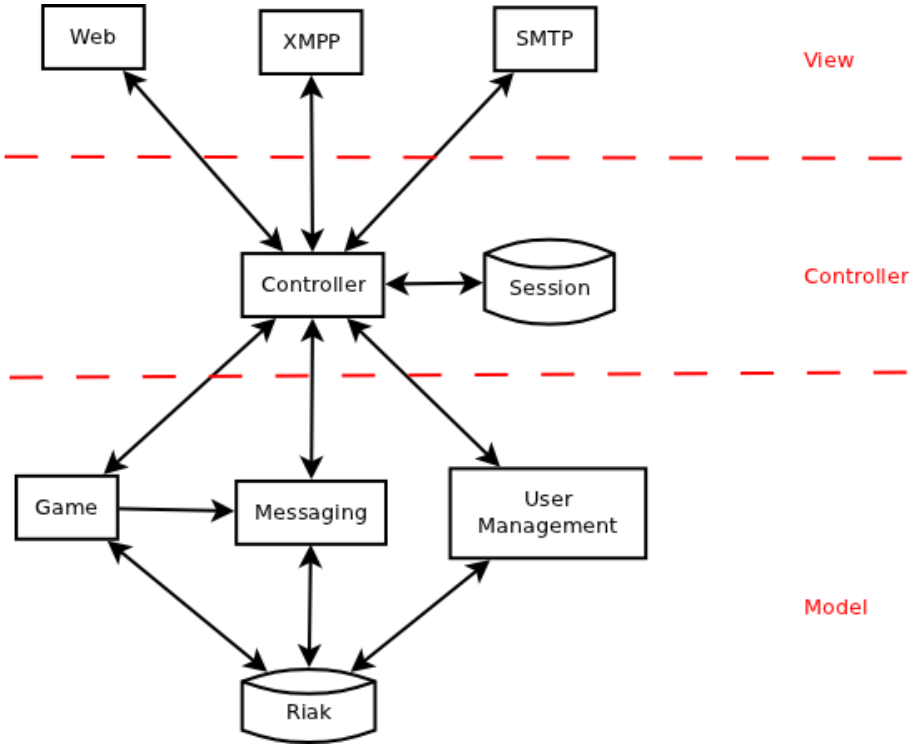

Figure 2.2: *Communication between applications.*

Arrows in Fig. 2.2 represent messages, where one application sends a message to another one. Instead of direct message-passing between applications, an application calls a function in the API module of the target application, which sends a message to a worker process in the target application. The worker process is chosen from the process group for workers (see Fig. 2.9) in that service application. In Fig. 2.2 you can see two databases. The one in the bottom, Riak (see 2.2.4), is the main database. The other one, Session (see Mnesia 2.2.4.2), is part of the controller and stores sessions.

2.2.1.2 Modularity — and its Relation to Scalability

One thing that is not immediately visible in our architecture (see Fig. 2.1) is the fact that the individual applications are very independent of each other. Even though the architecture graphic shows a complete example of a running cluster, we can choose to run some applications alone.

If we would, for example, find out that our system is slow because the web frontend is the bottleneck, we could add more servers to the cluster that run only a web frontend (see Fig. 2.1, "Server n-1"). We can do the same with each of the frontend types, with backends and with Riak nodes. Our architecture, therefore, inherently supports scaling out very well.

The controller application can be on any physical machine and any node. At least one instance of the controller application must be running. Extra instances will set up local copies of the session database, and session processes are distributed among the instances. We decided to run it on each backend node, since sessions would be created on the node where the controller worker process handling the login attempt is running. That distributes work evenly between backend nodes, except if some local application cannot deal with the request, and this was found to work as expected during load tests of various cluster sizes (see Sec. 4.3.1.1).

Since the frontends are independent from each other, we did not expect issues scaling with additional frontends. We therefore focused on testing the performance of our backends and Riak nodes, as shown in Sec. 4.3.

2.2.1.3 Process groups

The system uses process groups [6] to provide load balancing by distributing work across nodes. This is done by each process in the backend joining the process group for its application. A calling process can then use the process groups to find a process to call. The selection is done semi-randomly. Since process groups are visible globally, if there is no local process to call, a random process on another node is used out of that group. Work is then distributed across other running nodes, providing redundancy. Since selection is done randomly, it does not guarantee equal distribution between local or remote processes, but load testing showed this to work sufficiently (see 4.3).

2.2.1.4 Stateless design

One of the ways to scale a system is to use a shared nothing (SN) architecture [7] where every node is self-sufficient and the system has no single bottleneck. This allows for a distributed system that is both scalable and fault tolerant. The implementation of a SN architecture adds a constraint that individual nodes can only have state that could be lost or must be stateless.

With this in mind, our architecture was designed to allow applications to be distributed across the physical machines in various configurations (see Fig. 2.1). All the system state is moved to the database and nodes that have state can be restored from the database, in the event of failure.

We did have to use state in certain parts of the system due to the distributed nature of the application. Briefly, the parts of the system with state and the reasoning behind them are:

- Game timer — A game has various events associated with it and tracking these events with time needs a process with a certain amount of state.

- Game join process — Requests from multiple users joining the same game needs to be serialized to avoid concurrency issues.
- User session — Multiple requests from the same user needs to be ordered since without the ordering the requests can arrive at the backend at different times creating inconsistent data.

2.2.2 Frontends

While the three interfaces provide very different forms of interaction, namely via instant messaging, email and a web browser, we tried to make them as homogenous as possible in terms of the data transferred between the client and the server. This allowed us to make the implementation of front-end logic very generic. The frontends use the same API to the backend via the Controller, and responses and messasges pushed to the View do not vary between frontends (See Sec. 2.2.1.1). Each frontend transforms data between their respective format and that used to interact with the backend.

2.2.2.1 Web

The web frontend is running Yaws [8] as a web server. We initially decided to use Nitrogen [9] as a web framework, but had to drop Nitrogen because of the requirement of using WebSocket for interaction with the server. Yaws is configured as a "/" application module (appmod) so that we can have a single entry point for handling all the requests. Server side includes (ssi) are also used to load static content via AJAX [10].

The web frontend relies heavily on JavaScript [11] since it contains a large amount of the logic handling user interactions, requests and responses to the server and gameplay support for the user. We use the JavaScript library jQuery [12] to update the DOM elements and make AJAX requests.

We use the toolkit Bootstrap [13] for the user interface which allows us to quickly develop web pages. Bootstrap includes basic CSS and HTML for typography, forms, buttons, tables, grids, navigation, and more. Using Bootstrap reduced the time needed for web development, allowing us to focus on the backend.

We had the option of extending Nitrogen to support WebSocket, but after investigating how much work would be involved, we came to the conclusion that it would not be feasible given our time constraints. Nitrogen uses SimpleBridge [14] which provides a standardized interface to all its supported web servers. We found that adding WebSocket support would mean adding and changing a lot of code in a lot of places, because of the way Nitrogen/SimpleBridge is structured. Instead we implemented the web frontend ourselves with HTML and JavaScript directly served by Yaws.

Throughout the course of the project the WebSocket protocol has been under development. Only towards the end of the project has a protocol (RFC

6455) been released that has been proposed to be the official standard. Since the protocol might have changed significantly during the project, we decided to commit to a particular WebSocket draft version, and a web browser and server version that support it. This gave us a stable base to develop on.

We chose the "hybi-10" draft of the protocol [15] which was supported by Chromium 14 and a fork of Yaws 1.91 [16]. This draft version was chosen because it appeared likely to become the basis of the final standard [17] while the earlier drafts were incompatible.

When RFC 6455 was released, only minor changes were needed to make the branch of Yaws we were using compatible. We now support any browser that implements the "hybi-10" and newer drafts, or RFC 6455.

Yaws has since merged a fork with support for RFC 6455 but we have not updated our code to use this version of Yaws due to time constraints and instead still use our fork of Yaws 1.91 [18].

2.2.2.2 XMPP

Using ejabberd [19] as XMPP server was an obvious choice since ejabberd is the standard solution for XMPP servers and ejabberd is implemented in Erlang which led us to hope that it would play well with our code.

There are three ways to implement integration with ejabberd:

- Client — A bot that uses an Erlang client such as exmpp [20] can be used to connect to the server. All users send messages to the bot via the ejabberd server. e.g. bot@example.com. An issue the jabber client has, is that it doesn't scale. Using a single bot will mean all the users will have to be in the bot's roster (user list). Ejabberd doesn't handle the scaling of rosters very well (known to fail for $>$ 40k clients [21]).

- Component — A component is a trusted piece of an XMPP server that can send and receive arbitrary stanzas. In other words, we can add a module to ejabberd and define a virtual domain to get all messages that are sent to this domain. Because the name of a component is a domain (example: tt.localhost), a component can pretend to be many users. Any stanza addressed to service@tt.localhost will be delivered to tt.localhost no matter what the value is of 'service'.

- S2S (server to server) — This is the next step for those who need very large scale. This was not suitable for us since it requires us to learn a new protocol and is most possibly not necessary for our needs.

We decided to use component to communicate with users. We added a module to ejabberd that registers a hook to get all the messages that are sent to tt.localhost and forwards it to our backend. Ejabberd automatically

spawns a new process for a new user who sends messages to this component. If we need to communicate with a user, we use the corresponding process.

In this way, not only users that have an account on our ejabberd server, but also users that have account on any XMPP server can send their commands to the specified address and use our system.

2.2.2.3 SMTP

SMTP, together with POP and IMAP are the three most prevalent protocols for today's email servers. SMTP is used for sending emails from clients to servers while POP or IMAP are used for retrieving email from servers to clients. For our email server, we found three qualified candidates: Erlmail [22], gen_smtp [23] and erlang-smtp [24]. At first sight, Erlmail seemed to be the most competitive one since it supports all these three protocols when others support only one or two of them. However, since our SMTP frontend doesn't directly talk to email clients, we figured out that there's no need for an email retrieving mechanism for our servers. Therefore, we picked gen_smtp server as our SMTP frontend since all of its modules are only focused on SMTP. In our system architecture, the SMTP frontend is the bridge between the client side and the backend (see Fig. 2.3). Once the SMTP frontend receives an email from the client side, it will extract the email content and pass it to the backend controller. The controller will interpret and execute the valid orders carried by the email content, then respond to client side through the SMTP frontend again.

2.2.3 Controller

The main characteristic distinguishing requests is whether they belong to a session or not. So the controller has two kinds of workers. Those responsible for requests with a session and those for requests without.

The simple ones without sessions are completely stateless and can handle requests from any user. A fixed number of them is spawned on startup and will handle requests on arrival. Their main responsibilities are registration, login and the like. Fig. 2.4 illustrates this.

2.2.3.1 Session Management

The session management consists of a group of processes and a Mnesia table. Each session spawns its own session process and writes an entry into the Mnesia table, so that one such process exists for each logged in user. The Mnesia table is necessary to have a mapping between users and sessions. The processes handle all requests for their session and have the session data (user data, knowledge of how to push events to the frontend, etc). The reasons for that are concurrency issues (more on that in 2.2.6).

