University of Warsaw
Faculty of Mathematics, Computer Science and Mechanics

Marek Sapota
Student no: 262969

Distributed hosting platform for Ruby on Rails

Master's thesis
in COMPUTER SCIENCE

Supervisor:
Janina Mincer-Daszkiewicz, Ph. D.
Institute of Computer Science
University of Warsaw

June 10, 2012

Abstract

This thesis presents the design and implementation of a new approach to hosting web applications. The new solution creates a private cloud, capable of hosting web applications, combining the power of manual server configuration, with the comfort of fully managed hosting services. Currently most interesting hosting solutions are presented, and compared to the new approach.

Thesis focuses on the solution's architecture, potential gains from using it instead of the already existing solutions and finally it presents implementation difficulties encountered during creation of this software. The system is mostly implemented in Haskell with a few parts written in Python.

Keywords

web application, hosting, distributed computing, concurrent applications, cloud, functional programming, Haskell, Python

Thesis domain (Socrates-Erasmus subject area codes)

11.3. Computer Science

Subject classification

C. Computer Systems Organisation
C.2 Computer-communication networks
C.2.4 Distributed Systems
Distributed applications

Polish title

Rozproszona platforma serwerowa dla Ruby on Rails

Contents

Chapter 1

Introduction

In recent years web has become a very powerful platform for delivering applications to users. A lot of people are moving away from desktop solutions to web based alternatives because web applications do not require manual installation and updates, often provide integration with other web applications and social networks, and can be easily accessed from any modern device.

As the web technologies develop, web applications become as powerful as their desktop counterparts and many people choose web solutions over the desktop ones. Some people choose to abandon desktop applications altogether and use operating systems that do not support native applications at all, such as Chrome OS [17] or Boot to Gecko [56]. While current attempts to sell laptops powered by Chrome OS were not very successful [76], web applications popularity can not be denied.

A survey made by Berkeley Lab in 2010 showed that 64% of people accessing Gmail, were doing so via the web interface. Also only 15% of people using the web interface did not like the service, compared to 48% dissatisfied among people using Gmail through an email client [4].

These numbers show that web applications play a huge role in software industry and with increasing number of users the problem of delivering these applications efficiently to users becomes more important than ever.

1.1. Current hosting solutions

Developers wanting to have their web application hosted can choose from several solutions. Two most interesting solutions today, that will be explored in more detail, are conventional servers configured by hand and, more modern, fully managed cloud hosting services.

1.1.1. By-hand hosting

There are many different possibilities when it comes to by-hand hosting, developers can choose from shared hosts, virtual private servers, dedicated hosts and many different hosting providers. Common is the fact that the server has to be manually configured to run the desired application. The configuration process is complex, requires knowledge of server administration and when done by an inexperienced person can easily result in sub-optimal performance and compromised security [83].

Unfortunately it is not only the initial set-up that is difficult. Maintenance of a web application in such an environment can be really tedious. When rolling out a new version one needs to manually propagate new sources to all servers, perform maintenance tasks, like migrating the database, and restart web servers on all hosts. While some tools, like Capistrano [7], make this process more bearable, the recipes[1] still have to be written and maintained by-hand.

Because of the manual, static configuration this solution is not flexible and lacks the capability of automatic fail-over in case of host failures. When a server goes down, administrator has to manually migrate its responsibilities to other machines. This is not a big problem when hosting only one application, as application providers can easily prepare backup servers that clone the set-up of production machines. A problem arises when they try to host multiple applications on a machine cluster. Because of the static set-up it is hard to provide expected service quality in case of host failures. For example, when running 20 different applications, requiring 20 instances of each application to be running at all times and a single host is capable of running 5 instances simultaneously, if the application provider wants to be prepared for two host failures, he will need 8 spare servers. When he decides to rise the number of non-fatal host failures to four, this number goes up to 16[2], while in a dynamic set-up, where a supervisor would automatically configure the spare servers to run only the instances that went offline, only four additional machines would be needed.

On-demand scalability is also a problem in this model. To increase the number of running application instances administrators often need to buy and configure new hosts, which is slow, and makes it hard to maintain a perfect set-up, that minimises the number of redundant servers.

Despite these problems by-hand hosting is still a very powerful solution. Because developers are configuring every aspect of the environment[3] the applications are running in, they can use any programming language and match any special demands these applications have. By-hand hosting can be the only possible solution for applications using less popular frameworks and programming languages.

1.1.2. Fully managed cloud hosting

Fully managed cloud hosting, sometimes called *platform as a service* or *PaaS* in short, is a completely different approach than by-hand hosting. In this model the platform provider manages servers and their configuration for the developers, giving programmers only limited control over the environment. Example PaaS providers include Heroku [37] and Engine Yard [13] which offer similar services and are very popular among people choosing to host their applications on a managed cloud platform [14].

Since in this model developers are not responsible for servers maintenance, they do not have to worry about host failures and redundancy. Also if they choose a well respected provider, they can be sure that the application will be run in a professionally tuned environment, and will achieve its best possible performance. On-demand scalability is usually not a problem either, as the most popular PaaS providers allow setting the number of running application instances arbitrarily, at any time, with practically immediate results.

Unfortunately all this comfort comes at a price. Since the environment is controlled by a third party, application developers have to give up most of the power that was available in case of by-hand hosting. When using PaaS programmers are limited to features supported

[1]Capistrano deployment scripts.

[2]When n servers crash, even in the best possible set-up, n instances of a single application might go down, so n backup instances of each application will be needed.

[3]Most importantly developers control the operating system configuration and the list of installed software packages, including their specific versions.

by a chosen provider, and while the list of supported frameworks, databases and additional tools can be quite extensive [39] it is still limited, and finding a provider supporting a more exotic programming language or framework can be challenging, if not impossible.

At first glance a PaaS solution might be far more expensive than by-hand hosting. While the cost of renting a shared or VPS host might be several times lower than the cost of hosting several application instances with a PaaS provider [46, 38], when we count in the cost of man-hours needed to configure and maintain the servers the expenses can become quite comparable [55].

1.2. Project goals

The goal of this project was to create a hosting solution combining advantages of both presented, already existing, solutions. The idea was to create a hosting platform that on one hand would provide easy configuration, scalability and efficient redundancy, known from PaaS, but on the other hand would be much more flexible than fully managed cloud hosting, giving developers much more control over the hosting environment. Additional control would allow application providers to host applications that do not fit into PaaS model, for example because they use rare frameworks or require installation of very specific software packages.

The concept was to build a private cloud solution that works similarly to fully managed hosting services but runs on hardware owned by the application provider instead of a third party [15]. In this paradigm administrators still have to manually set-up the operating system and install software on the hosting machines, but instead of configuring the servers by hand they only need to install a distributed supervisor that will do all the chores needed to set-up web applications for them. Due to its dynamic nature, such a supervisor can adapt to random events like host and network failures without any administrator's involvement making private clouds more reliable and much easier to maintain than manually configured servers.

There are some existing private cloud solutions but they run applications on virtual machines [9, 3] which introduces unnecessary performance overhead and makes them hard to use on low-end hosts or without superuser access. Because of these limitations they can not be used on shared hosts which cost much less than virtual private servers and are probably the cheapest hosting option available [57, 46]. A solution that would configure application servers on the native operating system would introduce less performance overhead and putting less restrictions on the operating system configuration would allow the private cloud to run on many more machine configurations. Combining these two features would make a private cloud solution much more cost-effective.

The hosting solution presented in this thesis, called Cortex, was designed to introduce as little performance overhead as possible over manually configured servers while at the same time making server maintenance much easier. The goal was to create a flexible solution that would be able to facilitate the needs of most web applications, in particular allowing execution of applications written in exotic programming languages and requiring installation of any supporting software. Due to the decreased administration effort and ability to run on low-end machines as an unprivileged user the created platform should also be one of the cheapest hosting options around.

1.3. Document structure

Chapter 2 presents several, currently popular, approaches to web application hosting and demonstrates how Cortex platform combines their strengths to create a new solution.

Cortex architecture and implementation highlights are presented in Chapter 3.

Chapter 4 shows how Cortex executes web applications and how its design allows execution of applications using almost any web framework.

Chapter 5 describes Cortex's data storage module in more detail, presenting used data structures and algorithms.

Chapter 6 contains performance and reliability tests of Cortex and its modules, comparing the results to existing similar software solutions.

Thesis is summarised in Chapter 7.

Appendix A presents the user interface of the Cortex platform.

Appendix B describes contents of the CD attached to this document.

Appendix C shows some situations in which using Cortex instead of other hosting solutions could be beneficial.

Chapter 2

Web application hosting

This chapter presents elements of a typical environment for hosting web applications, focusing on Ruby on Rails and Django frameworks. Section 2.3 shows how fully managed cloud hosting services, like Heroku, make application hosting easier. Section 2.4 presents my idea for a private cloud hosting solution and demonstrates how a private cloud can combine strengths of both PaaS and by-hand hosting to create a solution that is both flexible and easy to manage.

2.1. Hosting architecture

Usual set-up for both Ruby on Rails and Django hosting involves multiple worker processes behind a reverse proxy. This set-up is enforced by multiple factors. Firstly, the *global interpreter lock* also called *GIL*, present in most popular implementations of both Ruby and Python, allows simultaneous execution of only one thread [70, 18]. If administrators want to take advantage of modern multi-core CPUs they are forced to spawn multiple interpreters that will execute the application's code. Secondly, even if used compiler or interpreter does not have GIL, most web servers do not support concurrent request processing and can serve only one client at a time [42]. Finally, even mildly successful applications can overgrow capabilities of a single host and will require setting up servers on multiple machines.

This set-up has some flaws, when it comes to administration and maintenance. Each of the worker processes has to be separately configured, and while most of the configuration will be shared between instances some options, like the port number, have to be different for each worker which introduces a lot of boilerplate that will have to be maintained. Fortunately some web servers, like Thin, recognise this problem and support running a whole cluster of workers with one command [65], however with multiple machines administrators will still have to copy configuration files between them.

Since most application providers want all worker processes to be transparently available to clients through one port and IP address, they have to configure a reverse proxy that will forward requests from that port and address to multiple worker processes. Each change involving a worker process, like changing the port or moving it to another host, has to be followed by a change in proxy configuration, which can be burdensome if there are numerous workers.

Most web applications will require some additional software, like a database, mail transfer agent or a Memcache daemon. They are usually set up on separate machines so web server traffic does not affect the performance of these services, especially the database, which is often the performance bottleneck of a web application [86]. Since these additional services depend on individual needs of the web application, architecture of their set-up is out of scope of this document.

Figure 2.1 shows the architecture of a typical web application hosting environment.

