Overview Over My Bachelor's Thesis

Jannis Eichborn

June 5, 2014

*Things related to Usability later in the explanations

Contents

1 Introduction & Motivation

1.1 Coding in an Open-Source Environment

Brief Description and History
Here I want to talk about how software is developed, give a brief history on how open-source developed over time and present different approaches of people working together on large problems.
What are the main goals of this community and how are these goals expressed?

The Open-Source Community is Changing
What are recent developments, how is code shared, where do we want to go? Which ideals are fulfilled and which things are still in a messy state?

What is the Problem with Current Integrated Development Environments?
Introduction of usability in the context of working with IDEs
Description of problems in the workflow regarding spreading of code, packages in different forms and finding suitable discussion and documentation. Reuse of code is bad and tedious.

1.2 Our Idea for Working on Large Projects in the Future

Description of the idea of having more code in one place with interchangeable and competing solutions. Using crowd sourcing for evaluation...
How is the functionality of an online database for code integral part of the IDE's usability?

1.3 My Work & Goals - A Self-organizing Database for Code

What exactly will my work do in this context and what are my goals. Description of the structure of my thesis. What usability criteria do I aim for?

General notes - Abbreviations.

- I will use the term *function* for anything that is a function, method, routine or subroutine in a given language. Maybe just talking about Julia here?

2 Analysis & Formalisation

2.1 Description of Use-Cases - Client Requirements

This part will include extensive formalisations with respect to usability. How can this be formalised, what metrics can be derived? How do I define the client in this scenario and

what is important to this client?
I describe how people are going to use the client IDE and what requirements arise from this. I will talk about portability between different OSs, what performance clients want and other aspects which are important to the implementation. What are possible interfaces and how can I meet them?
First of all I will have to describe potential clients to my system. What kind of applications will make use of the database, how do they want to communicate and what does this interaction imply for the architecture of the database?
Well to begin with is should be obvious, that I will not be concerned with any kind of graphical interface to the database. It would store millions of entries with several different versions of each entry. Of course one could think of means to make an interactive query interface which can display the database in it entirety but in my mind this is no use case. Users should always query the database in the context of a more elaborate system on top. If you wanted to search for documentation or code only there are already thousands of possibilities to do so on the internet (TODO quote). From my point of view the system is designed to be used by an IDE or something equivalent. It does not mean that other more 'traditional' use-cases are not possible, but I will not be concerned with those in this chapter. Also the aimed database-representation would probably perform suboptimal for these kind of uses (multiple rapid queries at a time over simple indices). For these tasks there are solutions (TODO quote) out there and they cannot be the target use-cases for my system.
In my mind the focus of the system must be to aggregate over the contained data for more elaborate queries. For example a query in plain English would be something like this: "Give me all the function signatures which match this given set of parameters" or "Which functions have the same parameters and return the same type?".
These queries result from using the a possible IDE or smart query system on top of the database. Let me stick with using an IDE for a more extensive example: A User writes a function from the top of his head. He does not refer to anything in the database (yet), but instead simply writes ahead.
Right after he has entered the signature the system can check for the following:

1. Is there already a function with the exact *same signature*?
   Should definitely be shown to the user. Maybe somebody has already done all the work he is going to do now. This might even be an expert in this area who has spend approximately 42 hours upon perfecting the runtime-behavior of this single function. In this case users might be happy to just plug-in the given function as is, or at least make use of the code to get ideas for his own improvements. I think this case might sound pretty uncommon but with thousands of users which roughly confirm to naming-standards of functions, this feature might work pretty fine right out of the box. Also users could then go ahead and write their own variants of this function an re-upload it. This way entering the desired signature alone can be seen as a means to search for functionalities using all the knowledge that you put into the signature.