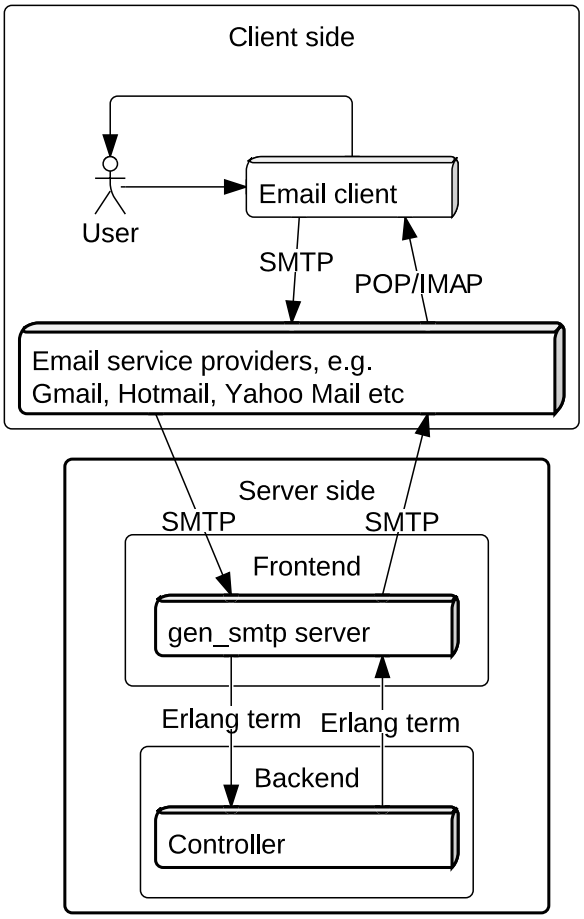

Figure 2.3: *SMTP communication between client side and backend.*

2.2.3.2 Access Control

Which users can perform which actions on the system is defined by a set of identities known as roles. The system has four roles:

- User — A registered member of the system who can create and play games.
- Blacklisted user — A user whose access to the system is blocked and his/her login credentials disabled. The Operator has the option to restore such a user's privileges.
- Moderator — A user with enhanced privileges to moderate games and help other users.
- Operator — An administrator who has complete control over the system.

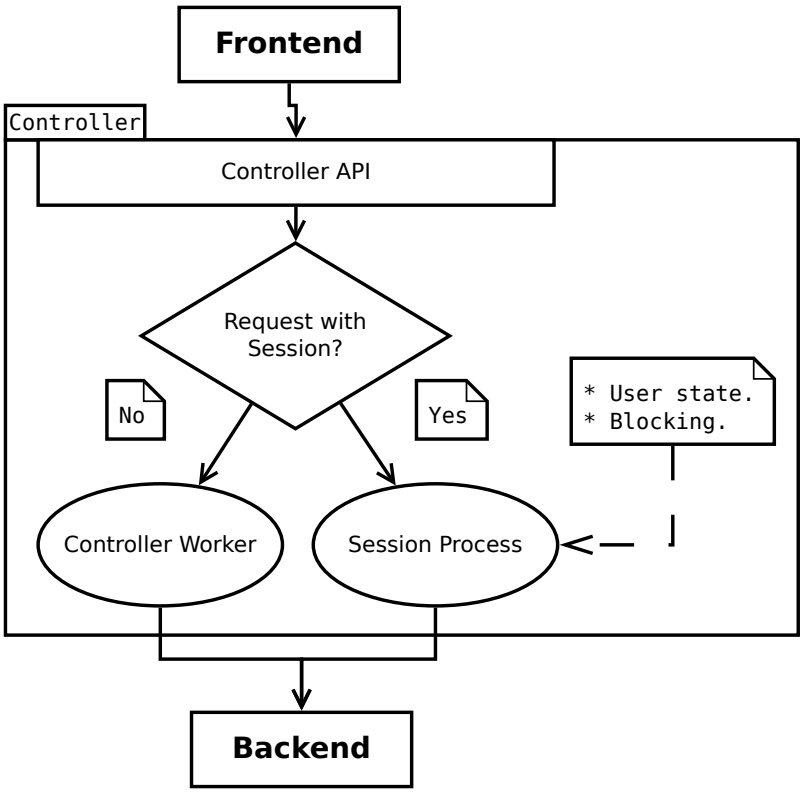

Figure 2.4: *Controller flow of a request with/without session.*

2.2.4 Riak

The choice of Riak [25] as database was a careful one: we evaluated Riak, couchDB [26] and Mnesia [27] but Riak was our favourite in the end — its scalability features are very simple to use and its potential for scalability is what we needed. Due to consistency issues, we had to resort to using Mnesia for sessions (see 2.2.1.4, 2.2.3.1) and the game joining processes (see 2.2.1.4), as it provides transactions.

Riak proved to be a good choice for us since its performance parameters (called "CAP controls" [28]) are very easy to tune and it is quite well documented how Riak behaves under load [29, 30]. The scalability problems we had came from using the search module too much (see 4.3.1.5).

Riak is a distributed key/value store that connects an arbitrary number of *nodes* to form a cluster. All of these nodes can — but don't have to — run on different machines (and do so in our standard setup). As soon as one node receives a write for one key/value pair, the node will make sure that the

data is written on a certain number of nodes. That number can be chosen from $[1 \dots N]$ where N is the number of nodes. No node in a Riak cluster is privileged, they all have exactly the same responsibilities which makes the architecture simpler. The ease of adding nodes to a running Riak cluster is one of its main advantages, data are redistributed in order to achieve fair distribution after one node is added. This can, of course, have performance implications but they were not measured by us.

Load testing was done in terms of how well the entire system behaves with different Riak configurations (see 4.3).

2.2.4.1 eLevelDB

Riak's storage backend was a problem. Riak has the feature to switch the storage backend — the way, key/value pairs are stored on a node. We initially used `eLevelDB` [31] because it supported secondary indices [32, 33, 34], a feature we intended to use. However, `eLevelDB` showed degrading performance in our context: the throughput of database operations was decreasing linearly over time, down to zero. We could not pinpoint the problem, so we had to switch the storage backend to `bitcask` [35], Riak's standard storage backend. Bitcask does not support secondary indices, and because of that it was necessary to use `riak_search` more — which led to scalability issues (see 4.3.1.5).

2.2.4.2 Mnesia

A user can access our system from multiple frontends resulting in concurrency issues. We dealt with them by making sure that the user can use only one frontend at a time.

Another issue was that multiple orders sent by the same user can arrive at the backend in a different sequence, thereby creating inconsistent data. This was fixed by serializing the user's commands.

The implementation of the above two solutions is not possible using Riak since it is an eventually consistent [36] database. Mnesia, which is distributed and has transactions, is the right fit.

We use Mnesia to keep track of user sessions. Combining this with the use of one Erlang process per session makes the implementation complete.

2.2.5 Backend

2.2.5.1 Game Managing

Managing games is split up in two main tasks: *game timing* and *order-processing*.

Game timing is implemented as a `gen_fsm` that changes states when a game phase needs to stop, eg. when the deadline for handing in orders is

over.

Before a phase is started, the rules processing is done by a module we call the "rule engine": the orders which were sent by the users before the deadline are read from the database and passed, along with the current game map, to the rule engine.

2.2.5.2 Messaging

Since communication is very important in the game — some even say, that the game is mostly about communication — the messaging module is a very central feature for us. We support two types of messages: *in-game* and *off-game*-messages.

In-game messages never involve the user nickname for tactical reasons: if someone remembers my nickname, he/she has an easier time to anticipate my moves since he/she will likely remember my actions in previous games. Or, worse: he/she might still hold a grudge against me. This is why you never communicate with players in-game by using their nickname, but by using their country.

Off-game messages on the other hand are sent to a nickname and the recipient will see the sender's nickname. The basic use case for the off-game messaging is giving users the chance to set up games for their friends and tell them about it.

2.2.5.3 Search

While search is an integrated part of our game application, it is important enough to mention here. Providing a search feature for users is important since it allows a user to find completed games or join an interesting game. To do this, users need to be able to search for games based on their properties (like all game parameters). For the implementation of search, we relied on the Riak extension `riak_search`.

Riak search is a search engine that is tightly coupled with the Riak datastore. We added a precommit hook so that whenever a new object is added or an old one updated, the object is indexed (tokenization with standard Lucene analyzers) and saved.

Riak provides a rich query language consisting of term searches, field searches, boolean operators and wildcards to fetch matching objects ordered by relevance.

This feature comes with a price though and should be used with caution. During load testing we discovered that it should be avoided for often updated data (See Sec. 4.3.1.5). As it has to re-index all fields of an object on every write it can kill the performance. Therefore we tried minimizing writes on search indexed data and if possible not to use the Riak search feature at all.

2.2.5.4 User Management

The user management module's purpose is to create, update, read, delete users in or from the database. The implementation is quite short and should contain few surprises.

2.2.6 Concurrency

2.2.6.1 Problem

Handling concurrency can be quite tricky. Especially with an eventually consistent database like Riak. The system is designed so that any node can handle incoming requests, since it is database driven, and on one node there are multiple workers, that can perform the same tasks. Thus two requests involving an update on the same key-value pair can, and will, end up on different workers or even nodes. This can lead to inconsistency, because they might have different work load and execute the tasks out of order. Fig. 2.5 illustrates this.

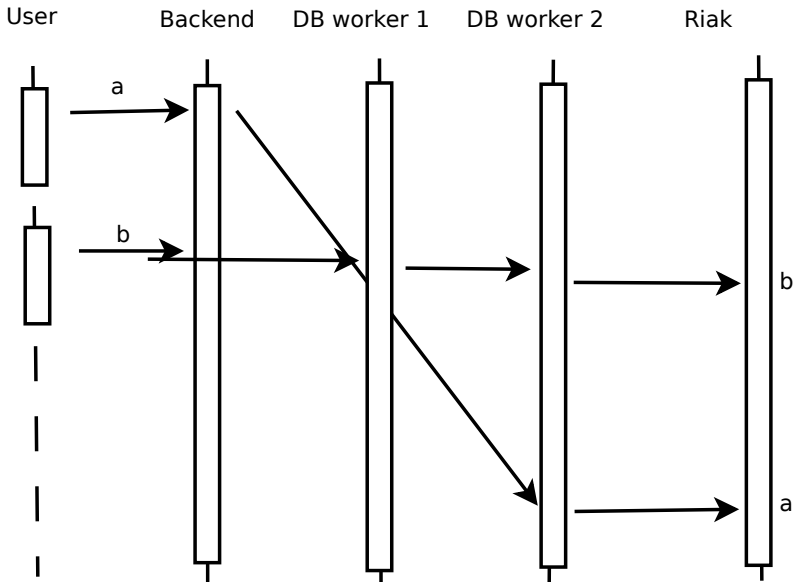

Figure 2.5: *First request written after second one, leading to inconsistency.*

Another problem is shown in Fig. 2.6. Two different nodes might end up writing to the same key without even knowing there was a concurrent write. Riak's CAP control specifies how many Riak nodes should respond a positive write before returning. In case 2/4 have to respond, two different writes can get the "ok". Our backends would not even know there was a

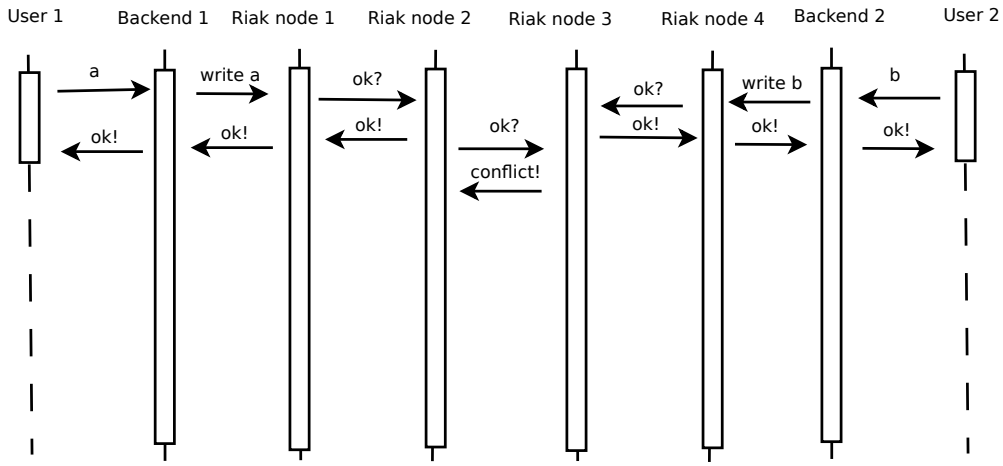

Figure 2.6: *Sibling creation.*

simultaneous write. Riak tries to automatically resolve this with vclocks [37]. If Riak cannot resolve this it will leave both values as so called siblings in the database. It is then up to the backend to resolve the conflict on the next read and write back the value which was decided to be the value that resulted after the conflict resolution.