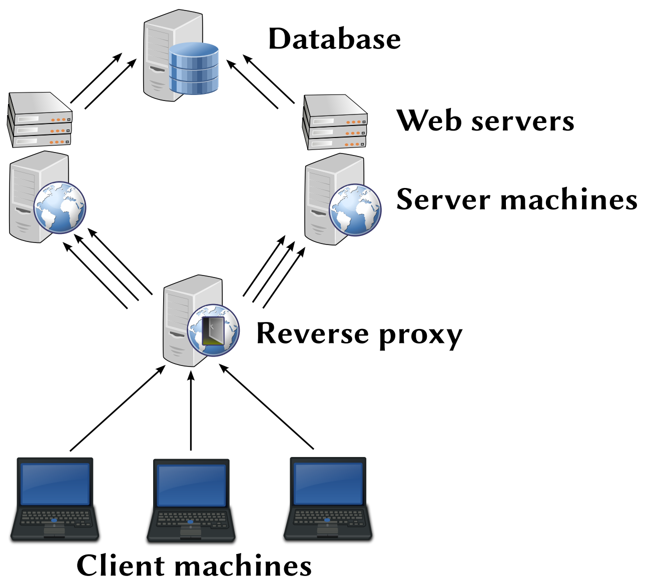


Figure 2.1: Common web application hosting architecture

2.1.1. Hosting static files

Besides serving dynamically generated content, usually HTML documents, most applications also serve static files that do not change between requests. CSS files, JavaScript libraries, and images commonly fall into the static category, and since serving these files can build up to be a significant part of all requests, it is imperative to handle those requests efficiently, which creates some challenges and problems.

Many web frameworks, including Ruby on Rails, do not serve static files, when in production mode [73], and there is a very good reason for that. Most web servers, used for executing Rails applications, like Mongrel, WEBrick or Phusion Passenger are not optimised for serving static files to users. They are many times slower than servers like Apache HTTP Server or Nginx, which are heavily tuned towards static file serving.

To check this claim I benchmarked several Ruby on Rails servers, comparing them to Nginx. This test showed that a dedicated Nginx instance can be thirty times faster than any Rails server. Benchmark was carried out with Apache ab requesting a 5907 bytes document 10000 times. WEBrick, the default Ruby on Rails server, managed to answer only 118 requests per second, Mongrel served 145 RPS, Unicorn 159 RPS and Thin proved to be the fastest with 178 RPS[1]. All these numbers are far behind Nginx which served 4976 requests per second, proving to be far superior, when it comes to serving static content (see Figure 2.2).

This benchmark shows that at least two, separate, web servers are needed to ensure the application will be quick and responsive. This adds complexity to configuration and architecture, and since the set-up has more separate parts it is harder to maintain.

[1]All servers were executing Rails 3.1.3, in production mode, using Ruby 1.9.2.

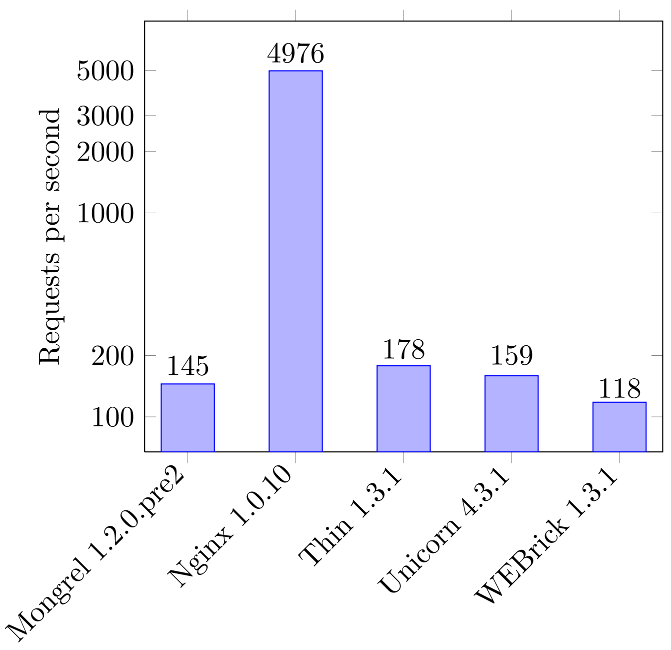


Figure 2.2: Number of static files served per second, using different web servers

2.2. Monitoring server status

To increase reliability administrators can use tools like Monit or God, to monitor status of web servers [54, 16]. If a server goes down, because of a bug in the server code or in the hosted application or because the operation system run out of resources and decided to terminate the server process, it should be restarted. If the hosting environment employs many worker servers, it would be difficult to monitor their status by hand, so it is common practise to use tools that can automate this process [19].

Using tools like Monit can also help developers deal with many bugs commonly found in web applications. Server monitoring tools can not only restart the application when the web server crashes, they can also handle memory leaks and other resource misuse, which,to some extent, can protect application providers against denial of service attacks [87, 47].

While server monitoring tools help to keep the hosting environment healthy, they also introduce quite a lot of configuration overhead. Tools like Monit require administrators to specify a command for starting and stopping each server instance, which might be difficult when also using tools that start a whole server cluster at once, as they usually do not provide access to information about separate spawned servers.

2.3. Fully managed cloud hosting

Heroku and other fully managed cloud hosting providers do all of the above configuration chores behind the scenes, and usually application developers do not even have to know, how the deployment process is really conducted. The price application providers have to pay for this comfort is loss of control over the hosting environment and high services cost, which might be a problem for startups not generating a lot of revenue [38].

Many cloud hosting services, including Heroku and Engine Yard, provide a client side application, that can be run from the command line, which allows users to interact with the hosting environment, for example configure number of worker processes or deploy a new application version [12, 35]. On Heroku deploying a new application requires only two steps, creation of a new environment with "`heroku create`" command, and publishing the application's code with "`git push heroku master`". Deploying a new version is also really easy. When Heroku detects new commits in the Git repository, it will automatically restart all worker processes to execute the new code [25, 36]. Adding any supporting services, like a different database engine or a Memcache daemon is pretty straightforward as well. Heroku provides a simple, click to add, web interface for managing the additional software.

If the application to be deployed fits into Heroku's environment, using this managed cloud hosting service is effortless. Unfortunately even the two above commands show that some applications may have problems with Heroku hosting. Especially existing applications might need some extra work, before they can use Heroku. The command for pushing application code to the cloud assumes that Git is being used for version control, and unfortunately it is the only supported *version control system* (*VCS* for short). If the application's repository is stored in another VCS, Heroku recommends using Git as a deployment mechanism, but it is an awkward solution compared to simplicity of using the native version control system for deployment [36]. Other frequent problem is the fact that the application is deployed on a read-only file system, which forces application developers to store user-uploaded files using a third party storage solution like the Amazon S3 [41, 40]. If the application is serving a lot of static files, application providers should consider hosting them on third party servers as well, as Heroku does not set up additional servers for static file hosting [6], which, as explained in Section 2.1.1, can be very slow.

Heroku's features make it a very good deployment choice for many applications, especially if for people that do not want to be concerned about deployment details and server maintenance problems, although it is not perfect for all tasks.

2.4. Cortex

My idea for a hosting solution differs from the ones described above. I like the idea of cloud hosting because of the easy management that allows developers to focus on the application instead of server administration. Unfortunately currently available cloud hosting solutions are not flexible and are quite expensive, when compared to other hosting options, which makes them unfit for many tasks. These solutions are not appropriate for applications that do not generate substantial revenue, like startups or non-commercial personal websites and blogs. I wanted to create a solution that would be easy to maintain and reliable like PaaS, but also cheap to run and flexible enough to accommodate any programming language or web framework.

Cortex mixes features of by-hand hosting and managed cloud hosting. It is a private cloud solution, where the hardware is operated by the application provider, like in by-hand hosting, but inclusion of an automatic supervisor relieves server administrators of some configuration and maintenance duties, like managed cloud hosting does. Instead of setting up and maintaining each web server instance manually, administrators have to set up only one supervisor process per machine and Cortex will take care of configuring all web servers and the reverse proxy. The platform only manages web servers, it does not help with management of any supporting services. In particular it does not deal with management of the database engine, although it will be needed by almost any application.

The platform does not only configure and start the servers but it also takes care of monitoring their status, without the need of manual Monit set-up. First of all, for each application running on the platform Cortex will maintain configured number of web servers executing that application. If any of these servers would crash, for any reason, Cortex will automatically restart it. Secondly Cortex monitors resources available on all machines belonging to the platform and will try to place web server instances in such a way, that the resources are uniformly consumed. This not only means that application providers can effortlessly use machines with different hardware configurations in one platform, as Cortex will automatically run more servers on more powerful machines, but this property also prevents resource misuse from slowing down all web servers running on the same host as the faulty application instance. When an instance starts misbehaving and overloads a host, Cortex will detect that this machine is being overused and will transfer some instances from the overloaded host to less utilised machines.

Since the platform is running on machines controlled by the application provider, developers can use any additional software, libraries and frameworks they want. There are also no restrictions when it comes to background jobs and scheduled tasks, which are often problematic on fully managed cloud platforms [66]. Unlike most fully managed cloud hosting services Cortex gives applications unrestricted access to the file system, which makes it easy to store user-uploaded files [40]. While Cortex is built to be as flexible as possible, due to its design it places some limitations on hosted applications. Because application instances are automatically managed by Cortex, developers have no direct control over them — web servers can be terminated and restarted at any time, possibly on a different machine. This makes Cortex unfit for hosting some applications, for example ones using long running background tasks as they might be stopped before they have the time to finish.

Similarly to many fully managed cloud services, developers communicate with the platform using a command line client application. The client allows application providers to configure their hosting environment, for example add new applications to the platform, deploy new code version or tune the number of web servers executing each web application.

Figure 2.3 shows the architecture of a web application hosting environment, when using the Cortex platform.

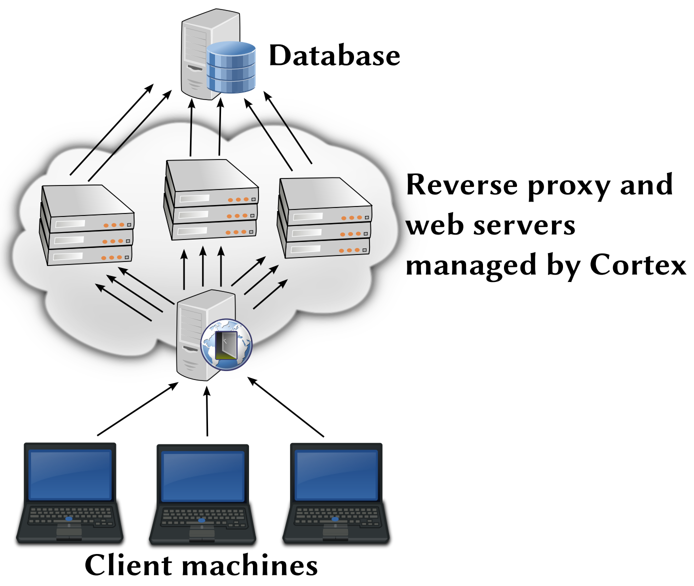


Figure 2.3: Web application hosting architecture, when using Cortex

Chapter 3

Platform overview

This chapter presents the Cortex platform in detail, describing its design concepts, architecture and implementation highlights.