2. Is there already a function with the *same name*?

Even if the signature is no a match, there might still be quite a lot of use in showing functions which have the same name. They might be different approaches to the same functionality - variations in the parameters might just be due to implementation details - or functions which are similar in their behavior and might need only slight changes to fulfil a new role. As it is the intention to make code reusable and having the database managing all the (sometimes rivalling) code-fragments.

3. Is there already a function with a very *similar name*?
   Using some sort of similarity-measure, one could determine a class of function names which are similar enough to the given function name. Best case this is done using the general naming conventions of the language in question. There are two cases that are worthwhile to consider:

   a) Are the parameters the same?
      Then it is most propable that the user will want to see these results. There is quite a lot of information already in the typed parameter to a function. If those correspond it is worth a look to see whether this function might actually what the user wants in this context.

   b) Are the parameters - while not the same - at least similar
      If the parameters are completely different it will be highly unlikely that the function will do what the user wants to achieve. If however the parameters differ only marginally it might be interesting to see also these function in the database.

In general this all means that the database must be able to process these kinds of queries with reasonable results for all the given cases. While case 1) seems pretty straightforward, there might be problems when it comes to versioning of code pieces and finding the "best" functions which have the same signature. It is of no use for the user to find himself confronted with more than a dozen nightly-hacked matrix multiplications before the "standard" built-in matrix-multiplication is shown.
For all cases in 2) and 3) there is some sort of ranking necessary to present the client with usable results. Also one must consider, that you don't ever want to return *all* results as bandwidth would quickly become a limiting factor to the system, which makes ranking even more interesting. The system should make use of as much context-information as necessary, even drop down to comments in order to rank functions for a given query.

Possible queries

- Retrieval of functions:
  - Finding matching code to a signature.(might also include the return type of the function)
  - Find code to similar signatures or even without a name at all, just so you can see what would be possible with a set of arguments.

  - Finding matching code to a block of code. (Potentially hazardous)
  - Retrieving a newer version of code (maybe just the difference)
  - Retrieving a whole package/module with all functions
  - Retrieving meta-information to functions. Documentation, connections to other code, dependencies/using/requirements, variations, older version etc...

- Uploading functions:
  - A completely new function, without any links.
  - Upload an entire module/package
  - An update/extension to a function which the user wrote
  - A variant to a function.
  - Comments to a function
  - A rating for a function
  - A connection between code

Defining the desired experience from a client's point of view

- Most obviously: Having a consistent, simple and well documented interface to the database. It is well defined what queries can be made (and how, maybe using DTD) as well as what one can expect as a response to those queries.

- The most important criteria: Having fast response times for simple queries (meaning super-fast retrieval of single functions). Searches for larger sets should yield responses quickly as well. Initial guess less than 500ms for first results to arrive.

- Finding the most current version of a function

- Finding the most relevant function if there are multiple implementations

- If one searches for a set of functions or if a query has mutliple matching results, the client will expect some kind of order on the results. This order should feel natural and consistent as people have become used to by modern search engines. If the database fails to deliever at this level people are far less likely to interact with the code and make updates to the system.

- Up to this point I do not want to aim for auto-completion. The client should provide one query and expect a timely answer to it (no streaming yet). Later this feature might get interesting, especially once the database gets partitioned over several clusters and results are aggregated from different sub-graphs. For now my goal will be batched processing of one query after the other.

- Having consistent results. The same query deliveres the same answers, similar queries lead to similar results. Priorization should be transparent to a client, or even configurable (using some sort of arguments for a search).

- If a client updates information in the database or uploads new content, he will want to get feedback whether this was successful or not. Some auxiliary information might be interesting (what version, where, what size, how long etc...). If the transaction failed for whatever reason the client wants to know why.

