Sebastian Probst Eide

Friend search for independent distributed online social networks

Computer Science Tripos Part II

St Edmund's College — 2011

Name: Sebastian Probst Eide

Title of Dissertation: Friend search for distributed social network

Examination: Computer Science Tripos Part II, 2011

Word count: 11949

Project originator: Sebastian Probst Eide

Project supervisor: Dr David Evans

Aims:

To create a search engine using Distributed Hash Tables as the data store allowing users of independent distributed social networks to reconnect with their friends regardless of in which independent distributed online social network their profile is hosted.

Completed work:

- Implemented the Distributed Hash Tables Chord and Pastry in Erlang
- Created a search server allowing predictive and fuzzy searches across the data in the distributed hash tables
- Created a web application allowing me to remotely control all the nodes participating in the search network
- Evaluated the performance of my Distributed Hash Table implementations using infrastructure provided by Planet-Lab

Special difficulties:

None

I Sebastian Probst Eide of St Edmund's College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed:

Date:

Contents

Appendix

1. Introduction

In the last few years, multiple independent distributed online social networks have been made with the explicit goal of breaking Facebook's monopoly on social networking services and empower users by giving them ownership over their own data. While I applaud these initiatives, I also see a shortcoming they all have in common. Facebook makes it easy to find one's friends through search, whereas independent online social networks, often spanning multiple installations and providers, require their users to know where and in which online social network installation their friends have profiles. Not only does this make for a bad user experience, it also limits the potential these services might have to reach wider audiences and greater success.

I created a search engine that honours the ideals of the independent online social networks of giving their users control over what and how much data is made publicly available, yet at the same time letting them find and reconnect with their friends regardless of in which online social network they decide to host their profiles.

I built a proof of concept distributed search engine allowing predictive searches and fuzzy matching for misspelled and incomplete names. This search engine was built on top of Distributed Hash Tables. Each installation of an independent distributed online social network runs a copy of my software, and together make a search network available to all users. The cost of running the search engine is then distributed between the online social networks using it, and there would be no central authority with complete control over the data. Additionally, the independent online social network installations, or even individual users, could themselves decide how much and what data is made available.

I made working implementations of both the Distributed Hash Tables Chord and Pastry. While my implementation of Chord works sufficiently well, my implementation of Pastry is of very high performance.

Small scale testing also shows that the search network works as expected, providing user-friendly predictive searches and showing profile images alongside search results that are updating as the user types the search query.

I also created a web application allowing me to control the network of search nodes remotely, as well as initiate experimental runs and collect experimental data for my project evaluation.

2. Preparation

In this chapter we look at what considerations dictated the design choices made, why I chose Erlang as the language of implementation and how software engineering principles played a role in the successful implementation of my project.

2.1 Requirements analysis

My goal was to write a very particular kind of search engine, a search engine which, in the end, would work very much like a distributed oversized address book. Directory services providing address book functionality at scale already exist, but there are several benefits in making such a system distributed. As soon as there are more than one party contributing nodes to the distributed network, there will be no single entity owning all of the data, nor any single entity having to foot the bill for running the service. In addition, a well designed distributed system scales horizontally rather than vertically, meaning you can increase the system's performance by adding more cheap nodes run on commodity hardware. In turn, one can avoid buying new and more powerful machines to replace existing ones, cutting costs.

The idea of not having a central authority with ownership over the data resonates especially well with independent distributed online social networks, most of which where inspired by the idea of allowing their users to themselves maintain ownership of their own data and have detailed control over what is made visible to the world at large.

Having established a distributed system is desirable, I needed to build it in such a way that nodes in the search engine could easily be hosted by independent providers of distributed online social networks. Not only would the search engine have to be easy to install and run, but it should also be easy to integrate into a wide variety of different services, all built by different teams. The search engine would also need to handle nodes joining and leaving without interruption to the service. Handling high node churn in distributed environments without sacrificing reliability and speed is what Distributed Hash Tables are designed for. I therefore decided I wanted to use Distributed Hash Tables as the back end data store for my search engine.

From experience, I know that web developers find it convenient to integrate services that expose an HTTP interface, so in order to meet the requirement of easy integration, I decided my search engine would expose its functionality through a HTTP based API and that the data would be sent as JSON (JavaScript Object Notation) for easy manipulation and consumption.

Today, both search engines and address books provide predictive search where results show as you type, and often also fuzzy matching, allowing for partially misspelled names. A search engine like the one I wanted to build would never become popular if it did not provide the same. I therefore decided from the beginning that I wanted to include fuzzy searches which had been proposed as a project extension.

2.2 Underlying theory

Distributed Hash Tables are key-value stores that store data under particular keys and, given a key, will return the data stored under it if it exists. What makes Distributed Hash Tables different from regular hash tables, or dictionaries as they are called in some languages, is that data is not stored on a single machine, but split up across a network of machines.

There are quite a few well known Distributed Hash Tables. After initial research I decided to implement and compare Chord, Pastry and Kademilia. I soon realized Kademilia differs from Chord in ways similar to what Pastry already does, and decided to drop it from my implementation. This allowed the time for a more thorough evaluation of the relative performance of Chord and Pastry.

Distributed Hash Tables are key-value stores. In other words, unless you know the specific key under which a data item is stored, you will not be able to find it by other means than by doing a linear search through the system.

Having established that I wanted to support predictive search, as well as fuzzy searches, where you are allowed to misspell the name of the person you are trying to find, I had to come up with a mapping scheme that allowed me to predict correct keys with insufficient data. Schemes for looking up keywords in inverse indexes through flooding and random walks on top of structured and unstructured peer to peer networks have been proposed [5], but do not give performance in the sub-second latency range that I require in order to provide predictive search.

I came up with the following solution:

Just like in regular address books, I store records containing all the information a user wants to disclose to the network. These *profile records*, stored under a key created by hashing the whole profile record data structure, would include the user's full name, a link to the user's profile page hosted by his or her distributed online social network provider and preferably the address of a profile image that could be displayed alongside the search results.