3.1. Definitions

This section provides explanations for several terms used in the rest of this chapter.

- *Application instance* — single web server process serving a web application, for example Thin or WEBrick process for Ruby on Rails applications or uWSGI process for Django.
- *Node* — can refer to either a storage node or an application node.
- *Storage node* — machine running the data storage module.
- *Application node* — machine running the application manager module[1].
- *Client* — depending on context it can mean the client module or an application user connecting to one of the nodes via a web browser.
- *Record* — a key and value pair stored in the key-value data store.
- *Cortex* — web hosting platform presented in this thesis.
- *Miranda* — data storage module, part of Cortex.
- *Saffron* — application manager module, part of Cortex.
- *Ariel* — load balancer module, part of Cortex.
- *G23* — consistency checker module, part of Cortex.
- *Vera* — client module, part of Cortex.

[1]One machine can be both a storage node and an application node.

3.2. Design concepts

Concepts behind Cortex were briefly explained in Section 1.2 and Section 2.4. This section shows the design concepts more comprehensibly, listing all the important ideas behind the Cortex platform.

- Easy application management — application developers should not be burdened with server configuration details. They should be able to focus on developing their web application while the Cortex platform takes care of hosting.
- Flexibility — application developers should not be constrained by the platform, they should be able to use any programming language, framework, library or supporting software.
- Platform scalability — it should be easy to expand an existing platform to new machines and Cortex capabilities should linearly depend on the resources collectively available to all hosts participating in the platform.
- Application scalability — application developers should be able to effortlessly scale their applications, by changing the number of application instances running on the platform, up to the amount of resources available to Cortex.
- No superuser access required — Cortex should be able to run under an unprivileged account. Especially it should be possible to use Cortex in a shared hosting environment.
- Reliability — Cortex should be resistant to host and network failures. In case of a machine malfunction other hosts should seamlessly take over that machine's responsibilities.
- Low running cost — using Cortex should be relatively inexpensive. The platform should introduce only slight performance overhead so it takes as little resources from web servers as possible.
- No application modifications needed — applications using a typical hosting architecture shown in Section 2.1 should not need any modifications before they can be hosted by Cortex.

3.3. Architecture

Cortex has a modular and flexible architecture that allows users to optimise it for the hardware set-up they have available. It is composed of several modules that communicate with each other using a replicating key-value data storage. The communication is done by using active polling, each module periodically accesses relevant data stored in the data storage and compares it to values it received during the previous poll. While this method is slower than push-based communication, where a module immediately propagates data changes to interested parties, it is more resistant to network and host failures. In this approach, if communication fails, for any reason, the module will just try again later, while in the push-based model the message would have been lost.

Whole platform is based on the idea of eventual consistency. Requested actions are not carried out immediately, but given sufficient time platform will converge to the desired state. This choice was made to avoid the need for locking and transactions, which could severely decrease performance in a distributed environment.

Figure 3.1 shows how different platform modules interact with each other.

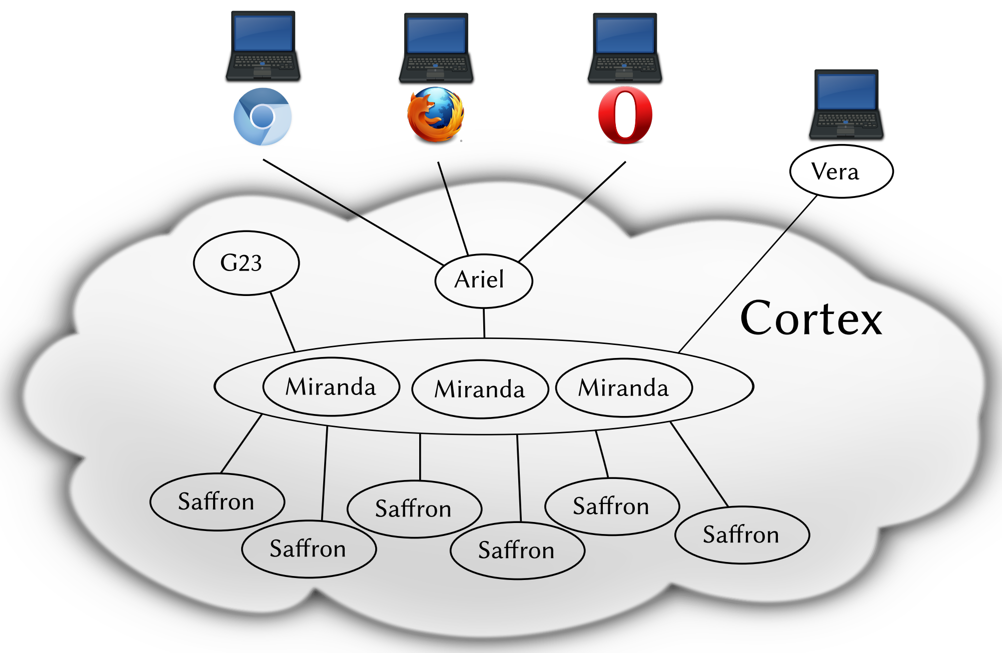


Figure 3.1: Interaction between platform modules and web browser clients

3.3.1. Data storage module — Miranda

Miranda is a replicating key-value store and most important part of the platform, as it serves both as a database and a communication bus between all other modules. It is used to store application sources, their configuration and information about the current state of the platform which includes the list of all currently running application instances. Since all platform modules frequently contact the data storage module it has to handle high number of concurrent connections. Miranda is designed for performance, especially high performance lookups.

As many other key-value stores, Miranda supports a very limited set of commands [2]. Data storage module supports the expected "set", "delete" and "lookup" operations, which respectively store a value identified by a key, remove a key and its corresponding value from the storage, and retrieve a value bound to some key. Apart from this, it also implements several alternative lookup variants, tailored to the needs of other Cortex modules. In particular instead of the value, a lookup operation can return a SHA-1 hash of the value, for fast equality comparison. It is also possible to request multiple records, with keys sharing the same prefix, using only one operation. This functionality is used by other modules as a very limited form of searching through the records.

Each data storage instance periodically performs a synchronisation procedure with another, randomly chosen, instance. During that process, all operations performed on the first instance are pushed to the second one, so after the procedure the second instance has knowledge about all operations performed on both instances. Synchronisation is performed via a gossipping protocol, which spreads data to all replicas through series of simple two-node interactions.

Replication of records to all storage nodes makes Miranda resistant to breakdowns. The storage module is built to be fault tolerant as a cluster, meaning failure of any single node will not disrupt operation of the rest, and after the faulty node starts working again, it will be regenerated with all operations performed during its time offline. However a single node by itself is not resistant to breakdowns and might lose data on shutdown. This is usually not a big problem, due to the cluster fault tolerance properties, but it means that at least two storage modules have to be used, to avoid the risk of losing data.

One Miranda instance can answer requests from multiple modules, but more replicas can be added for greater performance, if the current ones can not handle that many connections efficiently, or to increase platform's resistance to host malfunctions.

A more detailed description of the data storage module, including a precise explanation of the used replication protocol, is presented in Chapter 5.

3.3.2. Application manager module — Saffron

Application manager module is responsible for maintaining requested number of running application instances by spawning new ones and terminating existing instances in case of application node overload. Saffron uses the data storage module to coordinate actions with other managers and maintain number of running instances requested by the administrator. This is the only module that has to run on a machine that is supposed to run application instances and serve requests from clients.

As mentioned in Section 2.4 Saffron is responsible for managing uniform resource consumption on all hosts. Monitored resources include the percentage of free memory and CPU load average figure for the last minute. It is worth noticing that these figures are not absolute values, but only fractions of each host's capabilities, which means they can be reliably compared even if compared hosts have different hardware configurations. This percentage of used up resources later will be called *host load*. Each Saffron instance tries to achieve host load as close as possible to mean host load measured across the platform. When Saffron detects that the host it is running on is using substantially more resources than other hosts[2] it will start terminating application instances until the load drops. Terminated web servers will be restarted by other Saffron instances. To ensure even load distribution application managers only start new application instances if their host is not overloaded[3].

This module has information needed to run applications using different frameworks, and can start an appropriate server for a particular application. It can also perform maintenance before deployment, such as migrating the database schema. Saffron supports multiple languages and frameworks, including Ruby on Rails, Django, Pyramid and Yesod. This module can also be easily extended to support almost any existing framework, and in most cases, it can be done by adding only several lines of code.

Chapter 4 describes how Saffron executes web applications and how its design allows running applications using any programming language or framework.

3.3.3. Load balancer module — Ariel

Load balancer creates a single entry point to all application instances, served by the platform, and forwards client requests to them. It communicates with application managers through the data storage module, collecting information about currently running web servers and generates reverse proxy configuration based on this data.

[2] $h - m > 20\%$, where h is load of the current host and m is the mean host load across the platform.

[3] $h - m < 10\%$, where h and m are defined the same as above.

Since this module's performance is limited by performance of the underlying reverse proxy implementation[4], users maintaining huge clusters might be forced to run several load balancers, so it does not become a bottleneck, capable of serving less requests than the cluster does. In usual set-ups only one load balancer will be sufficient, though.

As Ariel only translates information about currently running instances into a reverse proxy configuration file its implementation is quite simple and will not be described in more detail.

3.3.4. Consistency checker module — G23

Consistency checker is a supporting utility, that cleans up after web servers that, for some reason, did not exit cleanly. It periodically opens TCP connections to the web server instances running on the platform and if a connection can not be established it marks such application instance as offline by removing an appropriate record from Miranda. During normal operation, G23 is not needed, as the application manager marks each web server that exited as offline. If a Saffron instance gets unexpectedly terminated though, its subprocesses, including all web servers managed by it, will be terminated too, leaving orphaned entries about them in data storage. In this instance G23 will remove these records, the same way that application manager would, if it did not crash.

This module only opens TCP connections to web servers in a loop and performs a delete operation on Miranda if a connection failed, so its implementation is pretty straightforward and will not be presented in more detail.

3.3.5. Client module — Vera

Client module is the user interface to the platform, allowing administrators to manage their applications. Differently than the other modules, this one does not run on a server machine, it runs on developer's computer as a standard command line application.

Using Vera administrator can set-up new applications and deploy updates with just several commands. Client has built-in support for multiple version control systems, such as Git and Mercurial. It can export any repository revision and push that code to servers. Additionally it can deploy an arbitrary directory, in case used VCS is not supported, or no version control is being used at all.

Other important functionality is the ability to tune the number of web servers running each application. Administrators can almost instantly adjust, how resources are shared between applications running on the platform. This allows easy prioritisation of certain applications, that are expected to experience increased traffic, without the need of manually reconfiguring the servers.