Required funtionalities derived from the desired experience and possible queries

- Fast (indexed) access to the data in various forms:
  - function-names (some intelligent search like elastic search to be able to find similar names)
  - parameters (in combination with the name or without it)
  - return types in combination with the above.
  - metainformation (not a priority)
- Using these finding mechanism the database must be able to return:
  - A whole function, if there is exactly one hit, in a protocolled and serialized manner.
  - Requested parts of functions (comments, ratings, documentation)
  - Being able to traverse the graph in order to find connections (versions, similar functions, parameters, usage links etc...). **This functionalality is very interesting but is also vast and sometimes difficult. Thus it is not a priority.**
- Also the database must return sets of functions:
  - If a client wants to see/retrieve a package.
  - If a client searches functions similar to a given signature.
  - If a client only provides parameters and wants to find matching functions.

2.2 Non-functional (technical) Requirements to the System

This section focuses on things like scalability, robustness, the iteration speed of changes, whether and where changes to the design are possible and how security and safety standards can or have to be met. Which things are necessary and which ones are nice to have?

Building an interface
The first task is to provide the functionality of the database-system to a remote client. There must be a suitable protocol in place which can handle the transfer or queries and the responses. Criteria: Quick, secure, fail-safe, well-known and understood. The datastream needed should be minimal to avoid loss of data as well as reducing the time/bandwidth needed for the transactions. A client should be able to use the system without a top-notch internet connection.

Trying to minimize the time needed for the queries
This is a central requirement.. Making data accessible is what the database is all about. The client-acceptance of the system is dependent on quick response of the system. Having a neat interface (above) is the first step. Furthermore the system must exhibitthe following properties, which I will explain below:

1. Consistency of the data in order to keep the amount of space needed for the data minimal.
2. Monitoring as a means to detect failures and bottlenecks in the system.
3. Scalability.
4. Ranking of results.
5. Feedback when things timeout or fail.

1) Consistency of the data

- Being able to produce a *unique* ID for every entry which encodes name, signature and timestamp for the given function which was uploaded. This ID must be unique for every entry but still encode in an understandable manner. A function can be translated to an ID and it must be possible to retrieve a function from an ID. (Indexing or traversals)
- The database needs a valid scheme which represents structures in the data. Since I will use a graph-based system there has to be a graph-representation of the data, which is extendable and concise at the same time in order to prevent overhead in the data-representation while still having the means to extends functionalities in the system.
- Items which are put into the database have to be addressable, properly connected and indexed, so that there is no zombie-data in the database. Transactions need to be atomic for retries, dispatchement over different threads and support of thousands of clients.
- One very important aspect is proper versioning of functions. Initially the database should be able to store as many version as the user likes. Over time it could be interesting to implement some garbage collection behavior to sort out unused nodes in the database to prevent the system from becoming overloaded in slow. There must be some way to monitor this behavior.
- Having a proper indexing system.

2) Monitoring
In order to assess the functionality of the database there have to be mechanism to generate and use simple metrics over the system. These metrics can then be used to optimize the behavior of the graph and to evaluate the success of different functionalities. Possible metrics are:

- Number of transactions per second.
- Overall size of the graph; number of nodes/edges.
- Average number of connections between nodes. Distribution of types of connections.
- Average number of versions for one function (or other means of central tendency).
- How the client-access is distributed, how many calls of what category?
- Average times for operations.
- Error counts, failure statistics.

3) Scalability

- The performance of the database must be consistent with growing input. One cannot assume constant behavior but linear would already be pretty bad. Mechanisms must be in place which allow efficient indexing and sorting of data.
- Vertical scalability: If a machine has more resources it should be able to make use of those. More cores mean more threads. Atomic transactions have to be distributed over all available threads using the provided memory in the heap. That means the database will have to be designed for heavy concurrency in all possible places (reading, writing, garbage collection, metrics).
- Horizontal scalability: With more and more data it is vital to be able to partition the data over multiple machines/drives. This should be considered from the start. The database should never be a monolithic process in a single JVM, or at least it must be possible to change that without rewriting the whole codebase. This includes considerations about the consistency of the data over multiple instances or partitions.