In order to to find a profile record given incomplete or misspelled names, I also store *link records*. A link record contains the user's full name and the key of the full profile record, but are stored under keys generated by hashing fragments of a user's name.

I decided to generate link records separately for each of a user's names, n addition to generating a link record for each additional three characters in a user's name. A person with the name *Abcdefghijk* would get a link record for the first three characters of his name, *abc*, a link record created by adding the next the characters, *abcdef*, another link record adding the subsequent three characters, *abcdefghi*, and finally a link record for the whole name, *abcdefghijk*. As can be seen in figure 2.1, my surname *Eide* would result in link records for *eid* and *eide* while my first name *Sebastian* would result in link records for *seb*, *sebast* and *sebastian*.

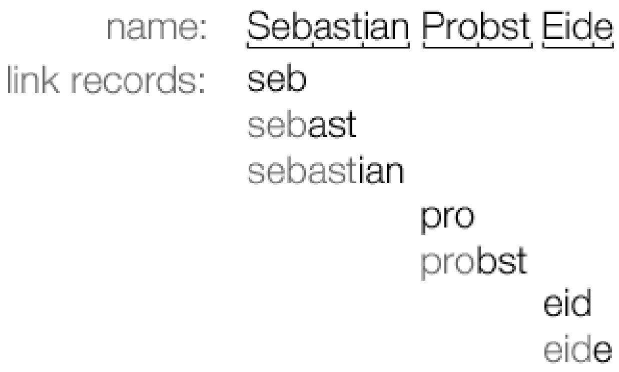


Figure 2.1: Names are split into name fragments used to generate keys for link records. The slightly darker coloured parts of the link record names indicate the additional three characters added compared to the previous link record name fragment.

When searching for a person the search system looks up *link records* while the user types. Since *link records* contain the full name of the person whose *profile record* they point to, the search system can try to find out which of the possible *profile records* the user is looking for by looking at the relative frequency at which the different *profile records* are being linked to by the *link records* downloaded. It can then download the highest ranked *profile records* and display them as search results to the user while he or she is still completing the search query.

There is an overhead both in terms of storage and lookup cost associated with this approach that I classify and evaluate in the evaluation chapter.

2.3 Language and tools used

Wanting to build a highly fault tolerant and distributed system with soft real-time requirements, I decided to use the programming language Erlang.

Erlang, while a language I had never previously used, was created exactly with these requirements in mind. Its standard libraries include frameworks for building hierarchies of process supervisors that ensure processes are restarted if they terminate prematurely or become unresponsive. Erlang also abstracts away inter process communication and encourages immutable state, making problems classical programming languages have with efficient concurrency and shared memory into non-issues.

I learned Erlang and its standard libraries before starting the project, and then selectively read up on more advanced topics as I needed them.

For creating the HTTP based API I used Webmachine, an Erlang framework designed for this purpose. I also used a build tool called rebar.

Testing was done using the built in unit testing framework EUnit, and erlymock was used for testing functions with side effects.

2.4 Software engineering practises and planning

The first basic implementation of my project followed a waterfall approach. Since I spent time up front clarifying what I wanted my product to do, and had specified the interfaces the components of my system should expose, this allowed me to quickly get a basic version of my system up and running.

Following the initial development cycle, I changed into short iterative bursts where I would adapt and extend the project as I found shortcomings or realized weaknesses in my initial design.

The implementation of the Distributed Hash Tables allowed for a very structured approach. The theory is well understood allowing me to write specifications in the form of tests based on the examples given in the Chord [3] and Pastry [4] papers to ensure my implementations behaved as expected.

I applied test driver practises where I write tests before writing any code for all major parts of the system.

This chapter analysed what is required for my project to be successful, why an HTTP based API is provided for integration with existing distributed online social networks, and how I proposed a scheme for searching across a key-value store using *link records*.

We also looked at why Erlang was chosen as the language of implementation and also how software engineering practises and planing played a role in the successful implementation of the project.

3. Implementation

I will first describe my software in terms of its main components before going into detail on how Chord and Pastry find the nodes responsible for keys, and how this was implemented in Erlang. Finally, I discuss Erlang's functionality for process supervision and hot code reloading and how these features were used in this project.

3.1 Process

The goals and milestones I set in my project proposal were expanded upon in the pre-development analysis which guided me through the implementation phase. Whenever bugs were found during the development and testing of the system, I reproduced the bugs through failing tests before fixing them. This process added a suite of regression tests to the existing set of unit tests written during development.

The Distributed Hash Tables Chord and Pastry are central parts of my design and also where the majority of my development time was spent. Therefore it is useful to discuss how they work and their implementation in this chapter.

3.2 High level system components

It is important that ones understands the difference between a host and a node in terms of this project. A host is defined to be a physical machine with a publicly accessible IP-address. Each host runs a single instance of the application, but each application instance can itself contain several Distributed Hash Table nodes running independently of each other. I can therefore scale the number of nodes in the my Distributed Hash Table networks, and by extension the search engine, by keeping the number of hosts, in other words machines, fixed, and just change how many Distributed Hash Table nodes each physical machine runs.

My project consists of two separate applications: the search server using the Distributed Hash Tables I developed, and a central control hub coordinating the search server instances. The central control hub also acts as a system-wide dashboard giving an overview of the number of machines participating in the search network in addition to how many Distributed Hash Table nodes are run by each machine and if they are Chord or Pastry nodes. The central control hub is also the application coordinating experiments across the network of search server nodes. As the central control hub plays more of an ancillary part in the project, its implementation will not be discussed further.

Figure 3.1 shows an overview of the components in the search server application running on the machines that are part of the search network. With small modifications in the controller component such as removing parts that are specific to the experimental setup and adding security features to the Distributed Hash Tables, the same setup could be used in a final release of the software.

