0.1

0.2

0.3 Declaration of originality

0.4

0.5 Introduction

0.6 Preparation

In this chapter I will briefly describe how I prepared for my project. What led to the design decisions I made and how they influence the outcome of the project.

I had two goals for my project:

Firstly I wanted my search server to be a program that would be run locally alongside installations of online social networks. By having the program run by different online social networks the burden of running the search network I provide is evenly distributed between the online social networks using it.

Secondly I wanted to make it easy for the creators of the online social networks to integrate my service into their online social network. I decided to do this by exposing my service through an HTTP based API because I know from experience that web developers find that convenient. It also helps creating modular system designs.

I decided to use Distributed Hash Tables as the backend datastore for my search network. This choice immediately presents benefits and drawbacks:

The benefits are that Distributed Hash Tables promise key-lookups involving a subset of the nodes proportional to the logarithm of the number of nodes in the network. If you are careful when assigning keys to data items so that they are uniformly spread across the keyspace, then the data and computational workload is evenly spread out between the nodes participating in the network. Also a plus, and eventually what made using a distributed hash table an excellent idea, is that it allows nodes to join and leave at will. This is extremely important in a search network that is built upon the notion of social networks joining as they themselves see fit, and where there is no central authority that can guarantee computational resources being available or for that matter staying available.

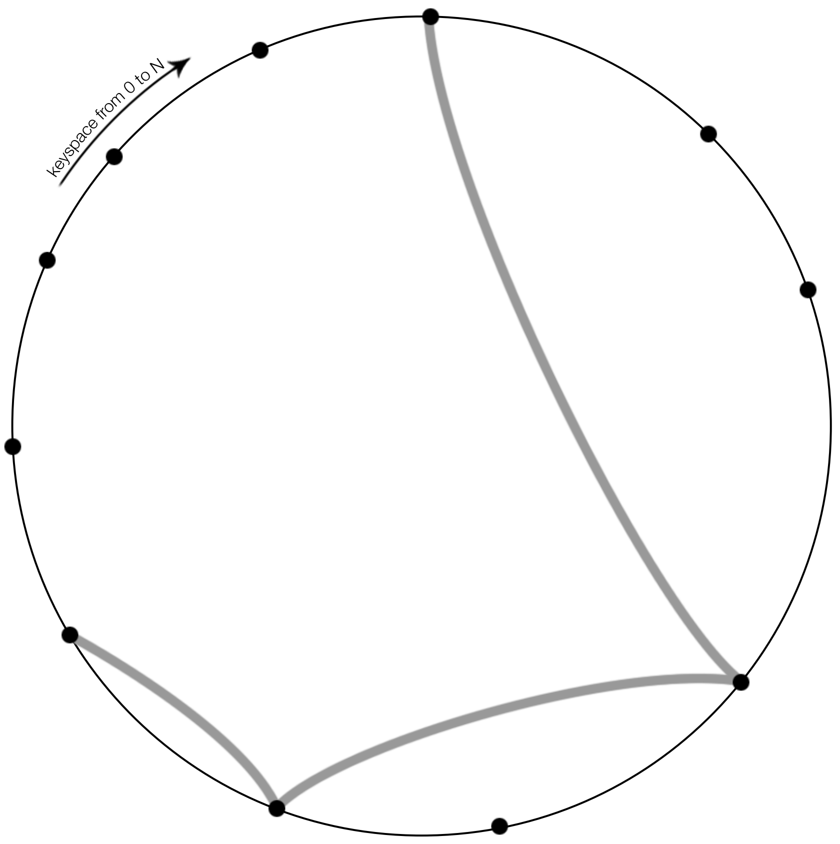

Figure 1: This diagram illustrates the linear increasing keyspace of a Distributed Hash Table network. The nodes in the network are shown as dots. The grey line connecting the dots illustrates how a key-lookup involves a subset of the nodes. Notice how the key space distance between consecutive nodes involved in the lookup decreases as we reach the no storing the value. This illustrates, although in a handwavey way how a lookup involved a subset proportional in size to the logarithm of the number of nodes in the network.