Functionality available through Vera can be found in Appendix A.

3.4. Implementation

Cortex is mostly implemented in Haskell, with the exception of Vera, which is implemented in Python. Haskell was chosen because of a high quality compiler [28], comprehensive standard library [23], multitude of additional modules available [21] and built-in support for green threads. Haskell is a statically typed and purely functional language. Both properties make code clearer, shorter and easier to maintain. Software written in Haskell also tends to have

[4]Currently provided by Nginx.

less bugs, than programs in imperative languages, mostly because of strong typing and no computation side effects [30, 34]. While Haskell was designed as a research language it has grown to a point, where it is actively used in real-life commercial applications [29].

3.4.1. Functional programming

With the exception of Vera, Cortex modules are implemented in functional programming style. Functional programming encourages use of immutable data structures and deprecates use of mutable variables and global state. This properties have some important consequences when it comes to performance and reliability.

One of functional programming characteristics is the fact that function execution does not have side effects. This means that executing a function can not modify any variables outside of the function scope — the only result of executing a function is the returned value and that value only depends on the function arguments. For the same arguments a function will always return the same result. This feature of functional programming directly reflects in code quality. Since functions can not modify anything outside of their scope it is impossible to accidentally change a variable that is also used somewhere else in the program, which simply eliminates many types of common programming errors. Additionally if a function does not work properly it is easy to debug the problem. As the result depends only on passed arguments it is simple to reproduce the error and see why the result is not what was expected. Lastly it makes really easy to write tests for the code as there is no need to set up global state before test execution and as the only result of function execution is a returned value, checking the that value is enough to ensure a function works properly [1]. These properties of functional programming allow developers to quickly produce bug free, high quality code.

Many functional languages, including Haskell, often use garbage collection instead of manual memory management. Use of pointers and manual memory management is often a source of bugs in imperative languages [75]. Since Haskell uses garbage collection and does not have C-style pointers allowing arbitrary pointer arithmetic, programs written Haskell do not suffer from memory access violation errors, which makes them resistant to yet another class of bugs.

Use of immutable data structures directly impacts performance of functional programs. On one hand such structures are slower than their imperative counterparts as each modification requires allocation of memory and copying the data to the new location. Fortunately modern compilers, such as GHC, can do such operations very effectively, usually only copying a part of the data structure [71]. Additionally the use of lazy evaluation, which is the case with Haskell, allows creation of immutable data structures that are asymptotically as efficient as imperative data structures [59]. Depending on the use case, immutable data structures can even be faster that mutable ones. In a multi-threaded environment if a data structure can be accessed concurrently, developers usually have to resort to some sort of locking so threads do not interfere with each other. A common way of doing so is to allow concurrent access to the data structure for non-destructive operations, but allow only one thread to operate on the structure if the operation will modify it [85]. With mutable data structures this means that a writer will have to wait for all readers to finish before it can start working, as it operates on exactly the same data. With immutable structures though, a writer can immediately start working, as instead of modifying the structure in-place it will create a new copy and will not interfere with readers that will still have access to the old version of the data structure. Test presented in Section 6.1 shows that this property can indeed make a huge difference in performance.

Cortex is almost entirely written in pure functional style with the exclusive use of immutable data structures. Functional programming paradigm does not fit concurrent programs, like Cortex, very well though. To allow access and modification of a data structure from multiple threads it has to be placed in some common location known to all threads. In Cortex data structures that can be modified by multiple threads are stored in *mutable memory locations* called *MVars*, which work similarly to references in languages like Java [26, 61]. These mutable memory locations also introduce a locking mechanism for accessing the stored value. Threads can "take" a value from an MVar which will cause all subsequent actions on that MVar to hang until a value is "put" back. MVars also allow a "read" operation which returns a value stored in an MVar without locking. Cortex uses the "read" operation for actions that will not modify the stored data structure, allowing multiple simultaneous non-destructive operations. While mutable memory locations are "safe" in the way that the stored value is always of the proper type and they can not point to an invalid memory location, MVars do not enforce releasing once taken lock. A thread could lock a data structure and exit without unlocking it, which would deadlock the whole program. To avoid these problems Cortex implements a library that allows atomic data modifications and abstracts the whole locking and unlocking mechanism. Putting these potentially dangerous operations in one place decreases the probability of errors in code dealing with mutable memory.

While technically all IO operations in Haskell, including interaction with the file system, network communication and pointers, are still pure [44], for all intents and purposes they behave like imperative actions which makes them susceptible to the same problems that imperative programs have. Because Cortex clearly separates functional code from the imperative parts it can still take advantage of functional programming benefits, while minimising the number of places venerable to errors introduced by function side effects.

3.4.2. GHC performance and concurrency

Glasgow Haskell Compiler is the leading Haskell implementation that includes an interpreter, native compiler and LLVM bytecode generator. GHC produces highly optimised code that performs comparably to other high-level language implementations [10]. Haskell proved itself, on many occasions, as a very good choice for creating high performance, concurrent networking applications. Benchmarks show that web servers implemented in Haskell can be several times faster than currently industry leading solutions, like Tornado or Node.js [78, 77].

GHC provides a flexible parallel computing interface that mixes green threads and native OS threads [48]. GHC's green threads use many to many threading model, which means multiple OS threads execute several user threads simultaneously. This model, known from Java [60], combines high performance of green threads with the ability to use all the power provided by modern multi-core CPUs. GHC even allows users to specify the number of OS threads, used by the GHC runtime, at execution time, which allows tuning the binary to number of available CPU cores, on different machines, without recompilation [32].

Haskell is a lazily evaluated language. Expression evaluation is deferred until it is needed by other computations, that means values are evaluated and consume memory only when they are actually used [31]. This is a very powerful feature, that can greatly increase application's performance by avoiding needles computations and reducing memory footprint [50]. Unfortunately without careful planing, lazy evaluation might introduce memory bloat. Since evaluation is deferred until the result is needed by another operation, it is possible to write code that will create a stack of delayed computations that will not be evaluated until much later and until then, they will be carried around in memory. Depending on the size of this stack, it might be an unnoticeable problem, but sometimes it might also result in the application running out of memory. Fortunately GHC has good profiling support, that allows easy identification of such problems, and since Haskell allows to explicitly enforce eager evaluation, these problems are usually easy to find and fix [62].

3.4.3. Lazy IO

Haskell's laziness, unfortunately, does not cover IO actions, like reading and writing to a file descriptor. Traditional IO actions are strictly sequenced and a read operation will immediately reserve resources for the data, regardless of when and if it will be used [49]. This combined with the fact that default IO actions operate on strings, and strings in Haskell have terrible performance, as they are implemented as singly linked lists of characters [27, 45] and have to be converted to and from an internal representation on all IO operations [8], can cause excess memory usage and additional work for the garbage collector, especially when dealing with large amounts of data. To avoid these problems Cortex uses lazy IO combined with more efficient lazy byte strings.[5]

Lazy byte strings are an efficient implementation of character vectors that divide the data into smaller chunks, so it can be efficiently processed without loading the entire vector to memory [20]. What is more, unlike standard strings, byte strings can be efficiently read and written without unnecessary conversion, which results in much lower memory usage, and less GC overhead.

Lazy IO is a technique that allows the IO operations to execute asynchronously in the background. It makes the IO operations behave similarly to all other, lazily evaluated, Haskell expressions. Lazy IO actions only load the data that is actually needed and consumed by other computations, and when combined with the lazy byte strings, lazy IO makes a perfect choice for effectively processing large amounts of data, without the need to store all of the data in memory [63].

Unfortunately this approach has problems of its own. When used without proper care lazy IO can cause unexpected problems. Firstly it breaks Haskell's purity. Haskell is a pure language, meaning function execution does not alter the environment outside of the function [74]. With lazy IO though, pure functions can accidentally cause side effects, like triggering an IO action reading a requested chunk of data [64]. This can lead to unintuitive ordering of IO actions [43] and can be problematic when an underlying IO operation encounters an error. Fortunately the errors are silently discarded and simply result in a truncated input/output [24], which can be easily dealt with.

Second issue with lazy IO is much more problematic. When lazy IO is used for reading from a file descriptor, instead of explicitly closing that descriptor after all the read operations, Haskell's runtime will automatically free the descriptor when it is no longer needed [24]. Current GHC's garbage collector implementation automatically closes handles that become unreferenced [33], unfortunately GHC does not trigger a garbage collection when it runs out of file descriptors [72]. If GHC runtime uses up all descriptors, it will not be able to handle new connections until the garbage collection runs. Fortunately this problem was not observed during normal Cortex operation and the issue only manifested itself during extreme loads. On a machine with maximum of $20,000$ file descriptors per process it took 100 threads constantly opening and closing connections to exhaust that file descriptor limit. It is worth noticing that the same program using $1,000$ threads invoked garbage collection more often and did not run out of file descriptors. This shows that very specific circumstances are needed

[5]In case of the data storage module, when implemented using standard strings and strict IO, attempt to set a value of approximately 110MB would result in using about 3GB of memory and more than 90% of time spent in garbage collection. Using lazy IO and lazy byte strings brought this values down to total memory use of 130MB and no time spent in garbage collection.

for the problem to arise and a real life application, doing many memory allocations, which in turn require more garbage collections than the test program did, should not be often affected.

In spite of these problems lazy IO is still a very efficient option, and since these problems are not fatal, Cortex uses lazy IO for all communication.

Chapter 4

Application manager

This chapter explains how Saffron executes web applications and shows how its design allows the use of any programming language or framework.

4.1. Running web applications

This section describes what steps are usually needed to execute a web application. To provide a concrete example I will show how one would go about starting a Ruby on Rails application, but this process is very similar for applications using Django or other frameworks.

First of all, most applications will need some setting up before the first run. For example most applications will want to create the database schema and sometimes populate the database with starting values. For Rails that would be "`rake db:migrate && rake db:seed`". Since this has to be done only once, Cortex does not involve itself in this process, application providers are expected to do this step on their own.

Secondly before actually executing an application administrators have to install all dependencies, like libraries used by the application, and migrate the database schema. This step has to be done only once after each application upgrade and is not needed when starting an additional instance of an already running application. Ruby on Rails uses Bundler to manage dependencies. This utility interacts with Ruby's package manager[1] and allows easy installation of all required libraries with a single command — "`bundle install`". Bundler only manages Ruby libraries and will not install other things, like C libraries that the Ruby packages depend on for example. To install these one would have to use a package manager specific to the used operating system. For upgrading the database schema Rails provides an already mentioned "`rake db:migrate`" command. It is worth noticing that both presented commands are safe to be run multiple times. If the database schema is already at the newest version or all needed libraries are already installed, they will simply exit without performing any action.

Finally after the preparations one would actually start a web server executing the application. For most frameworks, there are many ways of doing that. The simplest way to start a Ruby on Rails server is to use "`rails server`" command, but one can also use "`passenger start`" to start a standalone Phusion Passenger server or "`thin start`" to use a Thin server.