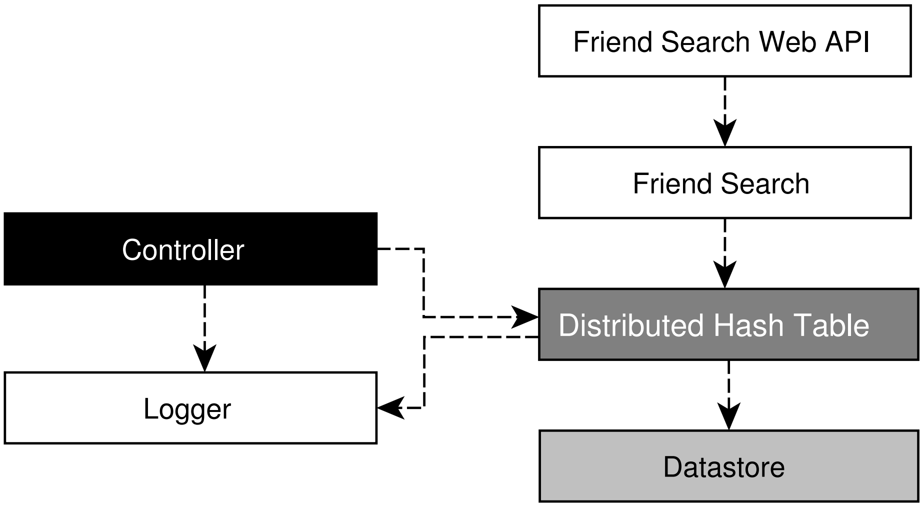


Figure 3.1: High level overview of the components of the search application. The arrows show the part a component relies on.

The arrows in figure 3.1 show which parts a component relies on to provide services for it. The Friend Search Web API exposes the HTTP API to the world and in turn uses the Friend Search server for resolving search queries into link records and then profile records that can be displayed to the end user. The Friend Search server uses the Distributed Hash Tables for finding the records, and the Distributed Hash Tables have local data stores that store the data items the particular Distributed Hash Table nodes are responsible for.

The controller is the component that interacts directly with the central control hub and starts and stops Distributed Hash Table nodes. It also regulates whether Pastry or Chord nodes should be run.

During experimental runs, the controller is also the component that issues requests to the locally run Distributed Hash Tables.

While a logger would certainly be included in the final release of the search engine, the one currently implemented would not be included as it is specific to the experimental setup.

3.2.1 Third party code

The Friend Search Web API in figure 3.1 uses an Erlang library called Webmachine for correctly handling HTTP requests. Webmachine in turn relies on another Erlang library called Mochiweb which the Friend Search Web API also uses in order to serialize the search results into JSON for consumption by the clients.

3.3 Distributed Hash Tables

Distributed Hash Tables are nothing but key-value stores where the data is stored across an array of machines — yet I am implementing two of them. The reason is that Chord and Pastry differ quite significantly in the way they approach finding out which nodes data-items are stored. Not only do they differ in the way they organize and store their routing information, but unlike Chord, Pastry uses a proximity heuristic to favour nodes geographically closer to itself when routing. These differences in routing directly translate into differences in performance as is discussed in the evaluation chapter.

My implementation used a bag approach where I allow multiple items to be stored under the same key. This is necessary to avoid link records with common keys replacing each other.

The Chord key-space is an integer key-space ranging from 0 up to $2^{160} - 1$. It wraps around such that $2^{160} - 1$ immediately precedes 0. It can be useful to think of the key-space as circular. Just like the Chord key-space the Pastry key-space is also an integer key-space. While the Pastry paper [4] suggests using keys in the range of 0 up to $2^{128} - 1$, I decided to let the Pastry key-space range from 0 to $2^{160} - 1$ just like for Chord, to increase code reuse between my Chord and Pastry implementations. The Pastry paper suggests using a 128-bit key space since they are easier to store efficiently in memory on 64-bit machines, which are common today, but other than that there is no evidence in the Pastry paper [4] indicating that the performance of the Pastry algorithm should be any worse using 160-bit rather than 128-bit keys.

All data in Chord and Pastry is stored under 160-bit keys. The Chord and Pastry nodes themselves are also associated with keys in the same key-space as the data items. I use a hash of the machines public IP-address and the port number a given node is listening to in order to determine a Distributed Hash Table node's key.

A node's key plays a significant role in splitting up the key-space between the nodes in the Distributed Hash Table. For this reason it is important that the keys are evenly distributed in the key-space so the amount of key-space each node is responsible for is roughly the same.

A Chord node is responsible for all the keys greater than the key of the node's predecessor and smaller or equal than the node's own key. In figure 3.2, the green node is responsible for the part of the key-space circle that is coloured green, the node coloured red for the part of the key-space coloured red, and likewise the yellow nodes for the part of the key-space that is yellow in colour. A Pastry node, on the other hand, is responsible for all key-values that are numerically closer to the node's own key than to the key of the nodes on either side of it in the key-space. In figure 3.3, you can see how the green and the red node are each responsible for half the key-space between them. Likewise, the red and the yellow node are each responsible for half of the key-space between their respective key-values.

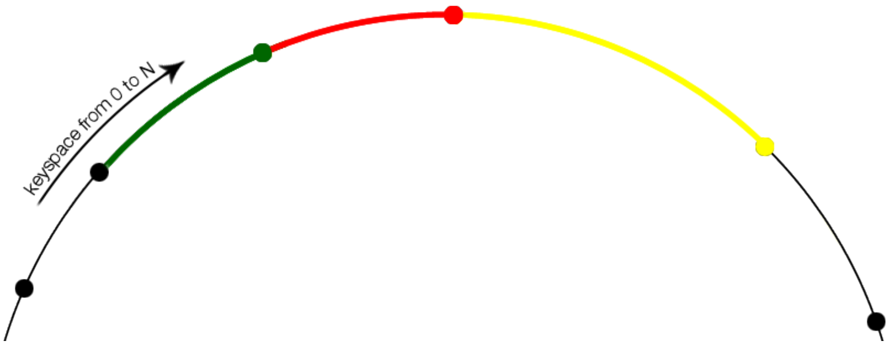


Figure 3.2: Illustration of how the keyspace is divided between Chord nodes.