2.2.6.2 Solution

A common solution for this problem is to serialize requests/writes to the same key. A sequential execution of tasks will always ensure the correct order and not create inconsistent data. In Erlang this is best done with processes. Each process has a message queue and can only handle one message after another. Thus requests are serialized if they have to go to the same process and if they are all matched against the same pattern in the receive statement.

In our system there are two kinds of interactions that need to be considered: single user concurrency and multi user concurrency.

Single user concurrency
Single user concurrency involves all the data that is written by only one user, like user profile updates and game orders. In order to serialize these there is one process for each session, and each user is only allowed to be logged in once at a time. To ensure only one active session the login has to consider concurrency as well. Fig. 2.7 shows how the login works. Any old session

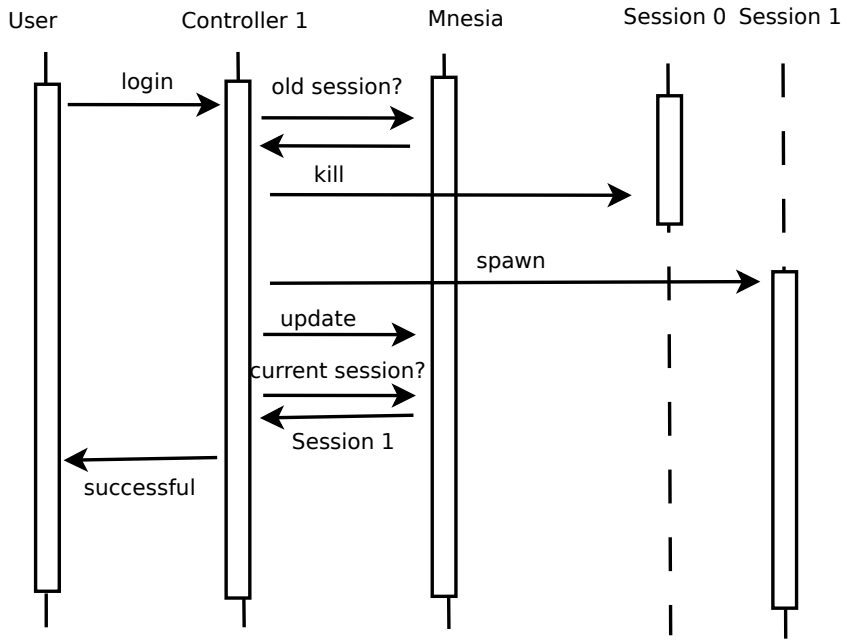

Figure 2.7: *User login.*

needs be terminated before we can start a new one. Also, we use Mnesia instead of Riak to store session information, as it supports transactions.

Unfortunately this still leaves one case. For example a user might re-login while his/her requests to update the profile is being handled and then tries to update the profile again before the previous update has been written. This is very unlikely to happen, as the user needs to do the update very quickly. However it still needs to be handled. We use a session history to resolve

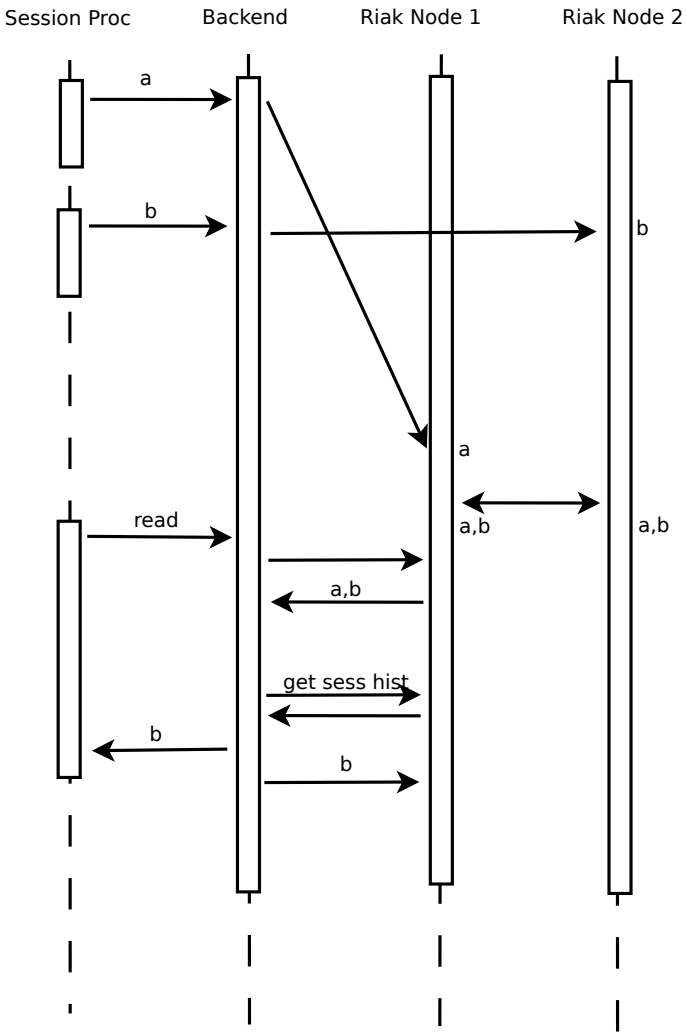

Figure 2.8: *Siblings resolutions using session history.*

siblings, which is updated during login. Additionally we store which session was responsible for the update. When the backend encounters siblings it then checks the session history which is the newer one and picks the corresponding value to be the correct one. Fig. 2.8 illustrates this.

Multi user concurrency
Multi user concurrency involves the data that can be written by multiple users, like joining or updating a game. In general we tried to design our database schema to avoid shared writable data as much as possible. For example the game orders for a certain phase could be stored as one value in Riak, instead we keep one value for each player. This leads to seven reads on phase change, but order writing does not have conflicts.

Nevertheless, we could not avoid it in all parts of the system. Multiple users might try to join the same game. There might be a conflict if they want to play the same country or only one spot is left. Furthermore, the creator of the game is not allowed to update the game once a player has joined it. Therefore we have a game joining process for every game that has not started yet. All join and update requests go through that process, thus ensuring consistency. To ensure only one such process per game we use Mnesia due to its transactions.

2.3 Code organization

The project consists of a set of Erlang applications logically grouped based on their functionalities. We have two more directories at the project level, one for external tests and the other for the release. Below is the high level directory structure of the project.

```
|-apps
|---cluster_manager
|---controller_app
|---datatypes
|---db
|---game
|---gen_moves
|---load_test
|---message
|---necromancer
|---service
|---smtp_frontend
|---system_manager
|---user_management
|---utils
```

```
|---web_frontend
|---xmpp_lib
|-ext_test
|---bench
|---fault_tolerance
|---smtp_integration_test
|---websocket_client
|---xmpp_integration_test
|-rel
```

A short description of each application can be found in Table 2.1.

Application	**Description**
cluster_manager	escript for management of the distributed cluster
controller_app	The controller application
datatypes	A central place for common configuration. It contains bucket names and records
db	The database wrapper that handles all the db requests
game	Contains game logic, game timer, rule engine and other game related code
gen_moves	Generates moves that can be used for load testing
load_test	Code used for load testing
message	Code used for handling messages
necromancer	Code used for resurrecting certain processes from dead VMs
service	OTP application library that provides functionality used by all service applications
smtp_frontend	Handles all the mail communication
system_manager	Single point of entry for configuring and controlling the whole system on a server.
user_management	Handles all user related functions
utils	Commonly used tools and utilities
web_frontend	Code for handling the web frontend, including client side code
xmpp_lib	Library for handling XMPP communication

Table 2.1: *Applications with their short descriptions.*

Additional notes on the code:

- General Erlang coding style and conventions were followed.
- The public API of all the modules has specs and edocs.

2.4 Supervision

A supervisor in Erlang is a process that supervises processes it has spawned. The supervisor can spawn new child processes and if one of them would die, it can act according to its configured restart-policy, for example, it could be configured to never restart children or alway restart them, independent of the "cause of death".

Each application (that is not only a library) in our backend and also the controller have a supervision tree structure that enables an operator to fine tune the number of workers of each application, it also makes it possible to inspect the status of applications on each server.

As seen in Fig. 2.9, there is one process group (see 2.2.1.3) for the application workers, and a "management" group. The processes in the worker group are the ones doing the heavy work in an application. The "manager" of each application in each node makes it possible to change the number of workers, it also makes it possible to traverse the group of managers to be able to get the status of each application on every node it is running on. The supervisors will always restart its workers if they die unexpectedly.

The game application's supervision tree is slightly different, as it also has a number of game timers, which don't belong to a process group as seen in Fig. 2.10. The game timer processes have their own supervisor. It will restart the processes in case if the process dies in an unnatural way, that is, a case in which neither the game has yet finished nor it has been stopped by an operator.

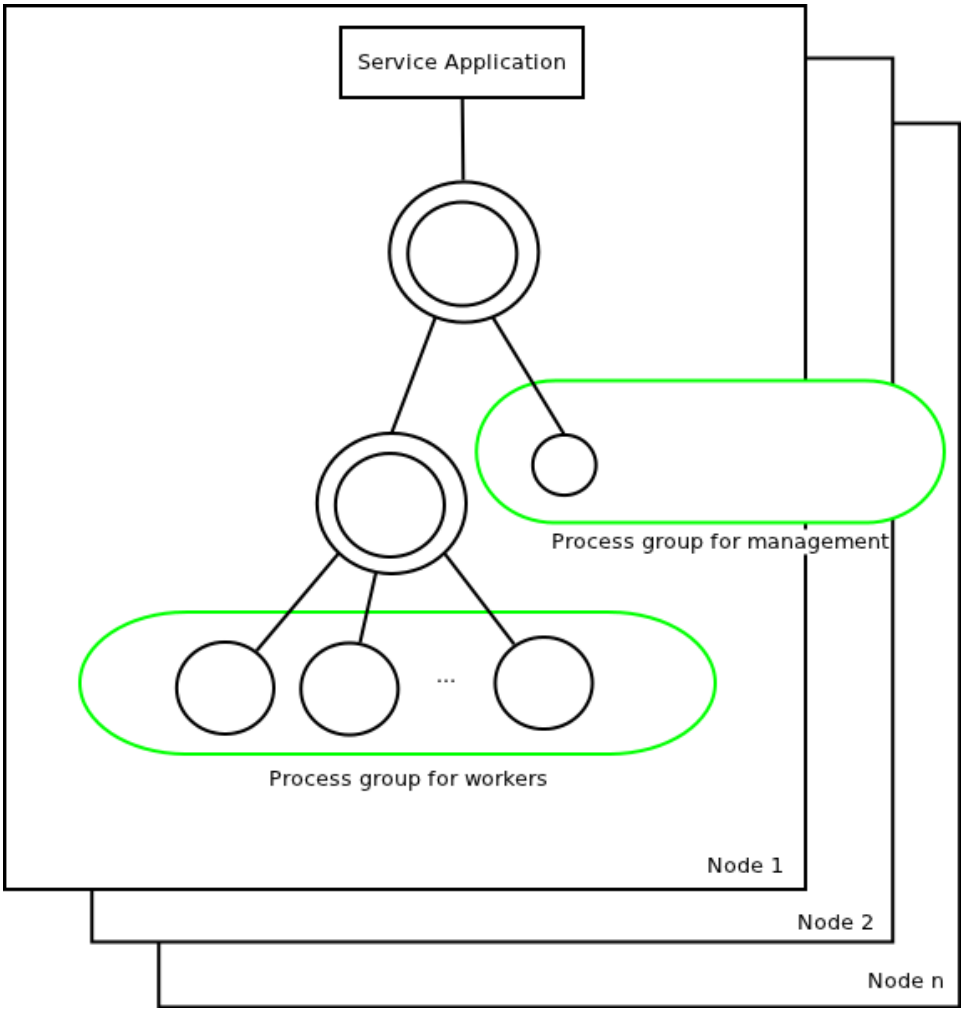

Figure 2.9: *General supervision structure for applications, double-rings are supervisors.*