[1] RubyGems

4.2. Application hosting with Saffron

The above process can be clearly divided into two parts: preparing an application to be executed and starting a web server. As previously mentioned the first part only has to be executed once after each application update, but Saffron assumes that it will be safe to rerun it multiple times, even when it is not needed. When Saffron detects a new application was added to the platform or someone published an update to an already hosted application it stops all servers executing the old version, downloads new source code, places it in a temporary directory and performs the steps needed to prepare the application for execution. These steps depend on the type of framework used in the application and can do arbitrary things needed by that particular framework. As each Saffron instance participating in the platform will execute above actions without any coordination with other instances it is easy to see that the preparation step will be executed multiple times[2], but since it is assumed that it is safe to run preparation actions multiple times it is not a problem. After the application is prepared, Saffron will begin executing application instances. Again the actual command used to start the servers depends on application type.

Saffron is very flexible when it comes to what the preparations can actually do. Technically they are Haskell functions that take a single argument — file system path to application's source code. Apart from the already mentioned fact that preparations should be safe to run multiple times, there are completely no restrictions for what those functions can perform. For currently implemented frameworks these functions are quite simple and only spawn subprocesses that execute required command line actions, like "`rake db:migrate`" for Rails applications.

Actions needed to start a new application instance are also implemented as an arbitrary Haskell function. This function takes the port number on which the server should listen, path to application's source code and returns a two-way communication channel that can be used by the application manager to request server termination and by the server to notify Saffron that for some reason the server has exited. This simple interface does not actually specify how the servers are supposed to be executed which introduces the possibility for many different implementations. Fortunately most of the time web servers will be executed with a single command, like previously mentioned "`rails server`". To make implementing handlers for these common cases easier Saffron provides a helper function that takes a command and returns a handler, implementing the interface specified above, that when called will execute that command in the directory containing application's source code. All server execution handlers for currently implemented frameworks use this helper.

Because Saffron does not strictly define what application preparation or server execution is, it can be used with any framework for which the application running process can be split into two parts described above. This includes most currently popular web application frameworks including Ruby on Rails and Django. Additionally, because of the included helper functions, support for most frameworks can be added to Saffron by writing only several lines of code.

[2]Once for each Saffron instance participating in the platform.

Chapter 5

Data storage

This chapter presents Cortex's data storage module in detail, including design concepts, used data structures and the replication protocol.

5.1. Design concepts

Miranda is used by Cortex modules as a database, but also it acts as a communication bus between these modules. The need to perform both tasks efficiently led to some important architectural choices:

- Cortex does not need a complicated, full-blown *relational database management system*, also shortened *RDBMS*. Due to the complicated design RDBMS tend to be slow, and as a communication bus Cortex's database has to handle a lot of concurrent requests. Instead of using an already available, off-the-shelf database solution, Miranda was designed and implemented from the ground up to be simple and efficient. Data storage module was implemented as a key-value store which has less features than a RDBMS, but because of its simplicity it has the potential to be much faster.

- Cortex modules require at least a limited form of searching through stored records, for example to list all configured applications or currently running application instances. To fit with the simple key-value store model, Miranda implements efficient partial key lookups, allowing clients to request all records sharing the same key prefix in one operation.

- Miranda is used to store large values, such as application code repositories, which can easily exceed several megabytes and are not feasible to be stored in memory, but on the other hand, the amount of stored records is quite low — only several records per application instance and a few for each application's configuration. Because of this all keys can be stored in memory, for faster access, while the values are stored on disk. Since on many occasions, Cortex modules are not interested in actual value, but only if the value has changed since previous check, SHA-1 hashes of values are stored in memory as well, improving access speed and decreasing the amount of transmitted data.

- Cortex is designed to be resistant to host and network failures, and as Miranda, being the communication bus for all modules, is a central part of the platform, it has to be resistant to failures as well. Data storage module replicates any action performed on one instance to all other instances participating in the platform, so even if an instance goes down no data is lost and Cortex modules that were using this instance can seamlessly switch to using any other Miranda instance.

5.2. Data structures

To store records Miranda uses an associative array, specifically designed to allow fast lookups. As mentioned before Miranda allows access to multiple records, sharing the same key prefix, in a single operation. The prefix can not be arbitrary, instead keys in Miranda use a special character sequence — "`::`" — to split the key into *tokens*[1]. The prefix can only contain full tokens. For example searching for all keys beginning with "`foo::bar`" will match both "`foo::bar::baz`" and "`foo::bar::qux`", but not "`foo::barbaz`", because "`barbaz`" is treated as a single, indivisible token that does not match "`bar`".

To allow efficient retrieval of records Miranda uses a prefix tree, sometimes called a trie, operating on tokens instead of characters like a standard prefix tree does [5]. Commonly prefix tree implementations use a static array of pointers, one cell for each possible character, connecting nodes with their children [82]. Since the number of possible tokens is not limited, instead of a static array Miranda uses Haskell's built-in associative array to store tree's structure. An associative array is slower than a static array, but does not place any limits on the number of possible child nodes. Figure 5.1 shows an example tree storing two records.

To allow replication, Miranda not only stores the records, but also keeps history of all "set" and "delete" operations. In following sections term *commit* will refer to a single "set" or "delete" operation performed on a Miranda instance.

History is kept as an ordered list of commits. Each commit is timestamped and the list is sorted chronologically. Commits have a unique id that not only identifies the performed operation, but also tracks the history of commits that happened before this one. Even if two commits are identical, having the same timestamp and describing the same operation, but the history before them differs, they will have different ids. This property makes it very easy to compare whole history lists for equality, as it is enough to check if latest commits have the same id. This idea is borrowed from Git VCS, where the revision id depends on ids of all parent revisions [11]. Miranda uses exactly the same concept, apart from the fact that the history is linear and commits can have only one parent. Technically the id is a SHA-1 hash of a commit's timestamp, key changed by the operation, parent's id and for "set" commits SHA-1 hash of the appointed value. Figure 5.2 shows how commit ids are generated. Additionally to the list Miranda also redundantly stores all commit ids in a set, which allows fast checks for existence of a certain commit in the list. This property is used during replication, which is explained in Section 5.3.

During replication, also later called *synchronisation*, commits present in one Miranda instance but not the other, are transmitted and inserted to the history list. Section 5.3 explains how two history lists are actually compared, and how required commits are transmitted between Miranda instances. This insertion operation is called *rebasing* because of the similarities with Git's rebase command [79]. Rebasing a commit is likely to change other commits in the list, as all commits that happened after the one being inserted will change their ids. This means rebasing is a potentially expensive operation that will have to recalculate ids for multiple commits. Operations that are bulk inserting many commits into the list, should carefully order the commits to minimise complexity. For example appending n commits to the list will take $O(n)$ time if they are appended in chronological order, but inserting these

[1]For example for "`foo::bar::baz`" the tokens would be "`foo`", "`bar`" and "`baz`".

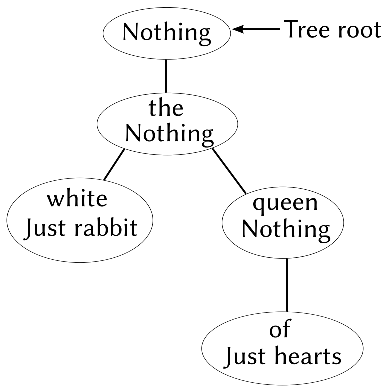
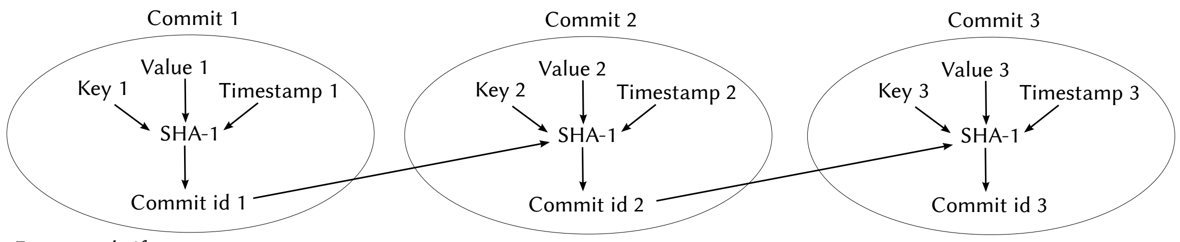


Figure 5.1: Prefix tree after inserting two records — “`the::white`” with value “`rabbit`” and “`the::queen::of`” with value “`hearts`”. Values stored in nodes use Haskell notation, where “`Nothing`” represents lack of value and “`Just X`” represents presence of value “`X`”.



For example if:
Key 1 = "the::white"
Value 1 = "rabbit"
Timestamp 1 = "2012.02.03 12:22:45

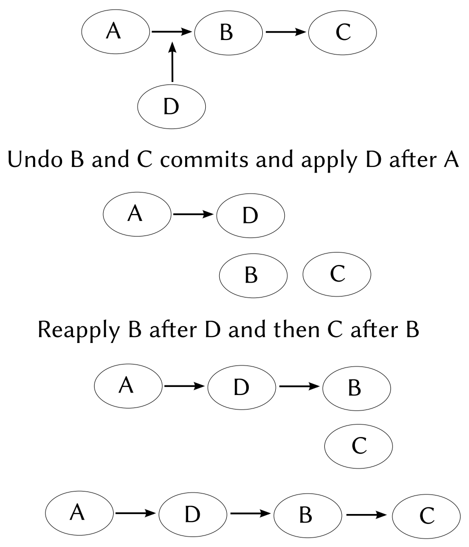
Key 2 = "the::queen::of"
Value 2 = "hearts"
Timestamp 2 = "2012.02.03 12:25:11

Then:
Commit id 1 = SHA-1(' the::white Set "rabbit" 2012.02.03 12:22:45') = "61503a8afa60204c095a0f657100c6b188252747"
Commit id 2 = SHA-1('61503a8afa60204c095a0f657100c6b188252747 the::queen::of Set "hearts" 2012.02.03 12:25:11')
= "ce142f476d8b0e114cb88b46fbff894620026884"

Figure 5.2: Generating commit ids dependant on whole commit history

commits in reverse chronological order will take $O(n^2)$ time, as adding a new commit will change ids of all already inserted ones. Figure 5.3 shows how rebasing a commit works.