Just like in regular hash tables, the performance is best if a good hashing function is used to generate keys so that the keys are uniformly distributed in the key-space. This minimizes collisions but, in the case of Distributed Hash Tables, also ensures that the nodes in the network store approximately equal amount of

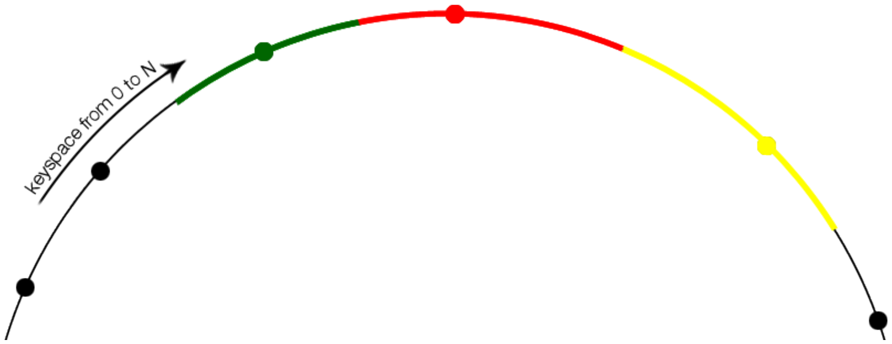


Figure 3.3: Illustration of how the keyspace is divided between Pastry nodes.

data. The traffic distribution in the network depends on the distribution of the keys being requested. This might not be as uniform, but if the keys requested are spread out in the key-space, the traffic, much like the data, will also be spread out between the nodes in the network.

In order to be able to use Chord or Pastry interchangeably I specified a minimalistic API they should implement. It has functions for setting and getting values and for starting and stopping Distributed Hash Table nodes.

3.3.1 Data structures

To better understand how the routing algorithms work, let us first look at the data structures used by Chord and Pastry for storing the routing information.

Chord's routing state

The Chord routing table contains an array of what, in chord jargon, are called fingers. Each finger contains a start value which depends on the finger's position in the array and the node's own id, as well as a reference to the first node in the Chord network with a key greater than that start value. Additionally, each finger contains an interval consisting of its own start value and the start value of the next finger. The node stored in the 0^{th} entry of the finger array is the node's successor node, the node in the subsequent entry represents a node slightly further away, and the entries following after that likewise contain nodes with increasing key-space distance. The routing table stores a total of 160 entries, one for each bit in the key.

Table 3.1 shows the Chord routing table and how each finger entry is calculated. It is taken from the Chord paper [3]. The values n, m and k are the node key, number of bits used for the key and index into the finger table array, respectively. The k^{th} finger entry has a start value that is half of the key-space greater than the node's own key-value. The $k-1^{th}$ entry a node quarter of the key-space further away, and so on until the nodes immediately preceding the node are found. This construction allows nodes to quickly close in on any given key-value by contacting another node which itself is at most half the key-space away from the key-value. In this implementation m is always 160. Note that the array starts at 1 and not 0.

Table 3.1: Values in the Chord routing table for a node with key n, using m-bit identifiers. k is an arbitrary array index into the finger array.

Notation	Definition
finger[k].start	$\left(n+2^{k-1}\right) mod\ 2^m, 1 \leq k \leq m$
.interval	$[finger[k].start, finger[k+1].start)$
.node	first node with key $\geq n.finger[k].start$
predecessor	the previous node in the key-space

Pastry's routing state

How does Pastry's routing table differ from that used by Chord? A Pastry node maintains three separate data structures: a tuple called the leaf set containing a list of the nodes preceding and a list of nodes succeeding it in the key-space, a list of nodes called the neighbourhood set that are it's neighbours in terms of the proximity heuristic used by Pastry, and a routing table. To make sense of the routing table let us first realize that while Chord looks at the key-space as an integer key-space in base 10, a Pastry node looks at the key-space as an integer key-space in an arbitrary base b. The routing table is a list of lists of nodes that share increasing number of digits in the key with the node itself. The first level entry in the routing table stores a list of nodes whose keys share no digits with the node. The second level entry stores nodes that share the first digit, and so on up until the last level entry where nodes share all but the very last digit. In a sparsely populated network, most of the entries in the routing table will be left blank. On the n^{th} level of the routing table, a node stores a list of nodes with keys that do not differ in any of the most significant digits up until the n^{th} key-digit. A node will try to store nodes for all possible values of the n^{th} digit, so ideally a node that itself has an n^{th} digit of 0 will have entries for nodes with values 1 through b - 1 in their n^{th} digit key positions.

Appendix A (page 56) shows how I translated the Chord and Pastry routing state into Erlang records.

Example routing state

The Chord and Pastry routing states can get quite abstract without a concrete example. In figure 3.4, I show a reference 8-bit key-space with nodes drawn as dots. The green dot represents the node whose routing state we will inspect. Each node has two keys. The top one is the key as seen by Chord, and the bottom key is the key as it is seen by Pastry which, in this case, interprets the keys as numbers in base 4.

Table 3.2 shows the routing state held by the Chord node with key 99, while table 3.3 shows the corresponding routing state for the same dot but as a Pastry node with key 1203.

3.3.2 Routing

The routing algorithms for both Chord and Pastry are reasonably compact. I will give you an example of how the routing works before showing the pseudo-code from the Chord [3] and Pastry [4] papers. Then, I show show the corresponding Erlang implementations.