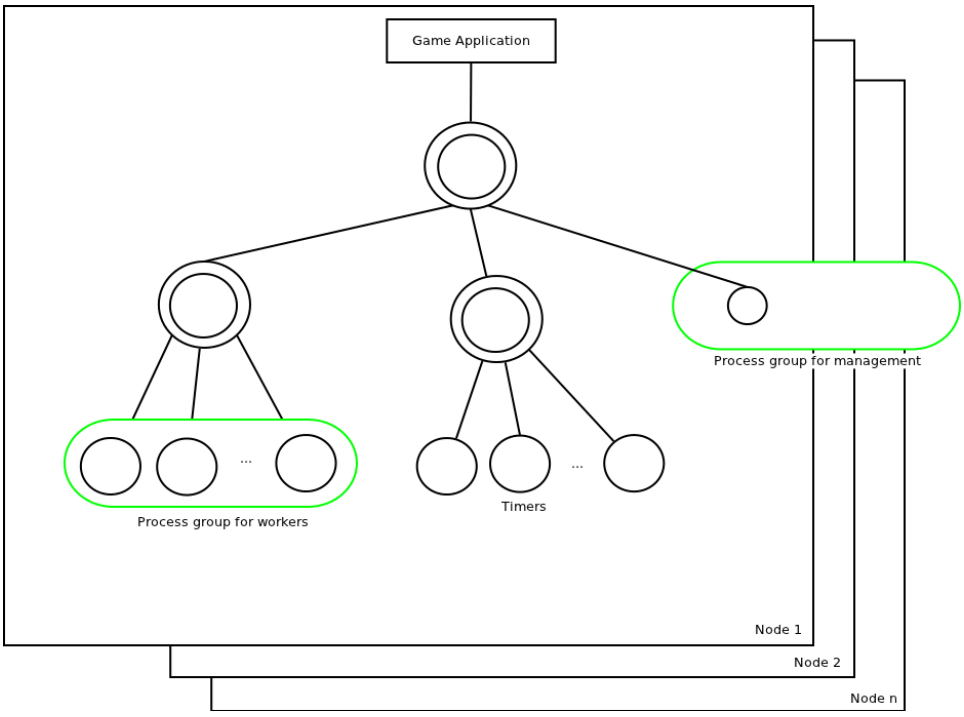

Figure 2.10: *Supervision tree for the game application.*

Chapter 3

Treacherous Talks

3.1 The Three Interfaces

3.1.1 HTTP

The user goes to the landing page, there he/she finds a link to register which will display a simple form for him/her to fill out. After the user is registered, he/she is able to log in using the login textfields on the landing page.

After logging in, the user is shown a dashboard page (see Fig. 3.1) where he/she is able to search for games, look at the games he/she is playing in (if any) and chat with other users in-game or off-game.

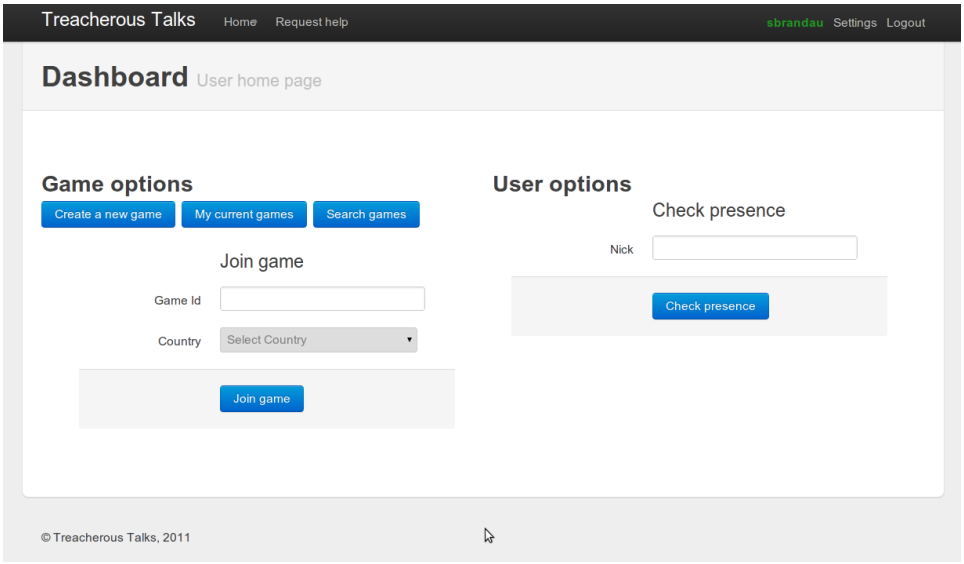

Figure 3.1: *The dashboard page.*

The web frontend has several key advantages over the two purely text based frontends:

- The user does not need to remember his/her session id.
- The user sees a graphical map (generated using a HTML5 canvas), see Fig. 3.2.
- It allows users to have a better gaming experience since the graphical map is interactive. By simply clicking and dragging the pieces on the map, corresponding orders will be automatically generated, see Fig. 3.3.

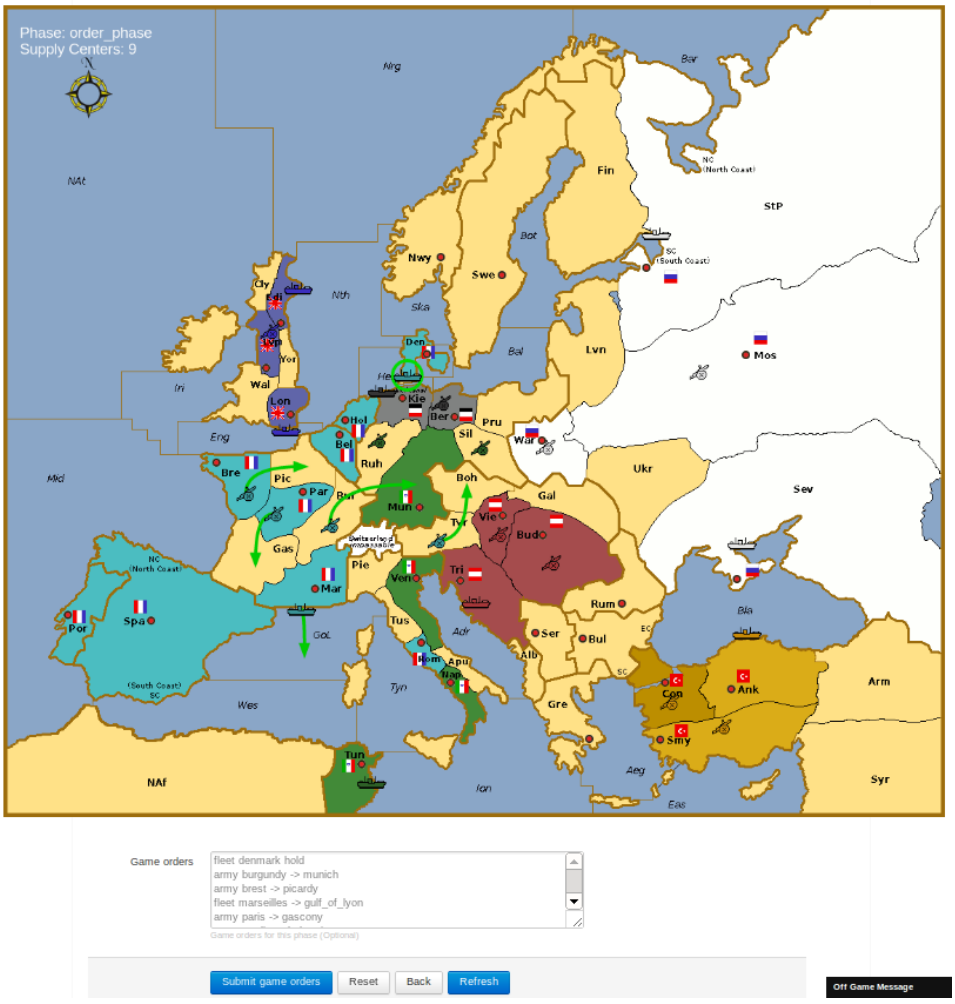

Figure 3.2: *The graphical map.*

3.1.2 XMPP

Users who log into our XMPP server, will get the address of the component that commands should be sent to in an instant message. If a user has an

Figure 3.3: *The generated orders.*

account on any other XMPP server, they can play the game by sending commands as instant messages to the server's user. For development, we used the user "service@tt.localhost", but this is only accessible from the host where ejabberd is running. To enable access from other ejabberd servers and connecting from other hosts, `tt.localhost` in the `tt_bot` module should be changed to a domain where ejabberd is configured to listen for connections.

If a user sends "HELP" or any unknown command to our component, it will return the list of all valid commands. All commands can be found in appendix C.2. Each command starts with the command keyword and must end with the "END" tag. If the user enters some text before the command keyword or after the end keyword, they are simply ignored. Users can get the list of all mandatory fields for each command by sending the command name and "END".

The user will get a unique session ID after logging in. The session ID should be stored, because it is necessary for all the future commands.

3.1.3 SMTP

Like most play-by-email games, we enable users to play the game using their own account from any email service provider. To connect to our game server, users need to write an email in a required format (see appendix C.2), then send it as an operation request to the email-address of our SMTP frontend server. After the email is sent, the users will soon receive an email in reply to their previous request from our game server. To users, the required format of the email text content is exactly the same as what they use in the XMPP

interface (see appendix C.2), but the difference is that users are likely to receive replies more instantly in XMPP than in SMTP.

3.2 Messages

In the web frontend, the user is shown two chat boxes: one for off-game chat and one for in-game-chat (they can be seen — minimized — in Fig. 3.1). The user manually has to enter the recipient (in in-game chat the recipient-country and the game ID). If the recipient is not online when a message is sent, it will be stored and delivered by the message application after the recipient logs in the next time.

Appendix C.2 contains more detailed information on interaction through the text based interfaces.

3.3 Playing

For game play, as is mentioned in previous sections, the web frontend provides an interactive interface which is much more convenient than typing text orders for beginners (Fig. 3.3).

On the contrary, the other two frontends (XMPP and SMTP) have to take text based orders. Instead of clicking, the user has to type them and additionally is required to supply his/her session ID which he/she receives after logging in. A full list of how to make text based plays can be found in appendix C.

Chapter 4

Evaluation and Testing

4.1 Overview

We approached testing very seriously from the start and are confident that this was one of the best decisions we made throughout the project. Unit tests are too low level to be covered here but they are of course there. We used EUnit for most of our testing and were generally happy with that choice except for one thing: EUnit declares a test as failed as soon as it runs for 5 seconds and there is no central way to change that behaviour. It's possible to change the default timeout for individual tests or test sets, but the code duplication in that case is, of course, sub-optimal. That 5 seconds "feature" was especially annoying in combination with continuous integration: our build server was very busy and therefore was interacting with our database a lot. When several builds were running at the same time, the database would get slower, therefore pushing tests over the time-limit, even though they ran perfectly fine on our local machines. Had we known this issue beforehand, we would have looked more into alternatives of EUnit.