A big drawback is that distributed hash tables are key-value stores. It is not immediately obvious how one can efficiently search across data stored under keys unless the keys are known. I arrived at a solution that in my opinion circumvents this shortcoming quite elegantly. I will discuss it further in the implementation chapter.

During my preparation I evaluated different distributed hash tables. I initially decided to implement Chord, Pastry and Kademilia, and to compare their relative benefits and drawbacks, but later decided that Chord and Pastry would give me sufficient data to study and free up time to do a more thorough analysis. This seems a worthwhile tradeof, and in the interest of the project as a whole, especially considering one of the main factors I wanted to evaluate was whether Distributed Hash Tables can successfully be used as the datastore for a search network, which can equally well be evaluated with two as with three Distributed Hash Tables.

I decided to use the programming language Erlang for my implementation. Programming distributed systems is one of the key strengths of Erlang, and the project is of a highly distributed nature. Another key strength of Erlang is writing fault tolerant systems. This is achieved through hierarchies of supervising processes that ensure the system stays alive. Together these two traits of Erlang makes it extremely well suited for my project.

In the first stages of my project I spent considerable time learning Erlang, it's libraries and frameworks. I also learned how to use third party libraries for creating webservices through erlang with webmachine, and compiling and analysing code with rebar.

In this chapter I discussed how I decided that my project should expose its functionality through an HTTP API, why I chose to develop distributed hash tables, and why Erlang was chosen as the language of implementation. In the next chapter I will discuss the implementation of my project.

0.7 Implementation

Intro

0.7.1 General discussion

Before I start discussing how I implemented my project, I want to discuss what data I am interested in storing in my search network, and how I built a search index on top of a distributed hash table.

My search network stores records with information about users, much like a normal address book does. I made the following fields mandatory: the person's name, a url to her online social network profile page, and a url to a profile image that can be displayed alongside the search result.

For the purpose of this project I did not want to lock down exactly what data should be stored, but before my system is used in a wider context it would be wise to look into using some well known public standard. At this stage it is outside the scope of my project.

As I briefly discussed in the preparation chapter, a key-value store does not immediately lend itself to search. If you don't know the key under which a data item is stored, you would have to do a linear search across the keyspace in order to find what you are looking for. The solution is to use the persons name as the key. Human names in their natural form do not easily map uniformly onto any keyspace. They are of different length and also cluster around more common names. I therefore used a hashing function to hash names into keys. This immediately presents us with another problem. If you don't know the exact spelling of the name that was used to generate the key you still can't get around a linear search through the datastore. You can partly solve this problem by normalising names before hashing them into keys, but the utility of a search engine that requires you to already know a complete and correct match of what you are looking for is arguably rather low. Users today increasignly expect search engines to deliver predictive searches where results show before you finish typing your search query. This scheme does neither accommodate this, nor misspelled names, or for that matter cases where you have forgotten your friends middle name.

I solved the problem in the following way: Instead of storing just profile records I store both *profile records* and *link records*. The profile records contain all the information about the user and are stored under a key generated by hashing the whole record. The link records on the other hand are lightweight pointers to the full records. They contain the full name and the key of the profile record, and are stored under a key generated by a subset of the name. Each profile record has multiple link records pointing at it. I will come back to how I am generating these link records shortly.

Using link records solves several problems. By storing links keyed by fragments of a users name, the search engine can start looking up information before the user has completed typing the search phrase. This enables predictive search. Since the link records also contain the full name of the user you can detect and work around misspelled names, or even searches that don't include all the users names.