I have omitted supporting functions from the Erlang sample code that do not add to the understanding of the routing procedure in order to keep the source code listings concise.

Routing in Chord

We now review what happens when a Chord node wants to get the value of a key. Let us call the node that wants the value the requester, and the node that is responsible for the key the target node. I will use figure 3.5 to illustrate the procedure. In the figure, the nodes with the names A, B, D and E are the nodes that directly participate in finding the target node F. When the requester wants to get the value of a key, it starts by finding the node in its own routing table with a key that most closely precedes the key it wants to get the value of. In our example this is node B. It then asks that particular node if it knows about another node more closely preceding the target key. In this case node B knows about node D which has a key closer to the target key than what B itself does. The requester, node A, repeats the same question to node D and is returned node

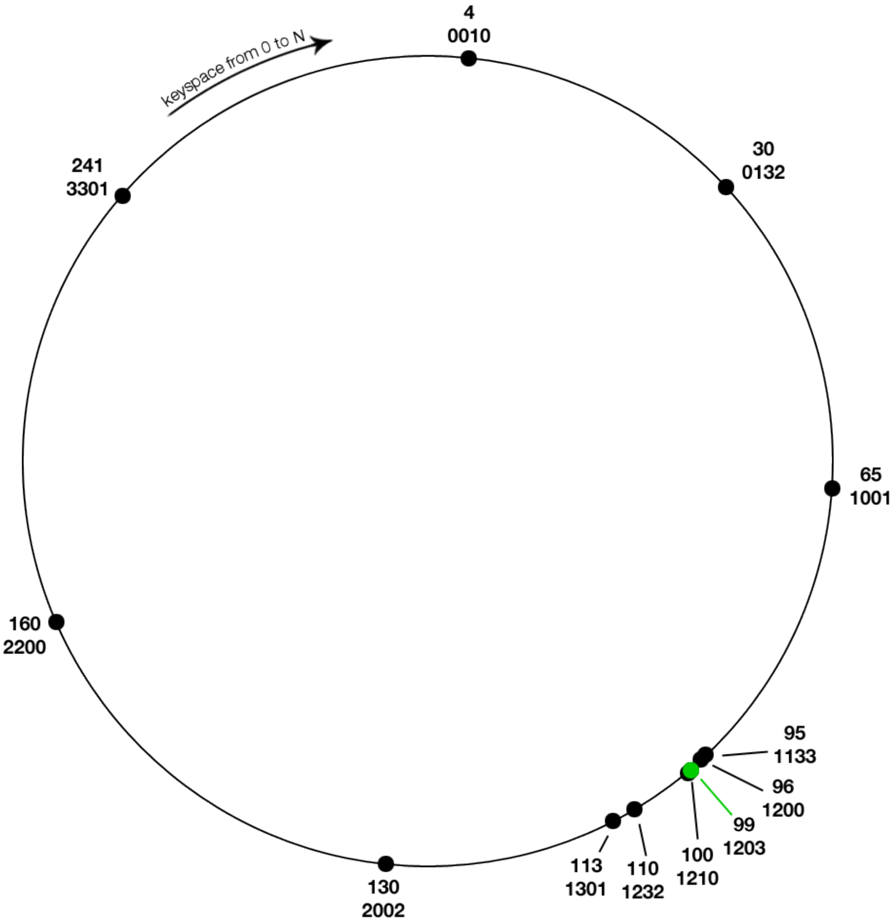


Figure 3.4: Key-space with nodes. Each node has two keys. The top one is the key represented in base 10 as Chord sees it, and the bottom one in base 4 as seen by Pastry in this example.

E. When node E is asked for the closest preceding node it returns itself, and it's successor F is asked for the contents of the key.

Notice how the Chord node A, the requester, is involved in all steps of the process of looking up which node stores the value of the key. This will become important in the evaluation section where we see what happens when we cannot connect to a node.

In the following code listings I will use the following conventions: The syntax `n.method_name()` indicates that the method with name `method_name` is executed *on* node `n`. So when `n'.method_name` is called on node `n`, a remote

Table 3.2: Chord routing table for nodes in figure 3.4

Field	Value
finger[1].start	$99 + 2^0 = 99$
.interval	$[99, 101)$
.node	100
finger[2].start	$99 + 2^1 = 101$
.interval	$[101, 103)$
.node	110
finger[3].start	$99 + 2^2 = 103$
.interval	$[103, 107)$
.node	110
finger[4].start	$99 + 2^3 = 107$
.interval	$[107, 115)$
.node	110
finger[5].start	$99 + 2^4 = 115$
.interval	$[115, 131)$
.node	130
finger[6].start	$99 + 2^5 = 131$
.interval	$[131, 163)$
.node	160
finger[7].start	$99 + 2^6 = 163$
.interval	$[163, 227)$
.node	241
finger[8].start	$99 + 2^7 = 227$
.interval	$[227, 99)$
.node	241
predecessor	96

procedure call to node `n'` is being made.

Code listing 3.1 shows the routing algorithm used by Chord as pseudo code. On line 10 in the code listing we see the method `closest_preceding_finger` being invoked on node `n'`. Using the routing example in figure 3.5 `n'` starts out as A, then changes to B etc.

Listing 3.1: Chord routing pseudo code

```
1 // ask node with key n to find key k's successor
```

Table 3.3: The Pastry routing state from the Pastry nodes in figure 3.4. The bold digits in the routing table indicate at which digit the node keys differ from the node's own key.