4.2 Integration Tests

Our integration tests tried to cover everything from the frontends down to the database. It showed, that the XMPP frontend was the easiest to be tested, so our tests for SMTP- and web-frontend are only testing the basics — "it's there and it reacts" — while the XMPP tests send orders, register to the system, log in, log off, and so on. Because of the small size of our interfaces' IO parts (the parsing was handled by a dedicated module), we reach reasonable quality of tests (in terms of test coverage) using this approach while greatly reducing the amount of tests to be written.

The exmpp library [20] was used to do integration testing via the XMPP interface. Exmpp helped us to provide fully automatic and repeatable test scenarios. Adding new functionality to our system was not considered

complete until its corresponding integration test was added.

4.3 Load Tests

When starting to load test the system, we found it very hard to get meaningful data from our testruns. But even the first, quite informal, load tests resulted in very valuable information.

Load testing was never fully automated. Although that would be very useful, this would have been impossible for us since we would have needed a separate cluster to do that and just could not get that amount of hardware. A smaller automated test on one dedicated machine would have probably helped already but was not implemented due to time-constraints.

From the start of the load testing, practically no night was unused: tests were running through the night and were evaluated in the next morning.

It was necessary to write a considerable amount of load-testing-scripts that distributed our releases across a varying number of nodes, started and connected them and did the actual load generation. But: the time spent on this was time spent very well, since it ensured that performance drops because of single commits were noticed in several instances — and their cause analyzed.

4.3.0.1 Gathering data

We used Basho Bench [38, 39] to drive several *custom test drivers*. During a test run, test drivers are repeatedly *invoked* by Basho Bench, which records statistics about the invocations, such as the number completed in a period of time, the average time an invocation takes to complete, etc.

Each invocation of our test drivers performed a number of operations on our system, for example creating a game, writing several game orders and sending several messages. We called such a compound invocation a *flow*. While a driver which performs a single specific operation can be used to test one part of the system, we estimated a common usage pattern based on experience playing the game, and used flows to simulate use of the system as a whole.

Following a test run, we recorded system information provided by the operator overview features, including the number of *reductions* performed on that Erlang Runtime System, as reported by `erlang:statistics/1`. This records the number of functions, including Erlang built-in functions, called since the runtime system was started. Since we started new Erlang instances for each test run, we could compare the values between machines in a cluster to evaluate whether load is being balanced properly between machines in the cluster. This might appear to be a crude measure, and one function might be a lot more computationally intensive than another. However, as long as each machine is running the same set of applications (as was the case when

we compared reductions), each machine should perform the same number operations for long enough that random variation becomes negligible.

4.3.1 Results

The load tests had a big influence on the overall system. The main changes were made to how Riak was being accessed. We discovered which Riak features are expensive and even discarded some completely. This section describes those discoveries and justifies our decisions.

4.3.1.1 Load balancing

One of the first issues that appeared with load testing was that the system hardly scaled at all. Once we compared the number of reductions between machines in the test cluster, we realised one machine was getting much more load than others, explaining the bad scaling result. Test sessions were being allocated to random machines but we were using too few sessions, resulting in an uneven distribution between backends. By changing our load test setup to create more sessions, we ensured that the test load was evenly distributed among backends.

Once this was fixed, the number of reductions on different machines indicated even balance of work between machines.

4.3.1.2 Riak & backend relation

First of all we were interested in how to setup our cluster - how many Riak nodes does a backend require to perform its best and should Riak and backend nodes be on the same machine?

Fig. 4.1 shows the different setups we tested. The x-axis is the time the tests were running and the y-axis the throughput. The flow included all game play operations like game creation, joining, order submission and messaging. Registration and login are not part of it. The graphs are the different setups, they are labeled with XR or XB. Meaning on machine X there is a **R**iak node or a **B**ackend node [1].

As you can see most of them performed quite bad. Fig. 4.2 shows only the best setups. These three setups performed more or less the same. One of these setups is a cluster consisting of three machines each with a Riak and a backend node. Since this is also the simplest setup we decided to use setups with X machines each having a a Riak and a backend node in the future.

4.3.1.3 Degradation

The major issue with the previous results was the obvious performance degradation over time. Fig. 4.3 shows that this occured for any cluster size

[1] 1B 2R means there is a backend node on machine 1 and a Riak node on machine 2.

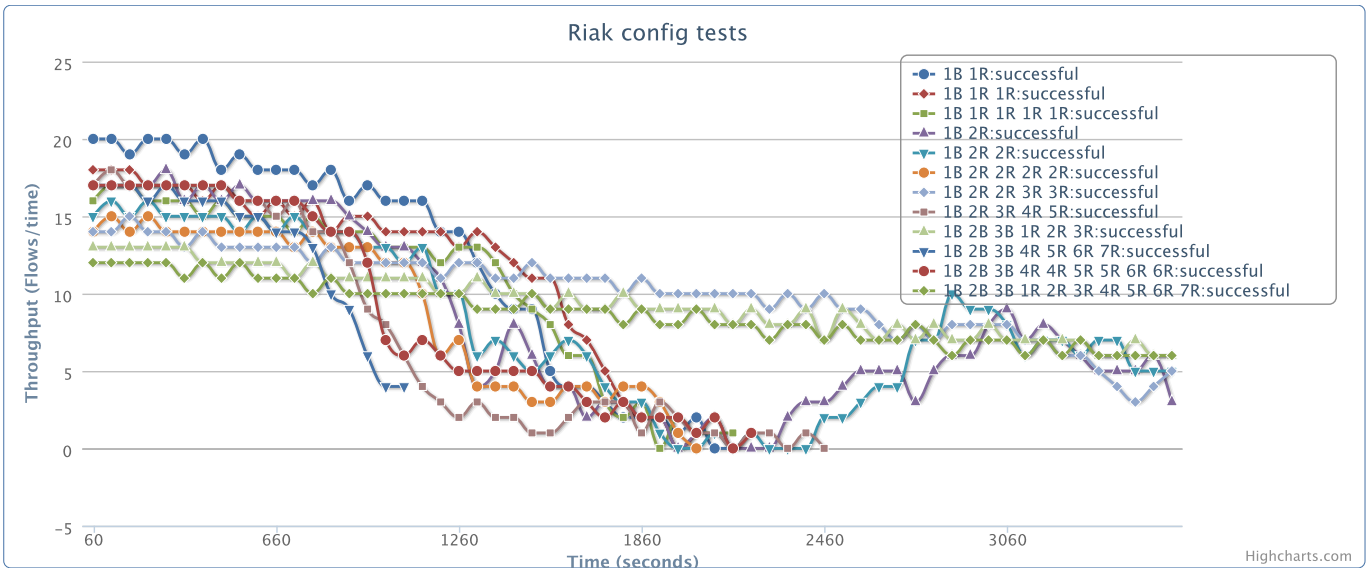

Figure 4.1: *Load tests of different cluster setups.*

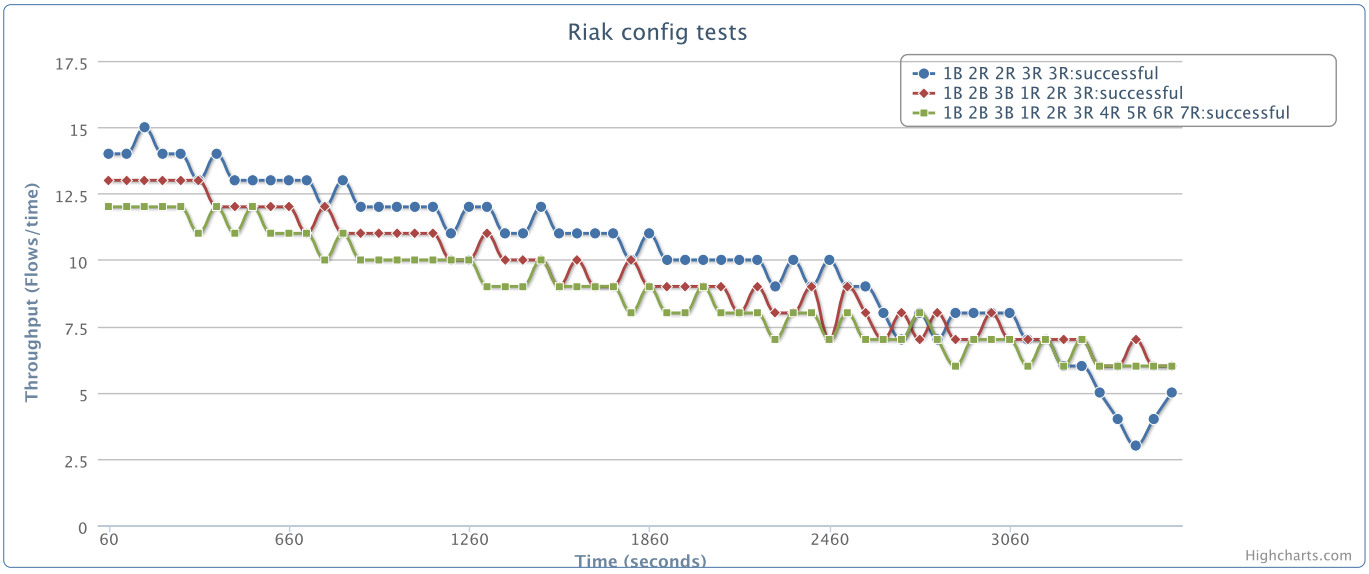

Figure 4.2: *The best graphs from Fig. 4.1.*

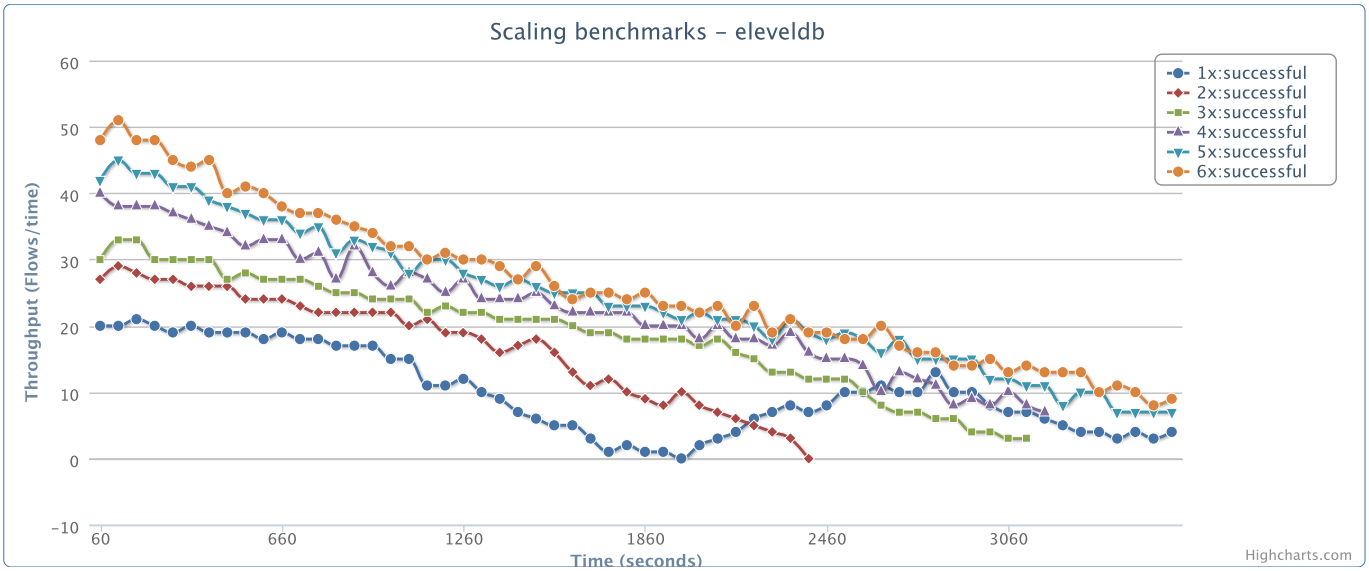

Figure 4.3: *Load tests for 1-6 machines.*

Preliminary prototype benchmarks did not show this pattern, which made us compare our system and the prototype we benchmarked. The main difference was the usage of eLevelDB as Riaks backend storage in the system, whereas the prototype used bitcask, the default storage backend.