In the current implementation I create link records for each three additional characters contained in a users name. This is illustrated in figure 2. The process is done separately for each name. First the first three characters of a name is taken. Then for the next key the subsequent three characters are added. This

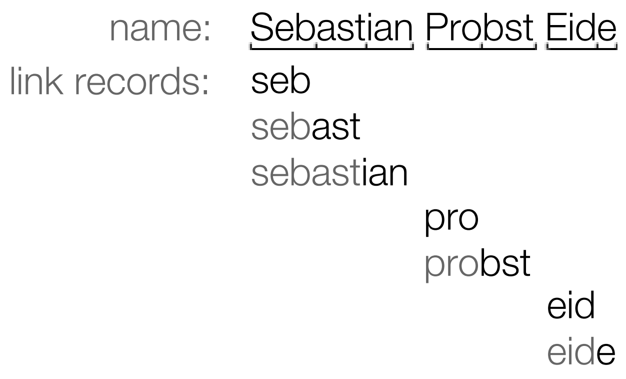

Figure 2: Here it is shown how a name maps into name fragments used for storing link records pointing to a profile record.

process is repeated until you reach the end of a name. When you reach the end of a name that is not a multiple of 3, an additional link record is created for the full length of the name. In the example you see this illustrated in how Eide results in the link records *eid* and *eide*.

Using link records does generate extra overhead when searching. A search will now no longer require a single lookup in the search index, but several. As the user types the search server will have to look up link records and resolve them to their profile records. If you searched for my full name, a total of 9 requests would have to be made. 8 for each of the link entries, and an additional one for looking up the full record with all the information about the user.

In reality it isn't quite as bad as it sounds. The desired result is likely to show up before the user has finished typing, in which case there won't be additional lookups for the remaining link items.

The link items also let the search server rank results by relevance. Let me illustrate this with another example. Say the system contains two profile records: one for Probst and one for Probstonius. If you search for Probst you get two link records, one for pro and one for probst. Both will return pointers to the profile records of both Probst and Probstonius. Since Probst is a complete match for the search term *probst* while Probstonius is only a partial match, the result for Probst can be given a higher relevance score. The current implementation does this.

¡¡¡DISCUSS COST OF THIS¿¿¿

0.7.2 Third party code used

In this section I want to briefly discuss what third party projects I was using in my project.

Most prominent is my dependence on OTP, the Open Telecom Platform, part of the Erlang standard distribution. It is a set of libraries that provide basic functionality needed to write amongst others general servers, and provide functionality for supervising processes within your application and restarting them when they fail. Most Erlang applications rely on OTP.

I also used Webmachine, an open source framework for creating HTTP interfaces in Erlang, and rebar, a build system. Both are developed by Basho.

For testing I used EUnit, part of the standard Erlang distribution, for regular functional unit tests and Erlymock for testing functions with side effects where I wanted to control the functions interactions with the environment.

0.7.3 Process

At this time it is well understood how Distributed Hash Tables work. The bulk of research in Distributed Hash Tables was done around year 2000. The research papers presenting Chord and Pastry also supply high level code that I referenced when implementing the algorithms. This provided an excellent foundation for doing test-driven development. All the core components of my system were unit tested, and in the tradition of test-driven development I wrote the minimal code needed to satisfy my tests. This encourages writing modular systems with self-contained and side effect free functions. I wrote additional integration tests where I felt it served a purpose.

Modularity was also a key design factor for other reasons. I wanted as many as possible of my modules to be ignorant of which Distributed Hash Table was being used. This encouraged me to keep the interface between the Distributed Hash Tables and the surrounding code strict and minimal. In effect the only methods the Distributed Hash Table's need to export are ways to get and set values by key.

0.7.4 Components of my system

I now want to describe the design of my system, and how all the different pieces fit together.

My project has two major parts, each its own application. The search application itself, and a hub application that runs on a central publicly known location.

The hub application acts as a rendevouz point for new Chord and Pastry nodes. The hub application is also used to control and initiate experiments.