To prevent the history list from growing indefinitely, “useless” commits are periodically removed. A commit is considered useless if another commit altering the same key exists later in the history list, which means that currently stored records would not change if the operation described by the useless commit did not happen at all. An operation removing all useless commits is called *squashing*. Commit squashes are not globally coordinated between Miranda instances and any instance can randomly perform a squash, which leads to some problems with instance synchronisation. When one instance that performed a squash would blindly synchronise with another one that did not, all of the squashed commits would be transferred back from the second instance to the first one, making the squash pointless. To counter that I designed a protocol that prevents synchronisation if it would undo all the work of a squash operation. Each instance stores the time of the last squash operation it knows about in form of a timestamp. When an instance decides to squash commits it resets its timestamp to current time. Before synchronisation instances compare their timestamps, if they are equal they go along with the synchronisation, but if they differ, instead of synchronising instance with the older timestamp performs a squash and sets its timestamp to the one it got from the other instance. This way squash operations can spread through all Miranda instances and removed commits will not get transferred back from other nodes.


A
B
C
D
Undo B and C commits and apply D after A
A
D
B
C
Reapply B after D and then C after B
A
D
B
C
A
D
B
C
Figure 5.3: Rebasing commits in Miranda

5.3. Replication

All Miranda instances used in one platform store the same records and replicate all "set" and "delete" operations performed on one instance to all other instances. Replication uses a gossipping protocol, designed by me, that spreads information about performed operations by series of simple two-node synchronisations. Miranda uses *one-way*, *push synchronisations*, meaning only data from the instance initiating the connection is transferred to the other instance. This protocol does not guarantee immediate consistency between the replicas and only gives assurances about eventual consistency of instances.

Replication protocol is designed to be as fast as possible when there is not much data to send. This choice was made because with frequent synchronisations the amount of new commits that will have to be transmitted is expected to be low. Instances first find the latest common commit[2] and then the initiating node transfers all commits that happened after the common one to the other instance. During the first phase, when nodes try to find the latest common commit, the initiating instance sends queries about commit ids, and the other replica responds if it has that commit in its history. Queries are performed linearly from the latest commit to the oldest [84], which can be very fast as the common commit is expected to be near the list beginning. In the extreme case when commit histories of both instances are equal, linear search will find the common commit on the first try. Generally this approach will be faster than a binary search when $k < log_2(n)$, where k is the number of commits that will have to be transmitted and n is the length of commit history.

Each transmitted commit will have to be inserted into the history list of the receiving instance which, as explained in Section 5.2, involves rebasing all chronologically later commits. To avoid unnecessary work commits are transmitted in oldest to newest order, so they can be applied chronologically, which will require less rebasing operations than any other order.

Commits are sent as a serialised list of Haskell objects. Serialisation is performed using a high performance library that allows marshaling any object to a binary string [22]. This library uses lazy byte strings to lazily serialise values on demand, which allows Miranda to transmit the commit list without the need to load them all to memory.

[2]Newest commit that is present in both instance's histories. Commits are considered equal if the have the same id.

Chapter 6

Tests

This chapter presents results of several tests performed on the whole Cortex platform, as well as on its separate components. Section 6.1 describes a performance test of Miranda, concentrating on the ability to handle high number of concurrent connections. Since in a Cortex deployment spanning many machines, Saffron instances will contact the data storage module many times a second, it is important that Miranda can efficiently handle numerous simultaneous connections. The second test presented in Section 6.2 checks how host failures affect application users. Cortex was designed with reliability in mind and one of arguments for using private platform solutions is the fact that they can adapt to host failures without administrator's involvement. This test examines if the self-repair feature of Cortex is good enough to provide uninterrupted service to application users.

6.1. Data storage performance

In this section Miranda's performance is compared to some popular database solutions. Tested software includes relational databases — SQLite [81] and PostgreSQL [69] — as well as MongoDB [52], which is an established key-value store.

6.1.1. Testing environment

Databases were tested for speed of "set", "lookup" and "delete" operations as well as their ability to handle many concurrent connections. Since Cortex modules do not keep open connections to the database and spawn a new connection every time one is needed, during the testing each operation was performed on a new database connection. In particular this means support for transactions present in some of tested databases was not used and data was flushed to disk after each operation[1].

Test consisted of n turns performed concurrently by a worker queue of k threads. During i-th turn, the testing application performed the following actions:

1. Assign value "i" to key "test::i".
2. Lookup value corresponding to key "test::i".
3. Remove key "test::i" and its value from the database.

If any of the above actions had failed, it was repeated until it succeeded.

[1]In case of MongoDB data was only queued to be written as this database uses a lazy write strategy [53].

With the exception of SQLite n was set to $10,000$. SQLite was so slow, that n had to be lowered down to 100, if the test was to finish in sensible time. Each database was tested for three values of k — 10, 100 and $1,000$[2]. Testing application was run five times for each of the database/k combinations to average out any performance hits caused by other background applications running on the system.

All database solutions were tested on a two core machine, with the testing application running on the same host as the database, which means they had to share CPU time and memory, but on the other hand the cost of sending data through network is negligible in such a set-up. Testing application was written in Haskell, using green threads, and while different bindings to each database required a slightly different testing application for each of the databases, the architecture stayed the same allowing fair comparison.

6.1.2. Results

Table 6.1 shows how long it took Miranda to complete the tests. Table 6.2 and Table 6.3 show the time figures for MongoDB and PostgreSQL respectively. Figure 6.1, comparing all the results, shows Miranda was more than one and a half times faster than MongoDB and more than three times faster than PostgreSQL for 10 concurrent connections. For larger number of simultaneous requests the differences are smaller, but Miranda is still much faster than competitors.

As shown in Table 6.4 SQLite did not do well in this test. With only 8.47 requests per second it proved to be the slowest of all tested solutions. This is mostly because SQLite does not support partial database locking and while multiple read operations can run simultaneously, any operation altering the database will lock the entire file for exclusive use [80].

PostgreSQL was run with default configuration options, in particular default limits for the number of allowed concurrent connections were used. The default value was so low that PostgreSQL experienced only minor performance changes in response to different k values. Unfortunately due to PostgreSQL implementation it was not possible to raise the number of allowed concurrent connections without tweaking the operating system kernel [67, 68]. As Cortex is supposed to be fully functional on machines application providers do not have root access to, it was only fair to leave kernel configuration at its default values.

Results for MongoDB can be surprising as, in contrast to Miranda, it does not write values to disk after each operation, instead flushing them on as-needed basis [53], which should have given it an advantage. Miranda might be faster because of the way it handles concurrent read and write requests. Similarly to SQLite, MongoDB uses a coarse global lock for write operations — any number of clients can simultaneously read values, but there can be only one writer and while a write operation is being executed, no readers are allowed [51]. Miranda handles readers and writers similarly to MongoDB, but because it is written in a functional language, using immutable structures, in practise it works very differently. As explained in Section 3.4.1, each reader gets its own copy of Miranda's data structures, and while writers lock the access to those structures, they only do so for new connections. Simply put, everyone has to wait for writers, but writers do not have to wait for readers to finish, instead a writer immediately gets access to the data, while all currently present readers can still operate on copied old data structures. This gives Miranda the advantage of, at least partially, allowing simultaneous read and write operations.

It is worth noticing that for $k = 100$ Miranda's times are very unstable. Slower times are caused by GHC runtime not running garbage collection often enough to free up file

[2]Since setting $k > n$ does not make sense $k = 1,000$ was dropped for SQLite.

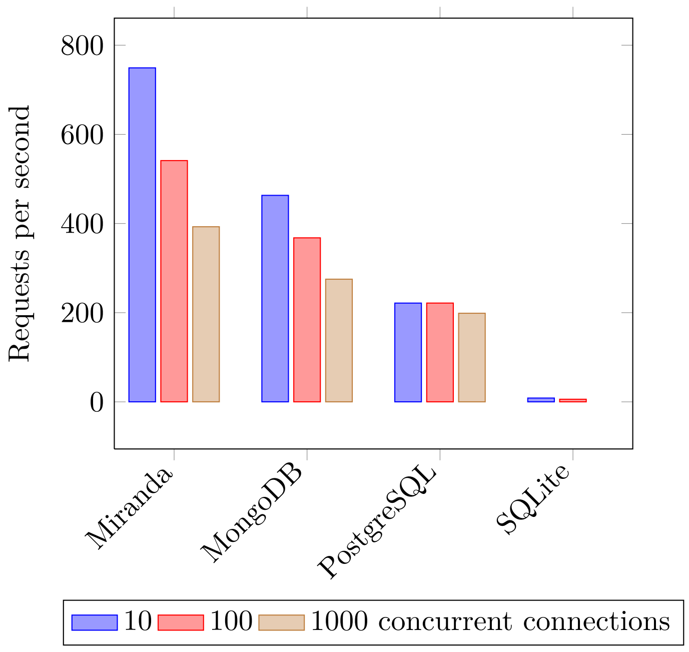


Figure 6.1: Database performance comparison

descriptors. As explained in Section 3.4.3, this caused Miranda to drop some connection attempts.

This test shows that Miranda performs very well in a highly concurrent environment, beating well established database solutions. The ability to efficiently handle connections in concurrent environments makes it perfect for the Cortex platform.

Table 6.1: Time needed by Miranda to process 30, 000 operations for different number of simultaneous connections

concurrent connections	10	100	1, 000
	39.709s	71.290s	77.610s
	39.524s	65.830s	77.430s
	39.413s	53.877s	74.840s
	40.700s	43.040s	77.240s
	40.878s	43.158s	74.800s
average times	40.045s	55.439s	76.384s
standard deviation	0.690s	12.906s	1.434s
requests per second	749.16 RPS	541.14 RPS	392.75 RPS

Table 6.2: Time needed by MongoDB to process 30, 000 operations for different number of simultaneous connections

concurrent connections	10	100	1, 000
	60.89s	68.89s	101.27s
	61.65s	79.56s	107.93s
	61.64s	82.97s	109.12s
	61.62s	82.68s	119.97s
	78.07s	93.66s	106.97s
average times	64.77s	81.55s	109.05s
standard deviation	7.44s	8.86s	6.81s
requests per second	463.15 RPS	367.86 RPS	275.10 RPS

Table 6.3: Time needed by PostgreSQL to process 30, 000 operations for different number of simultaneous connections

concurrent connections	10	100	1, 000
	135.03s	135.25s	153.28s
	136.89s	136.34s	150.01s
	135.58s	134.99s	149.47s
	134.78s	135.23s	152.05s
	135.04s	135.34s	150.15s
average times	135.46s	135.43s	150.99s
standard deviation	0.85s	0.52s	1.61s
requests per second	221.47 RPS	221.52 RPS	198.69 RPS

Table 6.4: Time needed by SQLite to process 300 operations for different number of simultaneous connections

concurrent connections	10	100
	36.181s	61.010s
	34.289s	56.493s
	34.570s	42.013s
	35.930s	40.143s
	36.199s	59.775s
average times	35.434s	51.887s
standard deviation	0.93s	10.026s
requests per second	8.47 RPS	5.78 RPS

6.2. Cortex reliability

Cortex was designed to be resistant against host failures. In this context a host failure is any error that makes a host inaccessible for the load balancer module or the consistency checker module. The problem may be an operating system crash, hardware failure or a denial of service attack. When a host belonging to the platform becomes inaccessible other hosts should automatically take over its responsibilities. In particular any crashed application instances should be restarted on other hosts, so applications can deliver expected performance all the time. This test checks how host failures affect application availability and performance by simulating crashes of Saffron instances.