The only reason we used eLevelDB was it being the only backend supporting secondary indices. We decided to do a comparison of those two backends, which required to remove the usage of secondary indices. Therefore we replaced it with Riak search. As you can see in Fig. 4.4 our assumption proved to be correct. Both start of the same but the throughput of eLevelDB version drops quickly, whereas the bitcask version remains almost stable.

4.3.1.4 CAP controls

Another small but effective change was the adaptation of the Riak CAP controls. Many writes to Riak have a quite low priority. The message logging for example is such a case. The system does not need to await confirmation of the write. Fig. 4.5 displays the increase in throughput this gave to our system.

4.3.1.5 Key filtering

Despite those improvements the system did still not scale as well as hoped. The Riak community[2] recommended us to take a closer look into our Riak search usage, since it has not been as well tested as other parts of Riak.

[2]IRC channel #riak on freenode.net

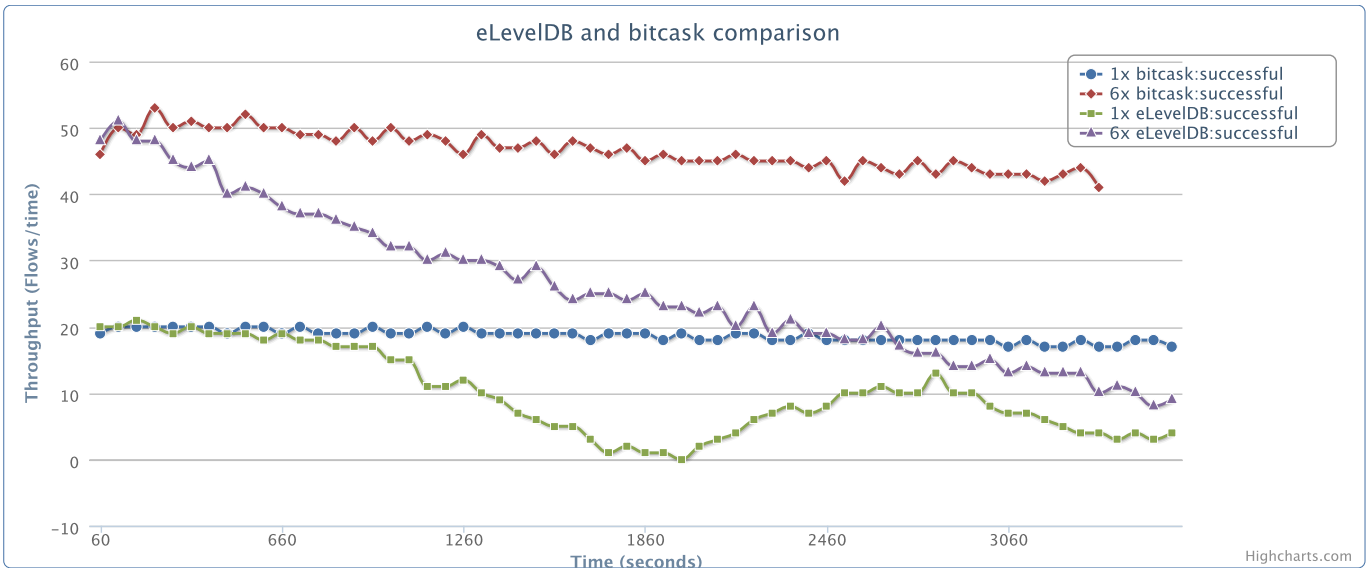

Figure 4.4: *Comparison of the storage backends bitcaks and eLevelDB.*

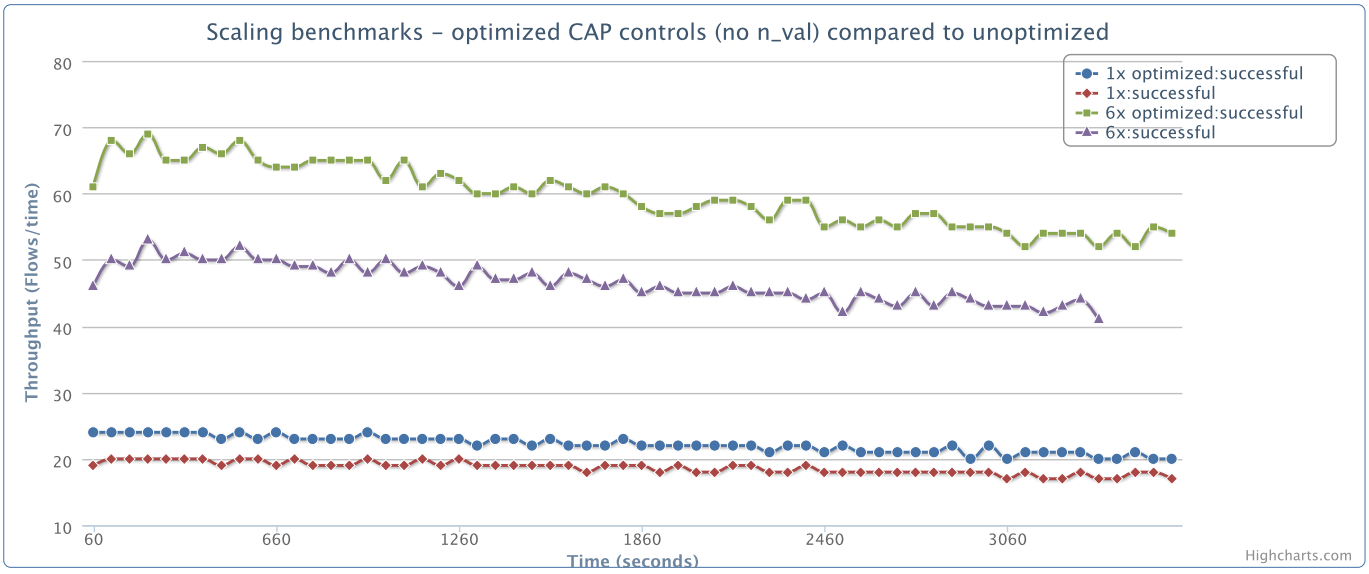

Figure 4.5: *Improvement by changed CAP controls.*

The system used Riak search quite extensively, especially after the change from eLevelDB. In many parts of the system it just replaced secondary indices. The full text search was not really necessary, as we just needed to find values without knowing the unique id. Therefore we tried to replace Riak search in parts with Riaks key filtering. For messages for example we introduced keys of the form $< unique - id > - < from > - < to >$. With key filtering it is then possible to query the messages for a certain user without the use of Riak search.

The advice from the Riak community turned out to be very valuable. Riak search should be avoided for values that are being written often, as the indexing is too expensive to scale well. Unfortunately though we no longer have the data to show a comparison graph.

4.3.1.6 Final results

Fig. 4.6 shows the performance of the final system. The y-axis is the amount

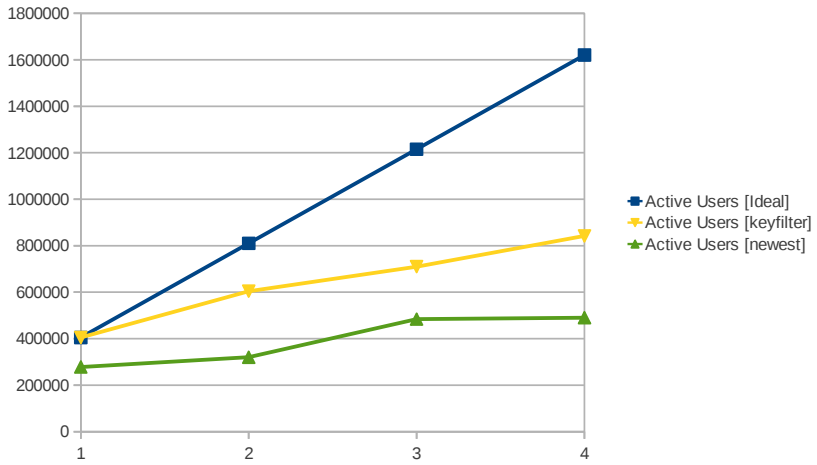

Figure 4.6: *Scaling.*

of model users the system can handle. Where a model user is assumed to perform about 20 requests a day and all users are distributed evenly throughout the day. The blue one is just for comparison and displays how perfect linear scaling would look like. The yellow graph shows the performance of the system right after we switched from riak search to key filtering. With one machine it can handle around 400k model users, and with 4 machines a bit more than double the amount. The green one is the system at the end of this project.

Unfortunately the performance dropped a lot. The cause for this is a single commit that introduced an increased usage of riak search again. Unfortunately there was no time left to get the system back to its previous performance. Still the system performs quite well and 400k model users is most likely more than the global count of diplomacy players.

4.4 Failure Tests

Our failure tolerance test starts up a cluster of two backends A and B and starts a game. The basic backend test interacts only via the Web Frontend for realism but simulates unexpected node or hardware failure by calling `erlang:halt()` on the relevant backend node.

Since we do not know on which backend the game has been started, we halt backend A. If fault tolerance works, we can now be sure that the game is on backend B (either it was there in the first place or it got restarted there).

We start the backend A again and then halt backend B. Now the game gets moved to backend A. This way, we can be sure that the game was moved at least once. If the game continues to run, the test was successful.

Instructions for running this test are in Sec. A.3.2.

Chapter 5

Related Work

Diplomacy has been around since the 1950s and has been played by email, snail mail and through web browsers. Obviously, we are not going to compare ourselves to the snail mail solution where a game host receives letters with move orders from the players and the players send each other letters.

Noteworthy browser based solutions are:

page	registered users
`http://webdiplomacy.net`	40613
`http://playdiplomacy.com/`	6849

The alternative email solutions seem to have even lower user numbers but precise details are hard to be found. Please note that the numbers in the table are about *registered* users, while our benchmarks are in terms of *active* users.

The advantages the existing solutions have, are generally better interfaces: the existing email systems are a bit easier to use since you reference games by name and not by a number and there is no session ID handling involved. These issues would be fixable with reasonable resources though.

They also provide several game modes (from minor changes to starting units to playing on a world map). We have the code to handle new maps, but we don't have the map data. Creating those would be an easy but work-intensive task since you have to specify all provinces, all connections, starting units, and so on.

Our rules are not perfect yet, they are still a bit rough around the edges but in probably one or two person weeks would work satisfyingly well. When it comes to load, we see no problem in handling all web diplomacy players worldwide (email- and browser-based) with one backend-machine and maybe more extra frontend machines.

So, in short: to fully catch up with the alternatives, we would need to invest in bug fixing and user interaction.

Chapter 6

Conclusions and Future Work

We are very happy with the work we have done and that the product is, even though the seemingly endless scope, quite close to being "ready". We found that Riak's performance depends heavily on the usage pattern. Particularly, we found that Riak Search is best for storing data that is not written very frequently, as discussed in Sections 2.2.5.3 and 4.3.1.5.