Leaf set		
Greater:	1210	1232
Smaller:	1200	1133

Neighbourhood set			
0010	1301	3301	1001

Routing table			
none	**0**010	**2**200	**3**301
1st	1**0**01	1**1**33	1**3**01
2nd	12**1**0	12**3**2	
3rd	120**0**		

```
 2 n.find_successor(k)
 3   n' = find_predecessor(k);
 4   return n'.successor;
 5
 6 // ask node with key n to find key k's predecessor
 7 n.find_predecessor(k)
 8   n' = n;
 9   while (k not in (n', n'.successor])
10     n' = n'.closest_preceding_finger(k);
11   return n';
12
13 // return closest finger preceding key k
14 n.closest_preceding_finger(k)
15   for i = m downto 1
16     if (finger[i].node in (n, key))
17       return finger[i].node;
18   return n;
```

Let us now take a quick look at each of the three methods from listing 3.1 in turn and compare them to their Erlang equivalents.

The `find_successor` method takes a key and finds its predecessor before returning the predecessor's successor. Listing 3.2 shows my Erlang implementation. We see there is a very clear correspondence with the pseudo code implementation. First, the predecessor node is found on line 7 and it's successor then returned on line 9. What complicates the Erlang code slightly is that I wanted to implement the `find_predecessor` while-loop using recursion. I therefore had to move the assignment on line 8 in the Chord pseudo code out to the `find_successor` function.

Listing 3.2: Finding the successor of a key

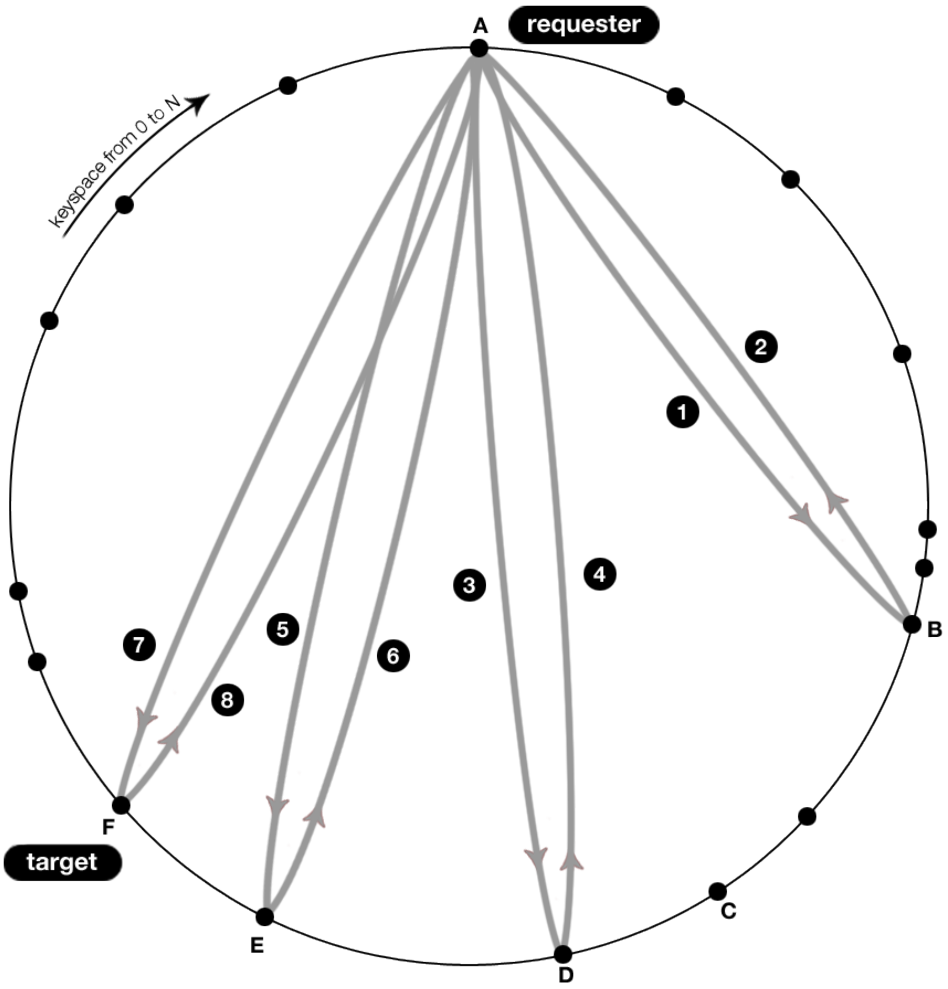


Figure 3.5: Chord's approach to routing.

```
perform_find_successor(Key, #chord_state{self = Self} = State) ->
  % in the case we are the only party in a chord circle,
  % then we are also our own successors.
  case closest_preceding_finger(Key, State) of
    Self -> {ok, Self};
    OtherNode ->
      case perform_find_predecessor(Key, OtherNode) of
        error -> error;
        Predecessor-> {ok, Predecessor#pred_info.successor}
      end
  end.
```

In code listing 3.3, I show the Erlang equivalent of the Chord `find_predecessor` pseudo code. In the pseudo code there are two remote procedure calls being done. One to get a node's successor, and only conditionally one requesting the node's closest preceding finger. In my implementation these two remote procedure calls have been made into one and for this reason this call has been moved to the very front of the function. Notice how on line 11 in the Erlang implementation we use recursion rather than the while loop used in the pseudo code. The equivalent of line 8 in the Chord pseudo code had to be moved out to the `find_successor` function.

Listing 3.3: Finding the predecessor of a key

```
 1 perform_find_predecessor(Key, Node) ->
 2   case chord_tcp:rpc_get_closest_preceding_finger_and_succ(Key, Node) of
 3     {ok, {NextClosest, NodeSuccessor}} ->
 4       case utilities:in_right_inclusive_range(Key, Node#node.key, NodeSuccessor#
             node.key) of
 5         true ->
 6           #pred_info{
 7             node = Node,
 8             successor = NodeSuccessor
 9           };
10         false ->
11           perform_find_predecessor(Key, NextClosest)
12       end;
13     {error, _Reason} ->
14       error
15   end.
```

The procedure for finding the `closest_preceding_finger` follows below. Unfortunately it is slightly more complicated than the pseudo code equivalent. Since I for fault tolerance am storing a list of successors rather than a single successor, the case of the 0^{th} finger entry has to be handled separately. I factored out the bounds checking done at line 16 in the Chord pseudo code into a helper function called `check_closest_preceding_finger` to avoid having to duplicate it in the `closest_preceding_finger` function. This function in turn makes a call back to `closest_preceding_finger` if the finger entry is not a match. The function in listing 3.4 recursively looks at each finger in the routing table, checking if it is valid and returning the first one it finds that is.