I will first discuss the search application itself

The Search application

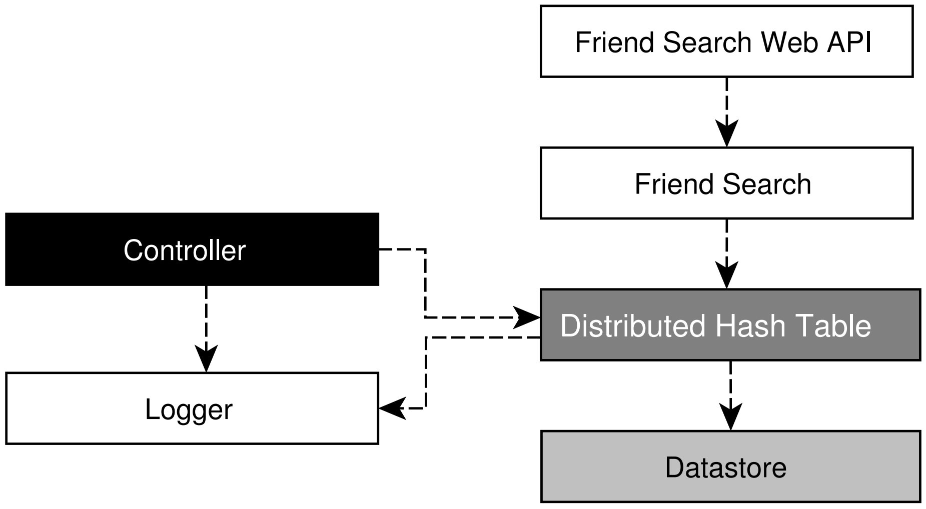

Figure 3: High level overview of the components of the search application. The arrows show how components interact.

Figure 3 shows the main high level components of the search system. I will briefly discuss them in turn:

The *Controller* is the heart of the application. It is the point of contact of the central hub application. Through the controller the hub application can select if the system should run Chord or Pastry nodes, and how many nodes should be run on that particular machine. The controller also issues requests during experimental runs. I will describe this in detail in the evaluation chapter.

The *Distributed Hash Table* component is an arbitrary number of nodes of either of the Distributed Hash Tables Chord and Pastry. Each node has a unique Id and is responsible for part of the keyspace. The nodes are fully autonomous parts of the distributed hash table network.

The *Datastore* is a key-value store responsible for storing the profile and link records. For efficiency reasons all the Distributed Hash Tables running on a single machine share the same Datastore. Each record has a time to live associated with it. If the record isn't updated before the time to live expires, the record is removed from the datastore.

The *Logger* is used during experiments to log the events taking place in the search network. Events can be everything from a node issuing a request, to a request being routed through a node.

The *Friend Search* server is the application actually performing the searches. When a user searches for a name, this name is converted into keys for link records. The search application uses the Distributed Hash Tables to look up the link records and then resolves them to the corresponding profile records that are displayed to the user. When an online social network wants to make its own profile records searchable, it adds them to its local Friend Search server. The Friend Search server is then responsible for storing the records and corresponding link records in the Distributed Hash Tables and refresh them periodically to ensure they are available in the network.

The *Friend Search Web API* exposes the Friend Search functionality through a HTTP API that can be used by the online social networks to integrate the Friend Search application into their online social networks.

Hub application

The hub application is substantially simpler than the search application. It acts as a rendevouz point for new Chord and Pastry nodes to get to meet in order to become part of the search network. The hub application also allows me to start and stop chord and pastry nodes remotely, as well as start automated experimental runs.

In a real world deployment of my project, the role of the hub application could be limited to simply being a rendevouz point.

Summary

0.8 Evaluation

0.8.1 Experimental design

In this section, *host* refers to a physical machine, while a *node* is an instance of a distributed hash table running on a host. Each host will run one or more nodes.

The focus will be on testing the practicality of using the distributed hash tables as back end stores for my friend search engine, and mainly focus on different aspects of resource consumption.