Despite all this, the work left to be done is considerable: An AI was in the original requirements but was not delivered due to time constraints. Bug fixing in the rule engine is necessary. Scalability could be improved by applying tracing to the cluster which would be lots of interesting work. A more polished interface, that is more appealing to the eye, is something that could be worked on.

In order to make the game playable by the public, a better system to find and join games would probably be necessary in the web frontend — and we would need to do lots of hallway testing and/or maybe publish an alpha version in order to collect feedback.

One more thing would be to think a lot about security — not much thought has been spent on it yet.

The rule engine does not support multi-fleet convoys yet but they are important for the game tactics (the classic Italian opening called "lepanto opening" [40] uses it, for instance). This could be done quite easily in the rule engine.

Appendix A

Installation Instructions

A.1 Requirements

Treacherous Talks was developed using Erlang/OTP version R14B03 on Ubuntu 11.04 (AMD64). It has not been tested with other versions of Erlang/OTP or other Linux distributions. Since some of our dependencies, notably Riak, currently doesn't support Erlang/OTP R15, we have not been able to test the system under that version of Erlang/OTP.

Since all Erlang-releated packages in Ubuntu 11.04 are out-of-date, we opted for compiling and installing Erlang/OTP by hand on our development machines. In general, we recommend that users do this as well since it is the easiest way to get all of the required parts of Erlang/OTP installed on your system.

We will not provide you with build instructions for the Erlang/OTP distribution, but under Ubuntu 11.04 you will most likely need the `make`, `gcc`, `perl`, `m4`, `ncurses-dev` and `libssl-dev` packages available before attempting to build it.

Before you can build Treacherous Talks itself, a few extra libraries and tools need to be present on your machine:

Name	Version	Ubuntu package name
libexpat	> 1.95	libexpat-dev
libxml	-	libxml2-dev
libpam	-	libpam0g-dev
git	-	git
wget	-	wget

There are two deployment options regarding the database Riak: running a standalone Riak installation or distributing it alongside the Treacherous Talks release package. Here we assume that you want to distribute Riak along with the release package. This involves building Riak as a part of the system build process.

In order to build Riak you will need the GNU C++ compiler (`g++`) installed and the package `libstdc++6-VERSION-dev` where `VERSION` is the current version in the package repository (in Ubuntu 11.04 `VERSION` is 4.5). For some reason there are no generic packages any more in Ubuntu, so you need to specify the version.

A.2 Building

With all dependencies and required tools in place, building Treacherous Talks should be straightforward. First you need to get hold of the source code. The simplest way is probably to download a tarball from Github. Download it from `https://github.com/treacheroustalks/Treacherous-Talks/tarball/master`. Untar the file and enter the source code directory.

First we build Riak since we want to distribute it together with the Treacherous Talks release package. There is support in the Treacherous Talks Makefile for building Riak, just run `make riak_release`. This will download, compile and release Riak. It then moves the release package into the directory `system-release` in the top-level source code directory.

Next it's time to build Treacherous Talks. This is done by executing `make`. This will invoke the build tool Rebar [41] that is present in the source code directory. Rebar will download all dependencies and compile everything in the right order. This step can take a while to complete.

The final two steps after compiling is to make an Erlang release and create a tarball that contains everything needed to run the system. By executing `make release` a release package will be created and placed into the `system-release` directory alongside Riak. After this step, we create a tarball of everything inside that directory with `make tar_release`. The resulting tarball is outputted to the `system-release` directory.

A.3 Testing

A.3.1 Unit and Integration Tests

After building, unit and integration tests can be run together using

```
make test
```

This is equivalent to running `make unittest inttest`.

A.3.2 Node Failure Tolerance Tests

An escript [42] `fault_tolerance` is generated when building a full release. This allows the tests in `ext_test/fault_folerance` to be run from the command line e.g.

```
./system-release/tt/bin/fault_tolerance   tt.config
```

where `tt.config` is a cluster configuration file. This requires the same setup as the Cluster Manager (see A.5) needs for starting a cluster. The `fault_tolerance` escript will exit with a non-zero exit status if any tests failed and outputs debugging information to the console.

A.4 Installing from a release tarball

The resulting tarball from the build step is self-contained and can be freely moved to other computers running the same version of the operating system. To install the system, simply extract the tarball to a location of your choice. As usual in Unix-like systems, take care to have the ownership of the files setup correctly so that the user running Riak and Treacherous Talks has the correct permissions on all directories and files. The simplest is to have the user executing the program owning the files.

If you intend to allow any part of the system to communicate over privileged ports, ensure that the user has privileges to do so.

A.5 Setting up and starting the System Manager

There is an application called the System Manager in Treacherous Talks. The purpose of this application is to configure, start and stop Riak and the rest of the Treacherous Talks system on a single machine. The System Operator uses the escript called Cluster Manager to interact with a cluster of System Managers via RPC calls.

Before we can start and configure the system we must set up and start the System Manager itself. There is only one major setting one can change, and that is the Erlang node name. Enter the directory where the release tarball was extracted. Edit the file `system-release/tt/etc/nodename.system_manager` and change the domain name or ip address as needed (the part after the @-sign). Note that you must set domain to something externally available if you intend to run the Cluster Manager on another machine.

When the node name is set, the System Manager is ready to be started. This is done by proceeding to the directory where the release tarball was extracted and then executing the command `system-release/tt/bin/system_manager start` in a shell.

A.6 Creating a system-wide configuration file

The main purpose of having both a System Manager and a Cluster Manager is to simplify the task of configuring and controlling a cluster of nodes running

on different machines. Even if you only run the releases on a single machine, it simplifies the task of handling the Treacherous Talks system and Riak.

To do this, a single configuration file is used. The file specifies for the Cluster Manager where the different System Managers are, what releases a single machine should run and the configuration of such releases. The configuration is given as a list of host tuples.

Each host tuple describes a host (machine) and contains the domain or ip address it has, the name of the System Manager and another list of release tuples.

Each release tuple contains a release name (riak, backend, smtp_frontend, xmpp_frontend, web_frontend), the actual node name for that release and a list of configuration options. These configuration options correspond exactly to the ones available in the release configuration file in the `etc` directory of a release package. Here is a minimal example for running all releases on a single machine:

```
[{host, "127.0.0.1", "system_manager",
  [{release, riak, riak,
    [
     {riak_core, [{http, [{"127.0.0.1", 8091}]}]},
     {riak_kv, [{pb_ip, "127.0.0.1"}, {pb_port, 8081}]},
     {riak_search, [{enabled, true}]}
    ]},
   {release, backend, backend,
    [
     {db, [{riak_ip, "127.0.0.1"}, {riak_database_port, 8091},
           {riak_protobuf_port, 8081}]}
    ]},
   {release, smtp_frontend, smtp_frontend, []},
   {release, xmpp_frontend, xmpp_frontend, []},
   {release, web_frontend, web_frontend, []}
  ]}
].
```

For more information on the available configuration options, please see the example configuration file in `systemrelease/tt/etc/example.config`.

A.7 Using the Cluster Manager

When the system-wide configuration file is finished, it is time to put it to use. The Cluster Manager is a small command-line tool (an escript) that reads the configuration file and connects to running System Managers to enforce the state given in the configuration file and/or performs a specific action.

The actions performed is controlled by adding switches when invoking the command. One can use multiple switches to perform more than one action. A short description of the more common ones follow.

By using the `--setconfig` switch, the Cluster Manager will try to connect to all specified System Managers and give them the configuration they should apply. Note that this does not automatically apply the configuration to a running system. To be able to use the new configuration, the system must be restarted.

The `--join` switch is intended to be used whenever a Riak node is added to the cluster. It will make any new Riak nodes join the already present nodes. This is only necessary when you add a new node, not after stopping or starting a Riak node.

Finally, the `--start` and `--stop` switches starts and stops the needed releases, including Riak if that is needed. To see what releases are running and responding across a cluster, the `--ping` switch comes handy.

To run the Cluster Manager, first go to the directory where the release tarball was extracted and then execute `system-release/tt/bin/cluster_manager`. You also need to supply the name of the configuration file to use and the action to perform.

Here is an example run of the Cluster Manager:

```
$ ./bin/cluster_manager --setconfig --start etc/example.config
update_config on "127.0.0.1" was ok
start_release riak on "127.0.0.1" was ok
start_release backend on "127.0.0.1" was ok
start_release web_frontend on "127.0.0.1" was ok
start_release xmpp_frontend on "127.0.0.1" was ok
start_release smtp_frontend on "127.0.0.1" was ok
informing 'backend@127.0.0.1', result was: ok
$
```

A.8 Running on a non-bundled Riak installation

There are a few additional things to consider if you want to run Treacherous Talks on top of an already running Riak installation instead of using the bundled one.

Some parts of Treacherous Talks are dependent on Riak Search and special search schemas. These schemas need to be installed on each Riak node in a cluster before starting the Treacherous Talks system. This is done automatically when using the bundled Riak instance and configuring it via the Cluster Manager.

When using a non-bundled Riak installation you need to do this yourself. Each schema in the `system-release/tt/riak` directory must be installed

in each Riak node. Please consult the Riak manual[1] for the exact steps you need to perform to accomplish this.

You can still use the Cluster Manager for configuring backends and frontends in this scenario. Remove the riak tuple from the example configuration and ensure that the db application in the backend knows how to connect to an already installed Riak node. Ensure that Riak is started before the rest of the system is started.

[1] http://wiki.basho.com/Riak-Search—Schema.html

Appendix B

Maintenance Instructions

B.1 Adding a host to a running cluster

Sometimes the need arise to add another server to an already running cluster. This is a rather simple operation when using the Cluster Manager.

First, ensure that the system has been properly installed (see A.4) on the new node. Start the System Manager if not already started on that node. Then add the configuration of the new node to the system-wide configuration file.

Finally, run the Cluster Manager with the `--setconfig` and `--start` switches. The result will be that all releases defined in the configuration file will be started, included any new ones. If a release is already running nothing will be done to it and it will be reported by the Cluster Manager as {`error`, `release_is_already_started`}. This is a harmless message indicating that nothing was done for that release.

B.2 Removing a host from a running cluster

Removing a server from a running cluster is normally a straightforward procedure, but a bit more complicated than adding a server. You cannot simply remove its entries from the system-wide configuration file and then expect the Cluster Manager to stop it for you since the configuration declares what *should* be running on the cluster, not the current state of the cluster.

You have three choices for removing a host:

1. Restarting the whole cluster.
2. Stopping the system manually on the specific host.
3. Creating a special configuration file for stopping a specific host.

The first solution is simple: just remove the host from the configuration file, stop and start the cluster using the Cluster Manager. Of course, this has the disadvantage of making the system unavailable for some time.

The second solution, stopping the system manually, is a more attractive solution. To do this, go to the installation directory on the host you want to remove. For each release running on the host, run the corresponding executable with the argument `stop`. Example: `system-release/tt/bin/backend stop`.

The third solution is probably the best one, but requires some effort. Copy your old configuration file before removing the entry describing the host you want to remove. Remove everything else in the copied configuration file except for the host you want to remove from the cluster. Now use this configuration when running Cluster Manager with the `--stop` switch. Only the host you want to remove has now been stopped.

Nothing else needs to be done for the Treacherous Talks system when removing a host. Any games running on the now stopped backend will be automatically moved to other backends.

If you want to remove a Riak node permanently, you should tell the Riak cluster about it so that it can redistribute the data on that node onto other nodes in the cluster. Please see the Riak manual[1] for more information.

B.3 System Operator Interface

Being a system operator opens up a few special features, some are available as text based commands but all are available in the web interface of the operator:

- Adding or removing moderators (web only).
- Stop games (web only).
- Overview games including moves and messaging (web only).
- Blacklist or whitelist players.
- Receive reports about issues from players (web only).
- Send messages to players in any game.
- View the status of the system (web only).
- View the status of Riak.