4) Ranking of results

- Using some metrics to cap the search-space and the amount of data which has to be retrieved.
- This also means not to squander with the number of individual transactions (possibly concurrent) to the database, but to make processing batched. This will help serving a lot of clients while still providing fast answering to every one of them.
- There has to be a good understanding of the use-cases and distribution of usage to tune this system. Therefore the initial capabilities will be limited to rudimentary features.

2.3 Goals of my Work

Which derive from the requirements above. Explicit goals with respect to usability.

2.4 Structure of My Work

What comes first. How do I want to accomplish the goals and what is the prioritization.

3 Relevant Basics

3.1 JVM - Choice of Language and Context

Why it is still relevant in the context of large-scale distributed computation

3.2 SQL to NoSQL - Why Graph Databases?

What are recent developments in the requirements on databases and how are those met. New types of databases are emerging.

3.3 Distributed Databases

Distributed computations require new forms of data-management. Distributed systems lead to more scalability and robustness in the case of hardware-failures.

3.4 General Thoughts on Performance & Scalability in Recent Software Design

What is the bottleneck in performance nowadays, how is that important to me.

3.5 Usability in the Context of Technical Systems

Usability without interfaces. Including thoughts about usability in the structure of a program/package.

4 Evaluation & Verification

4.1 Why Thinking About Testing Right from the Start Is a Good Idea

Test-driven design and other thoughts..
How do I test for the usability I introduced earlier?

4.2 What I Need to Test

Description of formal aspects which have to be tested. Derivation of suitable measures for the problems.

4.3 How I Test

What are the principles in the implementation of my tests.
Implementation of usability measures

5 Design & Implementation

5.1 Used Software Packages and Tools

Java has been around for nearly 20 years [1] and with it came the evolution of the Java Virtual Machine (JVM). As given above there were plenty of reasons for me to make use of the JVM throughout the design of my software. As I chose Scala as my programming language for the project, I can still make use of any JVM-based package which is very convenient on the one hand but also leaves an intimidatingly huge set of possible tools to implement the desired behavior. In this chapter I will name the tools I chose and try to explain why I did so with respect to the goals and specifications stated earlier.

5.1.1 Scala

Since Scala compiles into Java-Bytecode it can be seen as a direct surrogate for using Java. The programmer is free to inclue Java packages and dependencies and one can even use plain Java-code if desired. There were multiple reasons for me to choose Scala over Java:

Scala is more modern
Scala was developed from 2001 on and saw its first release in 2003. Martin Odersky was also involved in developing the *Pizza* programming language, the javac compiler and Java Generics in the late 90's which influenced the creation of Scala.[9]. The resulting language is a hybrid of functional programming elements and object oriented software design. In my personal opinion the simplification of code and the reduction for boilerplate code using functional paradigms is a general trend in the last years and will continue to become more important. Java 8 will include the use of lambda comprehensions and Apple just introduced its new programming language called *Swift* which builds upon Objective-C and includes features derived from functional paradigms to make code more concise and comprehensible. As it seems more purely functional programming languages such as Haskell or Curry did not yet prove their usability in a production context and are still way off from the object oriented languages such as C, Java and Objective-C which make up the first three entries in the TIOBE software index. [5] In the mentioned index Scala is still way back (place 35) but the point is that general interest in including more functional paradigms into languages exists such as in the high demand of C# , JavaScript or Python which all exhibit elements of functional programming mostly to make the code more concise and enabling efficient coding in a readable manner.
With these developments in the back of your mind I think it is more desirable than ever to harness the power of functional programming because the future points in this direction

and many of the smartest minds focus on the reasonable incorporation of functional elements into daily coding.

Why functional programming is of interest to me

As the supervisor of an internship I had in Silicon Valley put it "Scala reduces the amount of code written by rougly 30%. 30% less code results in 30% less stupid mistakes in my experience. That was the main reason to introduce Scala to the company". Less boilerplate means less copy and paste, less wrong variable use and more focus on the lines of code you actually write. This of course is a weak argument although it captures some of the nice benefits of functional programming. Nonetheless you could achieve the very same thing I wrote in Java. All tools used also work with Java.