Test was carried out using 4 Saffron instances running 12 application instances. During the first part of the test Apache ab performed $10,000$ requests, accessing an HTML file, with different levels of concurrency. Concurrency levels of 1, 4, 12 were chosen, which reliably simulate expected load levels of an application that would use 12 instances. In the second phase Apache ab was performing the same task, but Saffron instances were randomly shut down and restarted[3] 10 times during the course of the test. Results of this test were very good, not only no requests resulted in errors, but also the number of requests per second, the platform was able to serve, did not deteriorate.

The first feat could have been achieved because of Ariel implementation which internally uses Nginx to provide a reverse proxy. Nginx uses a simple round-robin method to choose an application instance that will serve a request, but when the chosen server fails to respond instead of reporting an error to the user it tries another server on the list until it succeeds or reaches a preset timeout. This way until at least one application instance is still running, all errors will be handled behind the scenes and users will not notice them [58]. Additionally as soon as G23 detects an application instance is down, it will be removed from Nginx configuration permanently and another application instance will be spawned by some other running Saffron instance. When using default configuration of G23 and Saffron modules maximum time between an instance failure and spawning another one is 2 seconds. This time can be easily decreased by tweaking some configuration options, but in this test default values were used as they proved to be good enough.

In the first part of the test Apache ab averaged 315 requests per second, 95% of requests took less than 63ms and 98% took less than 80ms[4]. Figures achieved in the second part were very similar, striking within 5% of the above results, which is within margin of measurement error, as such differences were also present on different runs of the first part of the test. This slight variance can simply be caused by network delays and host slowdowns that are an effect of background tasks consuming resources. Lack of performance decline can again be attributed to Nginx reverse proxy module, which does a great job of balancing the load thought available servers, and G23 working together with Saffron to respawn terminated application instances on other hosts. Quite importantly, as explained in Section 3.3.2, the hosts on which new instances are spawned, are not chosen randomly but instead Saffron prefers hosts with most resources to spare. This ensures uniform layout of instances on available hosts, which produces good performance results.

[3]Shutting down a Saffron instance also terminates all application instances that were spawned by this application manager.

[4]This figures are for 12 concurrent requests.

6.3. Tests summary

Cortex performed exceptionally well on both tests. Results of the first test show that Miranda's implementation should fit very well into the distributed Cortex environment, where many modules connect to the data storage every second. The second test proves that Cortex is a dependable hosting solution which can maintain application performance and availability in spite of host failures.

Chapter 7

Summary

The goal of this project was to create a hosting platform combining strengths of fully managed hosting services and by-hand hosting. The resulting product, Cortex, fulfils this goal by implementing a private cloud hosting solution that is flexible, easy to use, ensures reliable web application performance and is cheap to run.

While the ultimate goal of the platform was hosting Ruby on Rails applications, the flexible design makes it suitable for other popular web frameworks as well. Most currently available frameworks could be used with Cortex without much extra work. As a proof of concept Cortex was successfully used to host applications in Ruby, Python and Haskell, using Ruby on Rails, Django, Pyramid and Yesod frameworks. Quite importantly, because Cortex mimics a common application set-up procedure, most applications will not need any modifications before they can be hosted on the platform.

Created platform provides an easy to use interface for application developers. The interface is similar to the one found in popular fully managed cloud hosting solutions, such as Heroku, and allows performing most administrative actions, like updating a hosted application to a new version or changing the number of running web servers, with just one command. For some tasks the created interface even surpasses the functionality of well established solutions, for example by not placing any restrictions on used version control system[1].

One of Cortex's design concepts was resistance to host and network failures. The platform proved to be a reliable solution that was able to ensure application availability and high performance despite any unforeseen machine problems. Because of its dynamic nature, Cortex can provide that reliability in a very efficient way. Since Cortex actively monitors the state of web servers running on the platform it does not have to use multiple statically set-up machines providing redundancy for each of the hosted applications. Instead it can use only a few redundant hosts that will be dynamically configured to take over the responsibilities of machines that went down. This dynamic redundancy property makes Cortex a cost-effective solution, requiring less spare machines than a statically configured hosting environment that would provide the same level of failure resistance.

Cortex implementation does not require superuser access to the operating system and all modules can run as an unprivileged user. This feature makes it possible to use the platform on almost any host, allowing developers to choose any hosting provider they want. Most importantly developers on a budget can use cheap shared hosting and still get an easy to use and reliable hosting platform.

Implementation choices, that were made during the course of the project, proved to be correct. Use of Haskell and functional programming techniques allowed Cortex to be reliable

[1]Heroku only supports Git.

and fast, while the modularity of the platform made it very easy to set-up on any hardware configuration. The distributed, modular architecture also made Cortex a scalable solution that is easy to expand to new machines. Because a simple key-value store is used for communication between modules, future development of any additional software interacting with the platform would be extremely easy.

All these properties come together and create an interesting product that in some situations might be a better choice than currently popular hosting solutions. In particular people running a few applications and currently configuring their servers manually might benefit from Cortex's reliability and easy maintenance features. Also people using fully managed hosting services might benefit from added flexibility and lower running costs of a private cloud solution.

Future improvements to Cortex might include optionally running web applications in a secure sandbox, so the platform could be used to host applications from untrusted sources, and an authentication and authorisation mechanism that would allow many people to share one platform instance, without getting in each others way.

Cortex is released as free software, which makes it available for anyone that might profit from a private cloud hosting solution, but even people which are not searching for a hosting platform might benefit from Cortex. Because of the modular architecture different parts of the platform can be used separately. Especially Miranda is not strictly tied to other Cortex modules and can be used as a general purpose key-value store in other software. Because of its reliability properties, high performance and easy interface I intend to use Miranda in my future projects.

Appendix A

Vera commands

This appendix shows how Vera can be used to set-up and manage applications hosted on the Cortex platform. The client requires a configuration file — "`.veraconfig`" — to be present in the root directory of the application's source. Following examples will assume that this configuration file is being used:

```
[app]
; This is the only required value.  Other values are specified for
; convenience, but they can also be set as command line options.
name = TestApp

[miranda]
host = my.miranda.host.address
port = 8205 ; Default Miranda port number.
```

A.1. Setting up a new application

Following commands set up a new Ruby on Rails application on Cortex, requesting 5 application instances. All of these instances will be available through Ariel on port 8000. In this example Git VCS is used, but Vera also supports Mercurial, GNU Bazaar and directly exporting application's directory tree.

```
$ vera --new-app --app-type rails
$ vera --push --repo-type git
$ vera --ariel-port 8000
$ vera --instances 5
```

A.2. Updating application to a newer version

After the initial set-up only one command is needed to deploy a new application version. This will transfer new sources to Cortex, stop web servers executing the old application version and start new ones to execute the updated sources.

```
$ vera --push
```

A.3. Modifying the number of application instances

If the application is experiencing more load that currently running web servers can handle it is easy to ask the platform to execute more application instances. The same command can also decrease the number of running web servers and free resources that then may be used by other applications.

```
$ vera --instances 7
```

A.4. Monitoring platform status

Vera can also provide information about current application status. In this example previously set-up Ruby on Rails application is being executed by 4 web servers, which is less than requested 7 application instances.

```
$ vera --info
TestApp settings

Application type: rails
Port: 8000
Instances: 7
Running: 4
```

A.5. Adding a new Miranda instance

Apart from managing applications Vera can also be used to add a new data storage instance to the Miranda cluster. After starting up a brand new Miranda instance this command will connect it with already running instances and will start the replication process between them.

```
$ vera --add-node my.new.miranda.host.address:8205
```

Appendix B

Attached CD

This appendix describes the contents of the CD attached to this thesis.

- “`thesis.pdf`” — an electronic version of this thesis.
- “`thesis/`” — LaTeX sources for this document.
- “`Cortex/`” — sources of the Cortex platform.
- “`tests/`” — scripts used for the tests presented in Chapter 6.

Appendix C

Example use cases

This appendix presents two concrete examples of situations in which someone might want to use Cortex.

C.1. University

A university wants to host a university website, recruitment service and an application through which students can sign up for classes. To do that they set up two Miranda instances on two hosts, G23 and Ariel on a third one and install Saffron on multiple machines that will be used to execute above applications.

With this set-up the university can effortlessly publish new application versions with just one command and in case of a machine failure, all hosted services will still be available, as other hosts will automatically reconfigure to run the web servers that were terminated. Most importantly during the recruitment time, when the recruitment service will have to handle a higher load, the university can easily adjust the number of instances of each of the applications to give that service more resources. After the recruitment time is over, the university can tune down the number of recruitment service instances and use freed resources to scale up the classes sign-up application, which will be heavily used by all the newly recruited students.

In this example using Cortex allows the university to change priorities of all hosted applications without any manual changes to servers configuration.

C.2. Students on a budget

A group of students creates a startup company and develops several web applications that, for now, are not very popular. They can rent two VPS hosts for two Miranda instances and a third one for G23 and Ariel. These hosts can also be used to set up a database engine, Memcache daemon and any other software the applications require. For running the applications students rent several shared hosts and install Saffron on them.

By using shared hosting students can minimise the cost and because they do not have to manually configure the servers, they can focus on developing profitable applications. If any of the applications becomes a success, students can quickly scale it up, so it can handle the increased load. This would probably require renting more servers, but to connect the new machines to the existing platform, students would only have to install and configure a single software package — Saffron.

For a startup company using Cortex might be beneficial because of the low running cost, easy maintenance and flexibility that allows simple expansion of the platform.

Bibliography

[1] Slava Akhmechet. Functional Programming For The Rest of Us, 2006. http://www.defmacro.org/ramblings/fp.html.

[2] Apache. CassandraCli. http://wiki.apache.org/cassandra/CassandraCli.

[3] AppScale. Deploying AppScale. http://code.google.com/p/appscale/wiki/Deploy_AppScale.

[4] Berkeley Lab. Google Apps Survey, 2010. https://commons.lbl.gov/display/google/Google+Apps+Survey+Results.

[5] Boston University. Trie. http://www.cs.bu.edu/teaching/c/tree/trie/.

[6] brad. Heroku Cedar — Static Assets — Rails 3.0.x, 2012. http://stackoverflow.com/q/8781039/506367.

[7] Capistrano. Capistrano homepage. https://github.com/capistrano/capistrano/wiki.

[8] Carl and ondra. Poor haskell network performance, 2010. http://stackoverflow.com/a/4228721/506367.

[9] Cloud Foundry. VMware's Cloud Application Platform. https://github.com/cloudfoundry/vcap.

[10] Debian. GHC and Scala performance comparison. http://shootout.alioth.debian.org/u64q/benchmark.php?test=all&lang=ghc&lang2=scala.