Listing 3.4: Returns the node most closely preceding the key

```
% Returns the node in the current nodes finger table
% most closely preceding the key.
closest_preceding_finger(Key, #chord_state{chord_pid = Pid} = State) ->
  closest_preceding_finger(Key,
    State#chord_state.fingers, array:size(State#chord_state.fingers) - 1,
    State#chord_state.self).

closest_preceding_finger(_Key, _Fingers, -1, Self) -> Self;
closest_preceding_finger(Key, Fingers, 0, Self) ->
  % The current finger is the successor finger.
  % Get the closest successor from the list of successors
  FingerNode = get_first_successor(array:get(0, Fingers)),
  check_closest_preceding_finger(Key, FingerNode, Fingers, 0, Self);
closest_preceding_finger(Key, Fingers, FingerIndex, Self) ->
  FingerNode = (array:get(FingerIndex, Fingers))#finger_entry.node,
  check_closest_preceding_finger(Key, FingerNode, Fingers, FingerIndex, Self).
```

The bounds check is implemented in the helper function `check_closest_preceding_finger` in listing 3.5.

Listing 3.5: Check if a finger is the closest known finger to a key. Returns the node most closely preceding the key

```
% We check if the node for a given finger entry is between
% ourselves and the key we are looking for. Since we are
% looking through the finger table for nodes that are
% successively closer to ourself, we are bound to end up
% with the node most closely preceding the key that we know about.
check_closest_preceding_finger(Key, undefined, Fingers, FingerIndex, Self) ->
  % The finger entry is empty. We skip it
  closest_preceding_finger(Key, Fingers, FingerIndex-1, Self);
check_closest_preceding_finger(Key, Node, Fingers, FingerIndex, Self) ->
  case utilities:in_range(Node#node.key, Self#node.key, Key) of
    true ->
      % This is the key in our routing table that is closest
      % to the destination key. Return it.
      Node;
    false ->
      % The current finger is greater than the key. Try closer fingers
      closest_preceding_finger(Key, Fingers, FingerIndex-1, Self)
  end.
```

Routing in Pastry

The Pastry approach differs quite significantly from Chord's approach to routing. The most significant difference is that while a Chord node actively looks for the node responsible for a key, and when that node has been found contacts it to have it perform whatever task needs doing, Pastry instead performs message passing, where the message itself can contain information about whatever task the node wants done. Also just like when sending mail through the postal system, the node loses visibility of the messages progress as soon as it has been sent. This has the benefit of reducing the communication overhead and giving the sender time to perform other tasks or aid other nodes in their message sending while making it harder to notice if a message get lost.

I illustrate the process in figure 3.6 where the requester of a key-value, node A, sends a *lookup message* to the key it wants to look up. First node A checks its own routing state for the node it knows about with a key closer to the target key than its own and sends its message to it. In this case this node is node B. B in turn does the same and forwards the message to node D which in turn sends it on to the target node F. In the final step node F contacts the requester, node A, to let it know it is the node responsible for the value of the requested key.

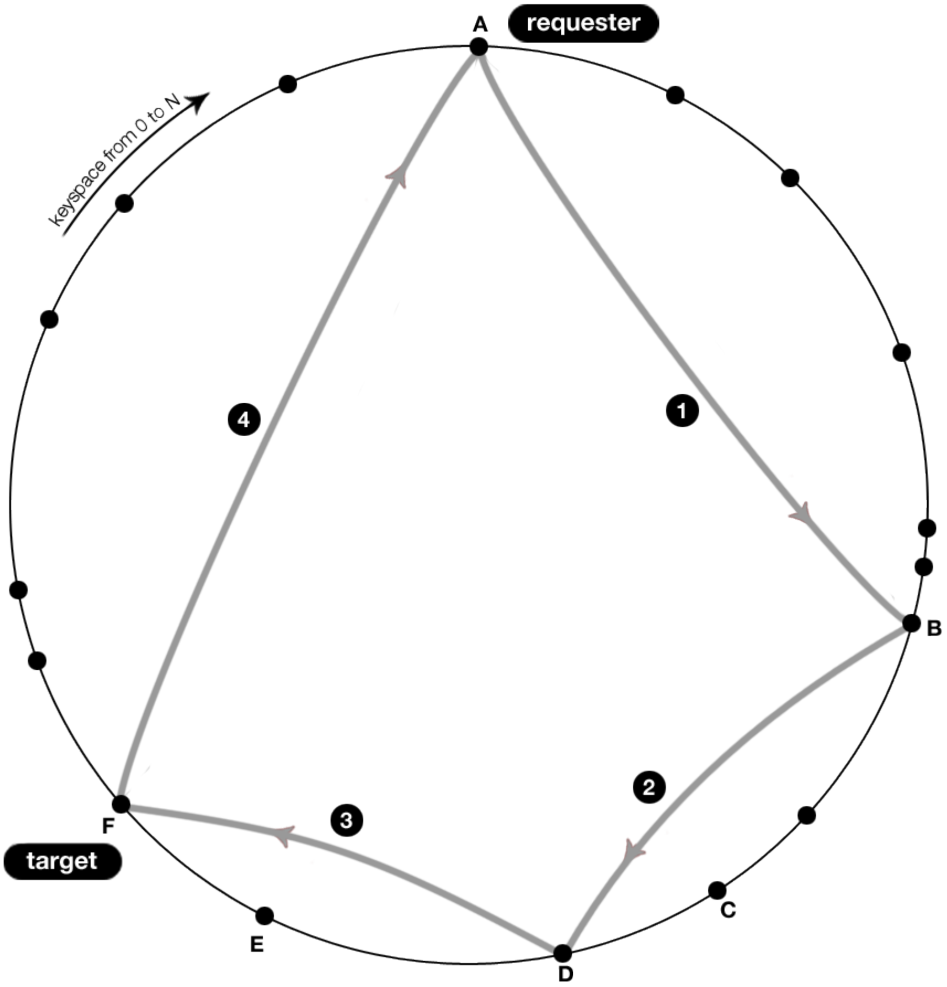


Figure 3.6: Pastry's approach to routing.