But there are some aspects which really make the use of Scala handy.
First of all there is the concurrency model I use which is Actor-based concurrency. The idea is that there are no globally shared, mutable states but that all state is encapsuled by actors. This paradigm was developed in the Erlang language, which is functional and was incorporated into Scala under the name of Akka 5.1.2. Scala feels much more native to this concept since good style in Scala means immutable variables and local scopes rather than global sharing of state.
The fluid transformations of datastructures enables by the rich toolset of Scala collections makes data-transformation which is needed in most steps of my program very intuitive and also efficient. Let me give an example for that: Say you want to take an incoming String from an input to the system which represents an xml and parse all Julia functions contained in it and then check the database, if this function already exists and create it or update it respectively.

```
for {
  function <- juliaParsing.parseFunctionsFromXml(xmlString)
} {
  Database.findFunctionBySignature(function.signature) match {
    case Some(f) => //update the data
    case None    => //create new entry
  }
}
```

The for-comprehension iterates over all the retrieved functions of let's say type `Function` without worrying about the specific type of the result, as long as it is an `Iterable<Function>`. The `findFunctionBySignature` method then retrieves the functions if there are any in the database. The result can than be pattern matches to implement the desired behavior. Variables are tighly scoped in this contruct and can be efficiently garbage collected after use. The `function` variable is very transient yet requires not mutably declared temporary variable. For comprehensions over the collections in Scala work mondadically and are optimized by the compiler which also makes them fast on runtime. Constructs like the one above appear very often in my system for retrieval of data and from my point of view the presented piece of code structures the behavior quite elegantly. It also capsulates the absence of data without the need for any kind of exceptions. Constructs like these make Scala so appealing to me and in my mind also lead to code which is better readable from the outside. My experience is that writing in Scala lets me focus more on the logics behind my transformations rather than looking up API's all day.

5.1.2 Akka

Akka [2] is a concurrency-toolkit which is based on the Erlang concurrency model of *Actors* [4]. The principle idea is that shared mutable state leads to a lot of problems in a multithreaded context. First of all the shared variables have to be locked or mutexed to prevent dirty reads or writes or even data corruption. This blocking behavior means that threads will be idling while they wait for their turn to access the locked variable. It also includes the introduction of boilerplate code to check for the lock and waiting behavior. Also this design is normally not intuitive for a programmer and means adapting to the mindset of a machine rather than having a model which can map a human understanding of parallelism on the machine in an efficient manner.
At the base of the construct is the actor-system in which all the actors live. It comprises a hierarchical structure of actors with their respective children. Thus each actor has supervisor or parent, which decides what to do when an actor crashes and a mailbox which is the only means to communicate with it. An actor is not equal to a thread although it can be implemented, that each actor lives on a single thread (which is not recommended, though). This makes actors totally asynchronous and non-blocking since they will capsulate their behavior, carry out their task when given processor-time and send a message with the result of the computation to another actor. So an actor implements behavior which is triggered by incoming messages, the output is generated by outgoing messages. In Scala these messages can be any kind of class which means there is no boxing needed for transportation between actors.
I want to give a very simle example on how this model works:
Say we want to do an everyday task also related to my work at hand: *Distributing incoming work from a port.*
As seen in figure 1 the resource is encapsuled by an actor and not directly accessible from the outside anymore. The Resource actor has sole access to the resource and internally bufferes data. The actors which want to process the incoming data need to be known to the Resource so that it can push the incoming work to the processing actors. The source actor will try to find a free actor for the input and then send a message containing the data to it. If