[11] Paŭlo Ebermann and Dustin Getz. git rebase — unexpected mismatch of my hashes, 2011. http://stackoverflow.com/a/5145714/506367.

[12] Engine Yard. Deploy from the CLI (Engine Yard CLI User Guide). https://support.cloud.engineyard.com/entries/21009927-deploy-from-the-cli-engine-yard-cli-user-guide.

[13] Engine Yard. Engine Yard homepage. http://www.engineyard.com.

[14] Engine Yard. Prominent Engine Yard customers. http://www.engineyard.com/company/customers.

[15] John Foley. Private Clouds Take Shape. *InformationWeek*, 2008. http://www.informationweek.com/news/209904474.

[16] God. God homepage. http://godrb.com/.

[17] Google. Project page of Chrome OS. http://www.chromium.org/chromium-os.

[18] James Edward Gray II. The Ruby VM: Episode III.
http://blog.grayproductions.net/articles/the_ruby_vm_episode_iii.

[19] James Gregory, Daniel Beardsley, and flq. Is there benefit to using Monit instead of a basic Upstart setup?, 2011. http://stackoverflow.com/q/4722675/506367.

[20] HackageDB. Data.ByteString.Lazy library documentation.
http://hackage.haskell.org/packages/archive/bytestring/0.9.2.1/doc/html/Data-ByteString-Lazy.html.

[21] HackageDB. Haskell packages library.
http://hackage.haskell.org/packages/hackage.html.

[22] HackageDB. Haskell serialisation library.
http://hackage.haskell.org/package/binary.

[23] HackageDB. Haskell standard library documentation.
http://hackage.haskell.org/package/base-4.5.0.0.

[24] HackageDB. hGetContents documentation. http://hackage.haskell.org/packages/archive/haskell98/2.0.0.1/doc/html/IO.html#v:hGetContents.

[25] Michael Hartl. Ruby on Rails Tutorial — Heroku setup, 2012.
http://ruby.railstutorial.org/chapters/beginning#sec:1.4.1.

[26] Haskell.org. Control.Concurrent.MVar library documentation. http://www.haskell.org/ghc/docs/7.4.1/html/libraries/base/Control-Concurrent-MVar.html.

[27] Haskell.org. Data.String library documentation. www.haskell.org/ghc/docs/7.4.1/html/libraries/base-4.5.0.0/Data-String.html.

[28] Haskell.org. Glasgow Haskell Compiler homepage. http://www.haskell.org/ghc/.

[29] Haskell.org. Haskell in industry.
http://www.haskell.org/haskellwiki/Haskell_in_industry.

[30] Haskell.org. Haskell.org: Introduction.
http://www.haskell.org/haskellwiki/Introduction.

[31] Haskell.org. Lazy evaluation.
http://www.haskell.org/haskellwiki/Lazy_evaluation.

[32] Haskell.org. RTS options for SMP parallelism. http://www.haskell.org/ghc/docs/7.4.1/html/users_guide/using-smp.html#parallel-options.

[33] Haskell.org. System.IO library documentation. http://www.haskell.org/ghc/docs/7.4.1/html/libraries/base/System-IO.html#t%3AHandle.

[34] Haskell.org. Why Haskell matters.
http://www.haskell.org/haskellwiki/Why_Haskell_matters.

[35] Heroku. CLI Usage. https://devcenter.heroku.com/articles/using-the-cli.

[36] Heroku. Deploying with Git. https://devcenter.heroku.com/articles/git.

[37] Heroku. Heroku homepage. http://www.heroku.com.

[38] Heroku. Heroku prices. http://www.heroku.com/pricing#5-0+shared.

[39] Heroku. List of Heroku supported software. https://addons.heroku.com/.

[40] Heroku. Read-only Filesystem.
https://devcenter.heroku.com/articles/read-only-filesystem.

[41] Heroku. Using AWS S3 to Store Static Assets and File Uploads.
https://devcenter.heroku.com/articles/s3.

[42] Jordan Hollinger. How to deploy a multi-threaded Rails app, 2011. http://www.jordanhollinger.com/2011/04/23/how-to-deploy-a-multi-threaded-rails-app.

[43] Karolis Juodelė, applicative, and ehird. hGetContents being too lazy.
http://stackoverflow.com/q/10485740/506367.

[44] keiter, mokus, and Dario. In what sense is the IO Monad pure?
http://stackoverflow.com/q/4063778/506367.

[45] Eric Kidd. High-Performance Haskell, 2007.
http://www.randomhacks.net/articles/2007/01/22/high-performance-haskell.

[46] Linode. Linode prices. https://manager.linode.com/signup/#plans.

[47] Guillaume Luccisano. Avoid memory leaks in your ruby/rails code and protect you against denial of service, 2010.
http://www.tricksonrails.com/2010/06/avoid%2Dmemory%2Dleaks%2Din%2Druby%2Drails%2Dcode%2Dand%2Dprotect%2Dagainst%2Ddenial%2Dof%2Dservice/.

[48] Simon Marlow, Simon Peyton Jones, and Satnam Singh. Runtime Support for Multicore Haskell, 2009.
http://community.haskell.org/~simonmar/papers/multicore-ghc.pdf.

[49] C. A. McCann and Bill. How do laziness and I/O work together in Haskell?, 2010.
http://stackoverflow.com/a/2777842/506367.

[50] Joel McCracken, mipadi, and Chris Eidhof. Why is lazy evaluation useful?, 2008.
http://stackoverflow.com/q/265392/506367.

[51] MongoDB. How does MongoDB provide concurrency? http://docs.mongodb.org/manual/faq/developers/#how-does-mongodb-provide-concurrency.

[52] MongoDB. MongoDB homepage. http://www.mongodb.org/.

[53] MongoDB. When does MongoDB write updates to disk. http://docs.mongodb.org/manual/faq/developers/#when-does-mongodb-write-updates-to-disk.

[54] Monit. Monit homepage. http://mmonit.com/monit/.

[55] Tom Mornini. Why should I use Heroku or Engine Yard's expensive hosting instead of Linode?, 2011. http://www.quora.com/Why%2Dshould%2DI%2Duse%2DHeroku%2Dor%2DEngine%2DYards%2Dexpensive%2Dhosting%2Dinstead%2Dof%2DLinode.

[56] Mozilla. Boot to Gecko project announcement.
https://wiki.mozilla.org/Booting_to_the_Web.

[57] Namecheap. Namecheap prices.
http://www.namecheap.com/web-hosting/web-hosting.aspx.

[58] Nginx. HttpUpstreamModule.
http://wiki.nginx.org/HttpUpstreamModule#upstream.

[59] Chris Okasaki. *Purely Functional Data Structures*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, 1996. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.64.3080&rep=rep1&type=pdf.

[60] Oracle. Java threading model.
http://java.sun.com/docs/hotspot/threads/threads.html.

[61] Oracle. java.lang.ref.Reference documentation.
http://docs.oracle.com/javase/7/docs/api/java/lang/ref/Reference.html.

[62] Bryan O'Sullivan, Don Stewart, and John Goerzen. *Real World Haskell*, chapter Profiling and optimization. O'Reilly, 2008.
http://book.realworldhaskell.org/read/profiling-and-optimization.html.

[63] Bryan O'Sullivan, Don Stewart, and John Goerzen. *Real World Haskell*, chapter Lazy I/O. O'Reilly, 2008. http://book.realworldhaskell.org/read/io.html#io.lazy.

[64] Bryan O'Sullivan, Don Stewart, and John Goerzen. *Real World Haskell*, chapter Side Effects with Lazy I/O. O'Reilly, 2008.
http://book.realworldhaskell.org/read/io.html#io.sideeffects.

[65] PickledOnion. Ubuntu Hardy — thin web server for Ruby, 2008. http://articles.slicehost.com/2008/5/6/ubuntu-hardy-thin-web-server-for-ruby.

[66] Lance Pollard, Lucas Jones, and Arto Bendiken. Running Cron Tasks on Heroku, 2010.
http://stackoverflow.com/q/2611914/506367.

[67] PostgreSQL. Connections and Authentication.
http://www.postgresql.org/docs/9.1/static/runtime-config-connection.html.

[68] PostgreSQL. Managing Kernel Resources.
http://www.postgresql.org/docs/9.1/static/kernel-resources.html.

[69] PostgreSQL. PostgreSQL homepage. http://www.postgresql.org/.

[70] Python.org wiki. Global interpreter lock in CPython.
http://wiki.python.org/moin/GlobalInterpreterLock.

[71] Norman Ramsey and devoured elysium. Immutable data structures performance, 2010.
http://stackoverflow.com/q/3233473/506367.

[72] Claus Reinke. Lazy IO and closing of file handles, 2007.
http://www.haskell.org/pipermail/haskell-cafe/2007-March/023535.html.

[73] Ruby on Rails Guides. Configuring Rails Applications.
http://guides.rubyonrails.org/configuring.html.

[74] Bobby S, Joel Spolsky, and Chuck. Pure Functional Language: Haskell, 2010. http://stackoverflow.com/q/4382223/506367.

[75] SadSido and sbi. Most common reasons for unstable bugs in C++?, 2009. http://stackoverflow.com/a/1346631/506367.

[76] Tony Smith. Chromebooks: the flop of 2011?, 2011. http://www.reghardware.com/2011/11/23/acer_samsung_chromebook_sales_struggle_to_top_30000_units/.

[77] Snap. Snap 0.3 Benchmarks with GHC 7.0.1, 2010. http://snapframework.com/blog/2010/11/17/snap-0.3-benchmarks.

[78] Michael Snoyman. Preliminary Warp Cross-Language Benchmarks, 2011. http://www.yesodweb.com/blog/2011/03/preliminary-warp-cross-language-benchmarks.

[79] Jarrod Spillers. git merge vs git rebase: Avoiding Rebase Hell, 2009. http://www.jarrodspillers.com/2009/08/19/git-merge-vs-git-rebase-avoiding-rebase-hell/.

[80] SQLite. Appropriate Uses For SQLite. http://www.sqlite.org/whentouse.html.

[81] SQLite. SQLite homepage. http://www.sqlite.org/.

[82] Top Coder. Using Tries. http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=usingTries.

[83] Mikayel Vardanyan. LAMP Security: 21 Tips for Apache, 2011. http://blog.monitis.com/index.php/2011/08/01/lamp-security-21-tips-for-apache/.

[84] Wikipedia. Linear search. http://en.wikipedia.org/wiki/Linear_search.

[85] Wikipedia. Readers-writer lock. http://en.wikipedia.org/wiki/Readers-writer_lock.

[86] Frank Wiles. Performance Tuning PostgreSQL. http://www.revsys.com/writings/postgresql-performance.html.

[87] Sudara Williams. That's Not a Memory Leak, It's Bloat, 2009. http://www.engineyard.com/blog/2009/thats-not-a-memory-leak-its-bloat/.