The status of the system gives information about which applications are running on which nodes, the number of workers they have and information

[1] http://wiki.basho.com/Adding-and-Removing-Nodes.html

about their message queue lengths. This can be used to determine the health of the system and see if an application needs to be started on another node or perhaps if it could be fixed by increasing/decreasing the number of workers.

B.3.1 Moderators

A moderator is a priviledged user which can access a moderator page in the web interface.

The following features are available for moderators:

- Blacklist or whitelist players.
- Send messages to players in any game.
- Receive reports from players reporting other players (web only).

Appendix C

Text based commands

C.1 Playing the game

The web interface is the only one to provide means to interactively select orders, but it is also possible to give text based orders as through IM and email.

Writing orders

Alternative ways to write a move order (also used for retreat orders):

```
army warsaw move prussia
a war m pru
a war - pru
a war -> pru
```

Alternative ways to write a convoy order:

```
fleet north_atlantic_ocean convoy army london move norwegian_sea
f nth c a lon m nrg
f nth c a lon - nrg
f nth c a lon -> nrg
```

Alternative ways to write a hold order:

```
army london hold
a lon h
```

Alternative ways to write a support order:

```
army galicia support budapest
a dal s a bud
```

Alternative ways to write a build order:

```
build army munich
b a mun
```

List of provinces and water bodies and their abbreviations

adriatic_sea - adr	aegean_sea - aeg	albania - alb
ankara - ank	apulia - apu	armenia arm
baltic_sea - bal	barents_sea - bar	belgium - bel
berlin - ber	black_sea - bla	bohemia - boh
brest - bre	budapest - bud	bulgaria - bul
burgundy - bur	clyde - bly	constantinople - con
denmark - den	eastern_mediterranean - eas	edinburgh - edi
english_channel - eng	finland - fin	galicia - gal
gascony - gas	greece - gre	gulf_of_bothnia - bot
gulf_of_lyon - gol	helgoland_bight - hel	holland - hol
ionian_sea - ion	irish_sea - iri	kiel - kie
liverpool - lvp	livonia - lvn	london - lon
marseilles - mar	mid_atlantic_ocean - mid	moscow - mos
munich - mun	naples - nap	north_africa - naf
north_atlantic_ocean - nat	north_sea - nth	norway - why
norwegian_sea - nrg	paris - par	picardy - pic
piedmont - pie	portugal - por	prussia - pru
rome - rom	ruhr - ruh	rumania - rum
serbia - ser	sevastopol - sev	silesia - sil
skagerrak - ska	smyrna - smy	spain - spa
st_petersburg - stp	sweden - swe	syria - syr
trieste - tri	tunis - tun	tuscany - tus
tyrolia - tyr	tyrrhenian_sea - tyn	ukraine ukr
venice - ven	vienna - vie	wales - wal
warsaw - war	western_mediterranean	yorkshire - yor

C.2 Commands for IM and Mail

The following commands are available for IM and mail users:

```
REGISTER
required: NICKNAME, PASSWORD, EMAIL, FULLNAME
optional: CHANNEL

UPDATE
required: SESSION
optional: PASSWORD, EMAIL, FULLNAME

LOGIN
required: NICKNAME, PASSWORD

LOGOUT
required: SESSION

ORDER
required: SESSION, GAMEID

CREATE
required: SESSION, GAMENAME, PRESSTYPE, ORDERCIRCLE,
          RETREATCIRCLE, GAINLOSTCIRCLE, WAITTIME
optional: PASSWORD, DESCRIPTION, NUMBEROFPLAYERS

RECONFIG
required: SESSION, GAMEID
optional: GAMENAME, PRESSTYPE, ORDERCIRCLE, RETREATCIRCLE,
          GAINLOSTCIRCLE, WAITTIME, PASSWORD, DESCRIPTION,
          NUMBEROFPLAYERS

OVERVIEW
required: SESSION, GAMEID

VIEWCURRENTGAMES
required: SESSION

JOIN
required: SESSION, GAMEID, COUNTRY

SEARCH
required: SESSION
optional: GAMEID, GAMENAME, DESCRIPTION, PRESSTYPE, STATUS,
```

```
          ORDERCIRCLE, RETREATCIRCLE, GAINLOSTCIRCLE,
          WAITTIME, NUMBEROFPLAYERS

MESSAGE (in game messages)
required: SESSION, GAMEID, CONTENT
optional: TO

MESSAGE (out of game messages)
required: SESSION, TO, CONTENT

GETPROFILE
required: SESSION

GETPRESENCE
required: SESSION, NICKNAME

REPORTPLAYER
required: SESSION, CONTENT

REPORTISSUE
required: SESSION, CONTENT


MODERATOR and OPERATOR only commands:

POWERMESSAGE
required: SESSION, GAMEID, CONTENT
optional: TO

BLACKLIST
required: SESSION, NICKNAME

WHITELIST
required: SESSION, NICKNAME
```

The general format to send any command is

```
COMMAND
FIELD: <data>
END
```

where `FIELD` is the name of the field and `<data>` is the input from the user, *all commands and fieldnames must be in capital letters.*

Example commands

```
REGISTER
NICKNAME: bob
PASSWORD: secret
EMAIL: bob@email.com
FULLNAME: Bob Cat
END

MESSAGE
SESSION: g2dkABFiYWNrZW5kQDEyNy4wLjAuMQAAA+QAAAAAAQ==
GAMEID: 12345
TO: england, france
CONTENT:
Hi! Would you like to help me take down russia?
END
```

Order writing is done as in the previous section, as content for the `ORDER` command:

```
ORDER
SESSION: g2dkABFiYWNrZW5kQDEyNy4wLjAuMQAAA+QAAAAAAQ==
GAMEID: 12345

army ven -> tri
army tyr support a ven -> tri
END
```

References

[1] Erlang solutions, . URL http://www.erlang-solutions.com/. Accessed 9 Jan 2012.

[2] Allan B. Calhamer. Diplomacy. URL http://www.wizards.com/default.asp?x=ah/prod/diplomacy.

[3] Internet Engineering Task Force (IETF). The WebSocket Protocol, 2011. URL http://tools.ietf.org/html/rfc6455. Accessed 5 Jan 2012.

[4] E. Gamma. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley professional computing series. Addison-Wesley, 1995. ISBN 9780201633610.

[5] Seved Torstendahl. Open telecom platform. URL www.erlang.se/publications/ericsson_review_otp_1997012.pdf. Accessed 8 Feb 2012.

[6] Process groups. URL http://www.erlang.org/doc/man/pg2.html. Accessed 5 Jan 2012.

[7] Michael Stonebraker. The case for Shared Nothing. *Database Engineering*, 9:4–9, 1986.

[8] Yaws web server, . URL http://yaws.hyber.org/. Accessed 8 Feb 2012.

[9] Nitrogen - nitrogen web framework for erlang. URL http://nitrogenproject.com/. Accessed 8 Feb 2012.

[10] Ajax technology. URL http://www.webhostdesignpost.com/website/webtechnology-ajax.html. Accessed 8 Feb 2012.

[11] D. Flanagan. *JavaScript: The Definitive Guide*. O'Reilly Media, Inc, Sebastopol, CA, sixth edition, 2011.

[12] The jQuery Project. jquery. URL http://jquery.com/. Accessed 5 Jan 2012.

[13] Twitter Inc. Bootstrap toolkit. URL http://twitter.github.com/bootstrap/. Accessed 5 Jan 2012.

[14] Simple bridge. URL https://github.com/nitrogen/simple_bridge. Accessed 8 Feb 2012.

[15] The websocket protocol draft-ietf-hybi-thewebsocketprotocol-10, 2011. URL http://tools.ietf.org/html/draft-ietf-hybi-thewebsocketprotocol-10. Work in progress.

[16] Yaws fork websocket_hy10 branch, . URL https://github.com/jbothma/yaws/tree/websocket_hy10. Accessed 9 Jan 2012.

[17] Takeshi Yoshino. New websocket protocol: Secure and extensible, 2011. URL http://blog.chromium.org/2011/08/new-websocket-protocol-secure-and.html. Accessed 9 Jan 2012.

[18] Treacherous talks yaws fork. URL https://github.com/treacheroustalks/yaws. Accessed 9 Jan 2012.

[19] ejabberd a jabber/xmpp instant messaging server. URL http://www.ejabberd.im/. Accessed 10 Jan 2012.

[20] The erlang library for xmpp. URL https://support.process-one.net/doc/display/EXMPP/exmpp+home. Accessed 10 Jan 2012.

[21] Jack Moffitt. Thoughts on scalable xmpp bots, 2008. URL http://metajack.wordpress.com/2008/08/04/thoughts-on-scalable-xmpp-bots/.

[22] Erlmail, . URL http://code.google.com/p/erlmail/. Accessed 8 Feb 2012.

[23] A generic erlang smtp server and client. URL https://github.com/Vagabond/gen_smtp. Accessed 8 Feb 2012.

[24] Erlang smtp and pop3 server, . URL https://github.com/tonyg/erlang-smtp. Accessed 8 Feb 2012.

[25] Riak, . URL http://wiki.basho.com/Riak.html. Accessed 8 Feb 2012.

[26] couchdb. URL http://couchdb.apache.org/. Accessed 8 Feb 2012.

[27] Mnesia. URL http://www.erlang.org/doc/man/mnesia.html. Accessed 8 Feb 2012.

[28] Cap controls. URL http://wiki.basho.com/Tunable-CAP-Controls-in-Riak.html. Accessed 5 Jan 2012.

[29] Mozilla Inc Daniel Einspanjer. Benchmarking riak for the mozilla test pilot project. URL http://blog.mozilla.com/data/2010/08/16/benchmarking-riak-for-the-mozilla-test-pilot-project/. Accessed 10 Feb 2012.

[30] Joyent. Riak smartmachine benchmark: The technical details. URL http://joyeur.com/2010/10/31/riak-smartmachine-benchmark-the-technical-details/. Accessed 10 Feb 2012.

[31] eleveldb. URL http://wiki.basho.com/LevelDB.html. Accessed 8 Feb 2012.

[32] R. Elmasri and S. Navathe. *Fundamentals of database systems*. Pearson Custom Computer Science Series. Pearson/Addison Wesley, 2007. ISBN 9780321369574.

[33] Querying riak just got easier: Secondary indices in riak, . URL http://www.slideshare.net/rklophaus/querying-riak-just-got-easier-introducing-secondary-indices. Accessed 10 Feb 2012.

[34] Secondary indices. URL http://doc.gnu-darwin.org/am/second.html. Accessed 8 Feb 2012.

[35] Bitcask. URL http://wiki.basho.com/Bitcask.html. Accessed 8 Feb 2012.

[36] Eventual consistency. URL http://wiki.basho.com/Eventual-Consistency.html. Accessed 8 Feb 2012.

[37] E. Florenzano. Nonrelational databases. In J. Allspaw and J. Robbins, editors, *Web Operations: Keeping the Data on Time*, pages 247–262. O'Reilly Media, Inc, Sebastopol, CA, 2010.

[38] Basho: Benchmarking, . URL http://wiki.basho.com/Benchmarking.html. Accessed 9 Feb 2012.

[39] basho_bench, . URL https://github.com/basho/basho_bench. Accessed 9 Feb 2012.

[40] The lepanto opening. URL http://www.diplom.org/~diparch/resources/strategy/articles/lepanto.htm. Accessed 8 Feb 2012.

[41] Build-tool for erlang projects. URL https://github.com/basho/rebar. Accessed 8 Feb 2012.

[42] escript. URL http://www.erlang.org/doc/man/escript.html. Accessed 9 Jan 2012.