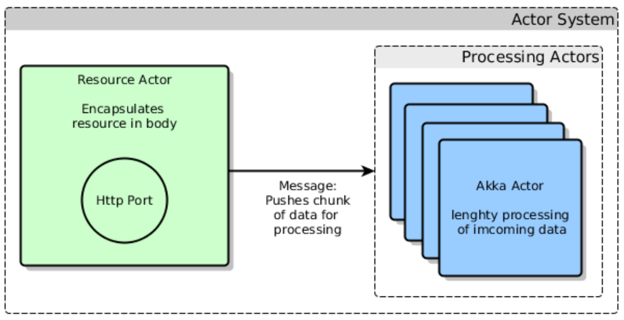
Actor System
Processing Actors
Resource Actor
Encapsulates resource in body
Http Port
Message: Pushes chunk of data for processing
Akka Actor
lenghty processing of imcoming data

Figure 1: Simple arrangements of actors

there is no free worker, it will
buffer the data. I am deliberately
leaving out lots of details which would be needed for a functioning implementation, this is just a demonstration of the design principles used. Already here some things become visible: Actors will not block a thread when there is no work to be done. Only when the source distributes work, the workers will come alive and start doing things, instead of busy waiting or blocking on the source. This is how the system gets more event-driven which makes a lot of sense in an application which is dependent on input from outside clients. Failure is another important aspect of an actor system. Let one task in a worker crash due to an unexpected error in the processing of the data, i.e. an error while parsing a data-structure from the input String. One principle of Akka concurrency is not to care too much about all possible exception but just letting the actor crash. If this happens it will send a termination message containing the error to its parent and/or supervising actor. The supervisor can then decide whether the actor should just continue working using the next message in its mailbox, be restarted, or stopped altogether. In Akka *and Erlang) this is called the *Error Kernel Pattern* which means that delicate tasks which might crash are deligated to disposable actors which carry no crucial data.
Now for the example the processing actors could be supervised by the source itself, which could decide what to do if one actor crashes. Or this role could be assigned to a designated supervisor actor so that the source and the processing workers live independently, where the supervisor with its children could represent something like a threadpool. See figure 2.

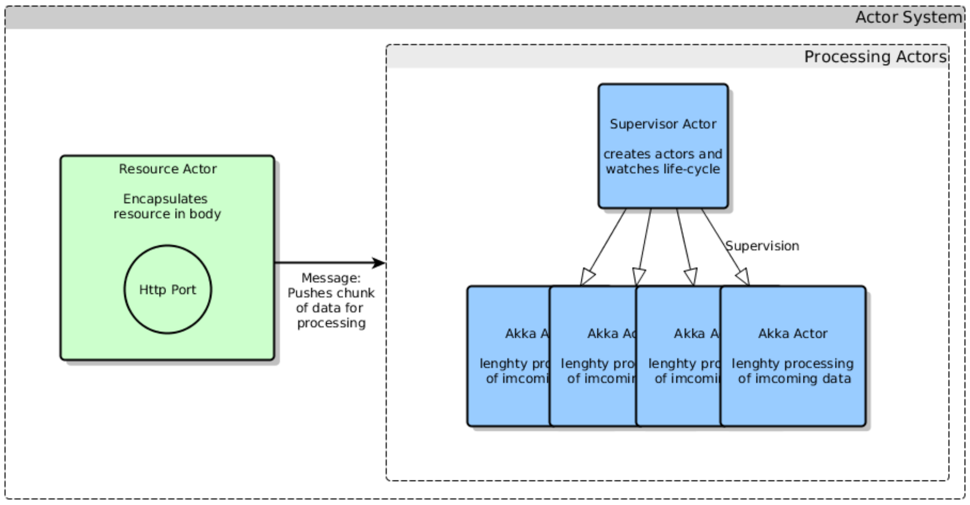

Figure 2: Arrangement of actors with supervision

This very flexible model for concurrency allows for the modelling of complex data-flows in large systems. Using supervision and messaging most software patterns such as producer/consumer scenarios or divide and conquer can be implemented elegantly without