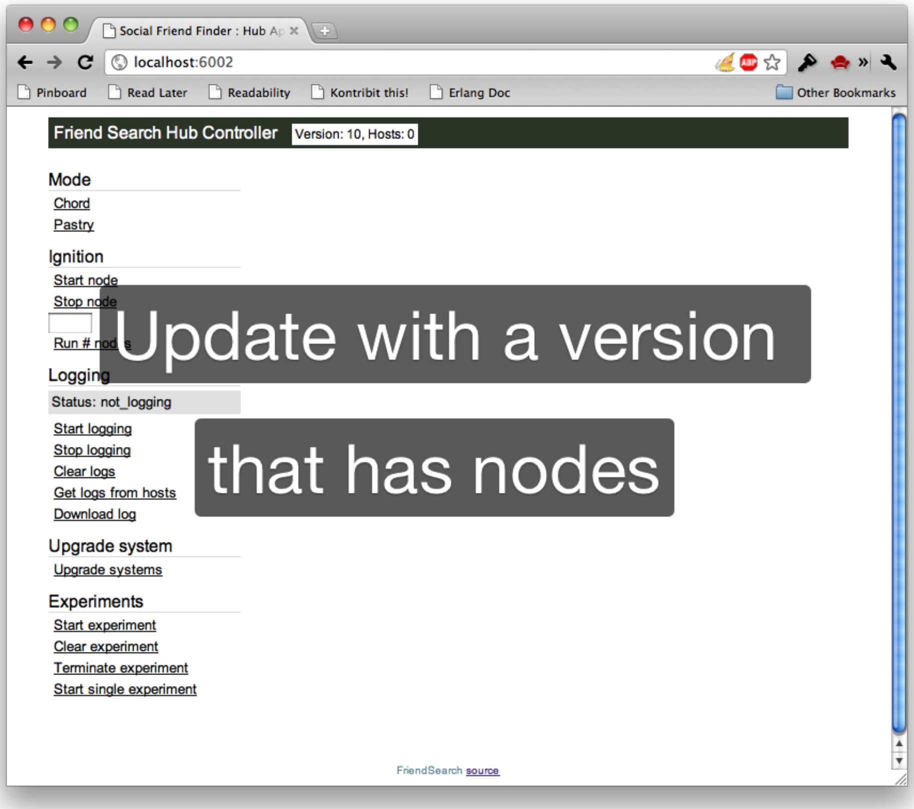

Figure 4: The Hub Application interface. A bare-bones and to the point web application that allows me to start and stop nodes, change between using Chord and Pastry, and start and stop experiments

What will be tested Four experiments will be performed:

1. The time it takes to perform a lookup
2. The number of nodes involved in a lookup
3. The amount of bandwidth required to do a lookup
4. The amount of bandwidth required to maintain routing state

Each experiment will be performed separatly for chord and pastry, in each case for networks of different sizes.

The experiments will follow a factorial design where the factors are:

- The number of nodes in the network

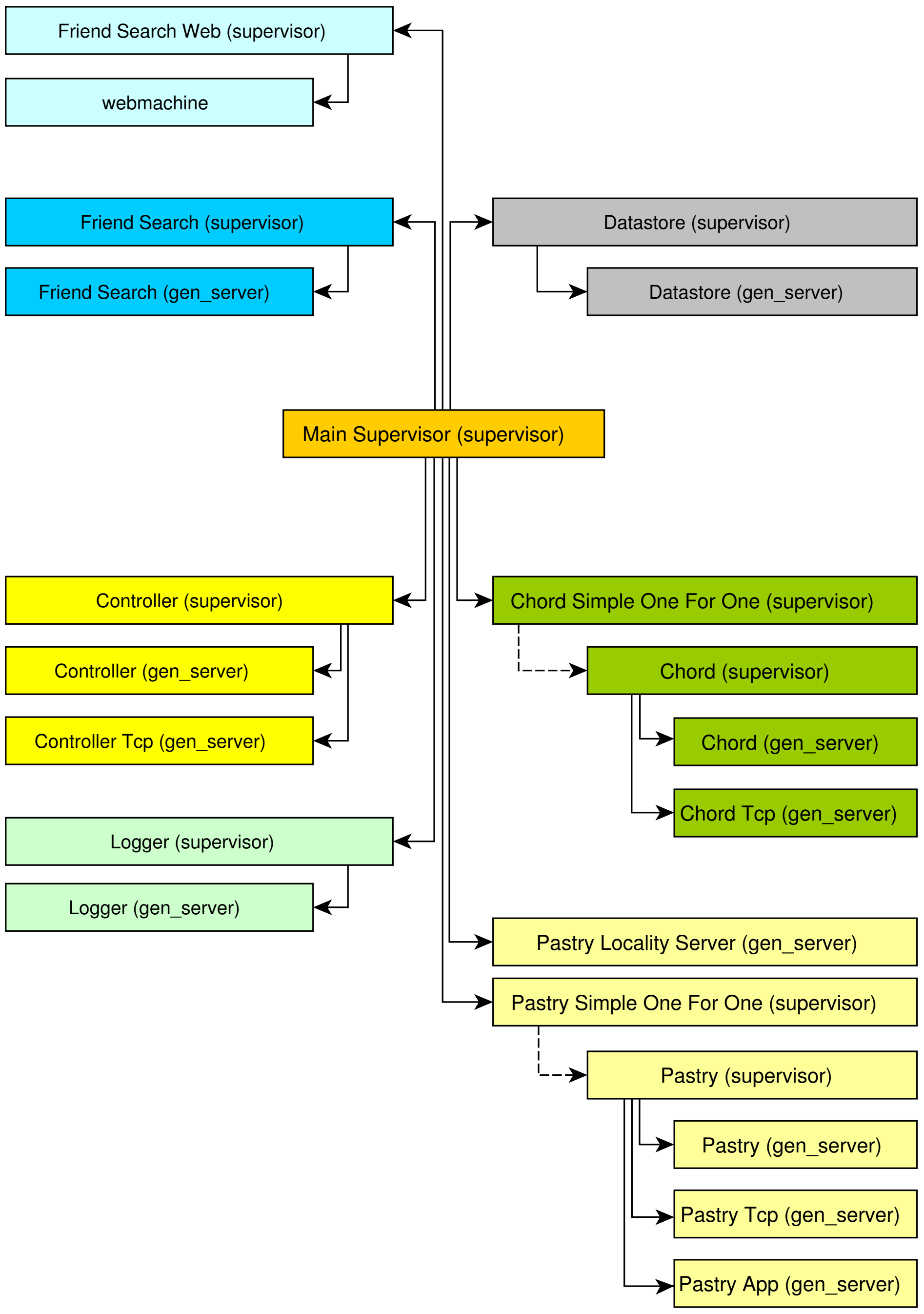

Figure 5: Shows the full supervision tree of all the components

- The rate of requests

In each experiment, given N hosts, the number of nodes will be 2^kN — k taking levels 0 through 3 per experimental run. As the number of nodes involved in any lookup is thought to be proportional to the logarithm of the number of nodes in a network, this seems likely to give interesting data to analyse.

None of these experiments rely on querying for real data. Keys to lookup will therefore be generated by hashing a combination of a counter value and the identity of the host performing the request. This is likely to give an even spread of keys.

The time it takes to perform a lookup This is an interesting experiment in terms of the expected end-user experience. We want to observe how the response time is affected by network size and request rate. The outcome of this experiment dictates how complicated queries the network can sustain, and more concretely if predictive searches in its current form can be sustained on a larger scale. Each host will individually issue request to produce a balanced load.

The factors are network size and the rate of requests. For each level of network size, each host will issue requests serially at increasing levels of parallelism until any one host sees a failure rate of more than 25%. Each level of parallelism will be kept for a minute.

The number of runs required for this experiment depends on how well the networks cope under load.

Number of nodes involved in a lookup This experiment focuses on what fraction of the network has to be involved in a request and how this scales with the size of the network. This is relevant as the number of nodes involved can indicate how well the network will manage under heavier load. The request rate is irrelevant for this experiment – the only factor is network size, and the response variable the number of nodes involved in a lookup.

For one run of this experiment, we need data for each of the 4 possible levels of network size.

The amount of bandwidth required to do a lookup This experiment is interesting in terms of real world applicability. The nodes are likely to be hosted by end users paying for the bandwidth consumed by their servers. The actual bandwidth consumption will to a certain extent be implementation specific, but the experiment will give us an indication of how chord and pastry compare.

This experiment will be performed as part of the experiment counting the number of nodes involved in a lookup.

The amount of bandwidth required to maintain routing state This experiment also helps determine if the implementations are usable in a real world setting. The bandwidth consumption will depend on the configurable options of chord and pastry. This makes a fair comparison hard. Configuration values will chosen that generally prove to give good *routing* performance.