race conditions, busy waiting, or excessive spawning of threads. Akka has shown some impressive examples on how to run large-scala systems. In Fall 2013 they ran a cluster of 2400 nodes on Google Compute Engine which shows how well scaling out via remoting is possible [8]. Also it is possible to scala up quite drastically. One benchmark shows how a single machine - although an immensely powerful one with 48 cores and 128GB RAM (one of the biggest Amazon EC2 machines, the r3.4xlarge, has 16 cores and 122GB RAM) [3] - throughputs around 50 million messages a second in a tuned out system. [7]
I have already worked quite a bit with Akka and am quite pleased with how fast I was able to learn the basics and scale out systems without having to refactor the whole code I wrote for testing. It is easy though to write actor-based code which is quite hard to read since computations happen all throughout the code. A code factoring and a clear documentation seem of outmost importance to me.
Akka is - all in all - a very powerful and flexible tool which provides the possibility to scale in all dimensions. It is a complex tool though with quite some overhead in the implementation. Once this is overcome though, it is an excellent tool to scale a system without having to restructure the code too much.

5.1.3 Play

Play is a framework which

5.1.4 Blueprints-Stack

5.1.5 Titan Database

5.1.6 Elastic Search

5.2 First Sketch - Design of my Implementation

how do I tackle the problems and requirements?

5.3 More Detailed Description of the Implementation

To a necessary degree of precision that is.

5.4 Evaluation of the First Sketch

What is good, what needs to be done...

5.5 Description of the Iteration

5.6 Further Evaluation

6 Conclusions

6.1 State of the Implementation at the End of My Work

6.2 Comparison to Requirements and Goals

What was fulfilled, what not and what might be critical. Can the application be extended? What have I achieved at what can people to with it at the time?
Is the code I wrote usable from the clients perspective?

6.3 What to Come - Future from here on

What are the next steps and which people can get involved. How do I see the chances in the future. Concluding thoughts on performance and scalability.

6.4 Summary

Testing features

This is a citation [6]
This is an image as a floating object with a ref to it: See Figure 3 on page 16

Figure 3: Ne fiese text is dat

Appendix A - References

References

[1] JAVASOFT SHIPS JAVA 1.0. `http://web.archive.org/web/20080205101616/http://www.sun.com/smi/Press/sunflash/1996-01/sunflash.960123.10561.xml`, 1996. [Online; accessed 1-June-2014].

[2] Akka Homepage. `http://akka.io/`, 2014. [Online; accessed 1-June-2014].

[3] Amazon EC2 Pricing. `http://aws.amazon.com/ec2/pricing/`, 2014. [Online; accessed 1-June-2014].

[4] Getting Started with Erlang User's Guide Version 6.0 - Chapter 3. Concurrent Programming. `http://www.erlang.org/doc/getting_started/conc_prog.html`, 2014. [Online; accessed 1-June-2014].

[5] TIOBE Index for May 2014. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`, 2014. [Online; accessed 1-June-2014].

[6] Fay Chang, Jeffrey Dean, and Sanjay Ghemawat. Bigtable: A distributed storage system for structured data. *... on Computer Systems (...*, 2008.

[7] Patrik Nordwall. 50 Million Messages per Second - On a Single Machine. `http://letitcrash.com/post/20397701710/50-million-messages-per-second-on-a-single-machine`, 2013. [Online; accessed 1-June-2014].

[8] Patrik Nordwall. Running a 2400 Akka Nodes Cluster on Google Compute Engine. `http://typesafe.com/blog/running-a-2400-akka-nodes-cluster-on-google-compute-engine`, 2013. [Online; accessed 1-June-2014].

[9] Martin Odersky. A Brief History of Scala. `http://www.artima.com/weblogs/viewpost.jsp?thread=163733`, 2006. [Online; accessed 1-June-2014].