This experiment will measure the bandwidth consumed per node over a 5 minute window per network size.

What will not be tested The behaviour of the networks depends on their different configurable options. In chord you can adjust the number of successors it keeps in touch with, and the frequency at which these relations and the routing table is kept up to data. In pastry in addition to choosing the rate at which the routing tables are maintained, there is a *b-parameter* influencing how many routing steps will be required but also the size of the routing tables.

Since the configurable parameters do not directly correspond between the two implementations, their levels will be fixed at values that generally give good and consistent *routing* performance.

While both my implementation of chord and pastry replicate their data for fault tolerance, this behaviour will not be tested. It is expected that the network size will not fluctuate wildly over time in a real world deployment, as the nodes are likely to be hosted on servers in operation over long periods of time. This is very much unlike other distributed hash table usage scenarios, like file sharing, where the amount of nodes joining and leaving the network at any given time can be significant (I don't have any data to back this claim with, but it sounds reasonable!?).

0.8.2 Experimental results

0.9 Conclusion

0.10

0.11

0.12

Friend search for distributed social networks

Sebastian Probst Eide, St Edmunds College

Originator: Sebastian Probst Eide

4 October 2010

Special Resources Required

Personal laptop for development and initial testing (1.86 Ghz, 2GB Ram)
CL machine with the Erlang VM installed as a backup
Virtual Server infrastructure, the likes of Amazon EC2, for parts of the evaluation

Project Supervisors: Dr David Eyers and Dr David Evans

Director of Studies: Dr Robert Harle

Project Overseers: Prof. Jon Crowcroft and Dr Simone Teufel

Introduction

It is hard to get started using independent, distributed online social networks as it is frequently difficult to find and connect with your existing friends, not knowing in which social networking system(s) they have their profiles and where those networks are hosted. The purpose of this project is to lay the foundation for a decentralised and distributed friend search engine that can be offered alongside installations of independent social networks allowing users to easily reconstruct their social graph in the online social network(s) of their choice. The focus of the project will be on the data storage layer of the search engine. I will compare and contrast different Distributed Hash Tables that I implement in Erlang. Erlang is chosen because it is known to be well suited for developing concurrent and distributed systems.

A front end, allowing basic searches to be performed, will also be created, but mainly to provide a way to rapidly exercise the data storage layer. More user-oriented functionality needed to allow the project to be used on a larger scale will be left out so as to limit the scope of the project.

Work that has to be done

There are a number of distributed hash table designs available.[2] I have decided to implement and compare the following three: Chord, Kademilia and Pastry. They were chosen because they are well documented algorithms, but differ in the way they perform their routing. As an example, consider how Pastry allows for heuristics based on anything from ping to available bandwidth or combinations thereof, Kademilia uses XOR arithmetic to determine the distance between nodes as a routing heuristic, while Chord has none of the above. These different approaches to the same problem make the algorithms excellent candidates to compare.

The main parts of the project are to:

- Implement the data storage layer of the search engine in Chord, Kademilia and Pastry using a uniform API that allows the system to use any one of the three without additional changes
- Implement infrastructure that facilitates testing and monitoring of the system. More specifically it should allow:

[2]Wikipedia currently lists 8 different protocols

  - Starting and stopping virtual servers across the different service providers to minimize server rental costs between testing sessions. The servers will be used to test different aspects of the distributed hash tables
  - Start and stop search nodes across the physical servers
  - Display how many search nodes are available in the system, and potentially some metric for how they are interconnected in terms of latency and bandwidth
  - Add and remove test data from the system. The system will be tested with *Database of names* from Facebook which I have access to. It is of significant size and importantly, contains keys with non-random distributions making for a more realistic dataset
  - Perform repeatable load testing on the system where tests as an example could compare read/write performance for different key and value sizes and different numbers of key-value pairs
  - Count the number of jumps and the time a key lookup needs in order to find a data item, in addition other appropriate descriptive statistical measures for writes and lookups in fixed key-value datasets
  - Being able to eliminate and add subsets of storage nodes in a repeatable fashion to test how the different Distributed Hash Tables cope with nodes disappearing and appearing

- Setup a Linux image that can be run across Infrastructure as a Service providers[3]
- Implement a web front end to allow users to perform basic searches across the data storage layer

Please note that while the following aspects of the search server are secondary to the project and will not initially be implemented, they all, should time permit, serve as excellent project extensions:

- Fuzzy searches allowing the user to misspell names.
- Support composite keys to allow searching for different attributes of a record
- Predictive searches

[3]Vagrant seems like a likely candidate to help automate this (http://vagrantup.com)

- Searches taking knowledge about social circles from online social networks, or similar metadata, into account in order to more intelligently prioritise and order the search results returned to the user
- Protections against malicious use of the storage network like broadcasting data, attempting to overload nodes with requests or using the network to store spam or pollute the namespace with spammy records

Starting Point

I have a reasonable working knowledge of Erlang and Linux and development of web based systems. The algorithms that will be implemented have all previously been implemented in other languages, and are used in production systems, so finding information about them should be possible. I have not yet used Amazon EC2 or any of the other Infrastructure as a Service (IaaS) providers that could be used to perform testing on the system in a distributed manner.

Success criterion

I regard the project as successful if I have working implementations of the three Distributed Hash Tables that allow me to set and retrieve values based on keys across a distributed network of machines, and also metrics for how the performance of key-lookup varies by node-, key-count and distributed hash table type and recommendations for future work based on the metrics collected.

The search component of the project, which is the application part that uses the distributed hash tables as a datastore to allow users to find their friends, has been left out of the success criterion due to it being harder to quantify and evaluate well.

Difficulties to Overcome

The following main learning tasks will have to be undertaken before the project can be started:

- To learn and fully understand the Chord, Kademilia and Pastry algorithms.
- To learn how to do network communication in Erlang other than the built in message passing. I prefer the individual nodes to communicate over TCP or UDP as that frees the design from assumptions regarding the language

of implementation. Additionally there are security issues when allowing Erlang VMs to connect directly in untrusted networks as any node is allowed to execute arbitrary code on any other connected node

- To find a way to test the system on geographically distributed nodes.

Resources

Some aspects of this project (amongst others the heuristics in Pastry that take locality into account when routing) are more interesting to test in nodes that are geographically distributed. For this reason using server instances from a provider like Amazon AWS or PlanetLab seem like a good idea. Ways of getting access to time on such infrastructure for academic purposes is currently being looked into. For the majority of the development cycle local testing will be just as interesting and can be done on my development machine.

This project requires no additional file space on University machines. I will be hosting the project source code and dissertation files in a repository on github.[4] If my machine breaks down, the development can be continued on any Unix based machine that has VIM, git and the Erlang VM installed.

Work Plan

Planned starting date is 15/10/2010.

Below follows a list of tasks that need to be done:

- Work through the theory behind Chord, Kademilia and Pastry and other items listed under *difficulties to overcome* (2 weeks).
- Implement initial version of Chord, Kademilia and Pastry in Erlang (4 weeks)
- Implement test harness to perform testing of the system (4 weeks)
- Implement a web frontend for search alongside a search server written in Erlang, that uses the distributed data storage layer for its data (2 weeks).
- Write dissertation (6 weeks).

[4]http://github.com/sebastian/Part-2-project

Michaelmas Term

By the end of this term I intend to have completed the research and learning tasks and have finished the first implementations of the Distributed Hash Tables in Erlang. In the vacation that follows I intend to make a good start on the testing harness and do a little work on the search server.

Lent Term

In the first half of this term I intend to finish the test harness and search server and spend time testing the system and solving problems. The tests could follow a factorial design where distributed hash table type, number of nodes, key-size, payload size, number of entries in the system and geographical distribution are all factors worth considering. What levels should be considered for each factor is yet to be determined and will become clearer as the project develops.

In the second half of this term I tend to get an initial draft of my dissertation written.

Easter Term

In this term I plan to polish the dissertation. The estimated completion date is the 15th of May, leaving a couple of days to let the dissertation rest before giving it a final read and correcting last minute mistakes before the due date on the 20th of May.