The following Pastry routing pseudo code is a slightly adapted version of the pseudo code presented in the Pastry paper [4]. It shows the method for routing a message for a target key k at a node with key n.

The Pastry routing approach works as follows: The node that is about to forward the message first checks if the recipient is in its leaf set. If it is it delivers the message directly. The recipient could also be itself. If there is no match in the leaf set, the routing table is used to find a node that shares an additional digit with the key with the message key compared to the node performing the routing.

If there is no match in the routing table either, the Pastry node falls back to forwarding the message to any node it knows about that has a key numerically closer to the message key than the key the node itself has.

Listing 3.6: Message routing pseudo code for Pastry

```
n.route_msg(msg, k)
  // Node n forwards a message to the node responsible for key k
  if (LeafSetSmaller =< k =< LeafSetGreater) {
    // k is within range of our leaf set
    forward the message to node in leaf set
        with key n’ such that |k - n’| is minimal
  } else {
    // use the routing table
    Let l = length_of_shared_key(k, n)
    if (there is a node n’ in level l of the routing table
        that shares it’s lth key digit with k) {
      forward the message to node n’
    } else {
      forward to any known node n’ that shares
          at least as many digits with k as n does
          but is numerically closer to k than n is.
    }
  }
}
```

The translation from Pastry pseudo code to Erlang implementation resembles the procedure used to implement the Chord algorithm in Erlang. It is described in more detail in Appendix B on page 58.

3.3.3 Replication of data

My implementations of Chord and Pastry both replicate data amongst their immediate neighbours for fault tolerance. While replication was only proposed as an extension in the Chord [3] and Pastry [4] papers, I thought it worthwhile adding this functionality since my intended use case was a search engine where I did not want data to become unavailable if an online social network provider decided to stop using my search network.

3.4 Search server

The search server is what glues together the client facing HTTP API and the Distributed Hash Tables. It receives the search queries from the clients, converts them into appropriate keys that it can lookup in the Distributed Hash Tables, and resolves link records it gets returned to full profile records that can be shown as search results.

The current implementation is rather naïve, and while it does basic ranking of results, prioritising what it predicts are better matches, it performs no caching of results which would significantly improve the performance.

3.5 Central control hub

The central control hub is an application that runs under a domain name known by all application instances of my search engine. When the search engine application is started on a machine, it registers itself with the control hub which in turn tells it what Distributed Hash Table should be used and how many nodes and of what Distributed Hash Table type should be run.

When new Distributed Hash Table nodes are started they also contact the control hub to get information about which other nodes they should try to connect to in order to join the Distributed Hash Table network.

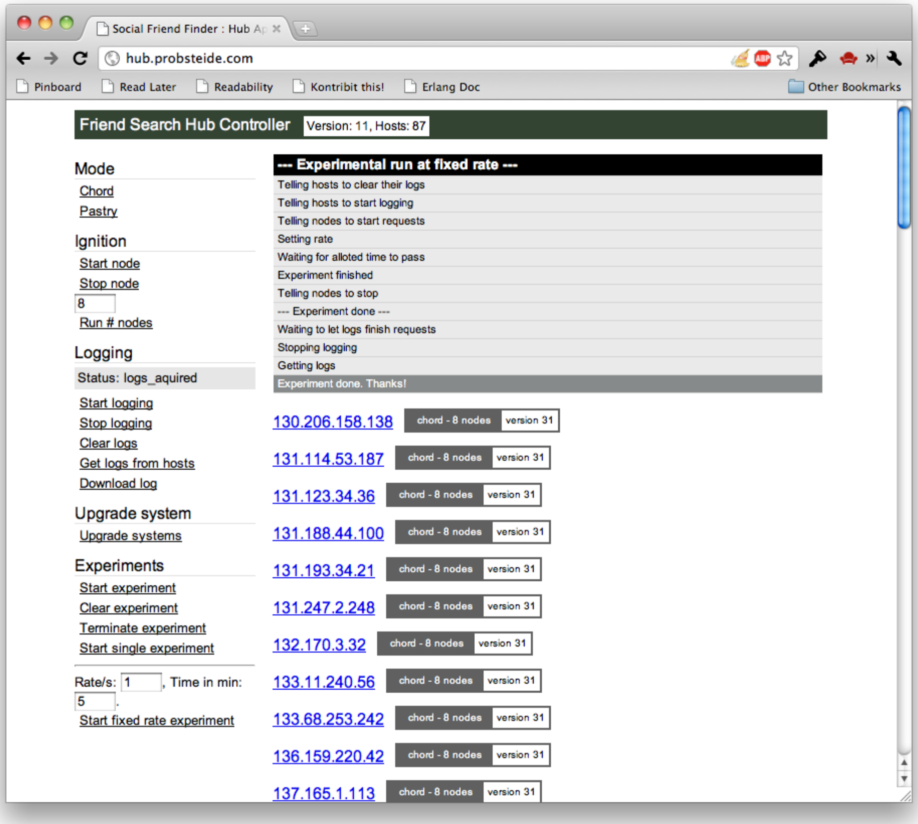
The control hub is the only single point of failure in the system. If it becomes unavailable, no new search engine hosts can join because they cannot register and get information about which Distributed Hash Table type is being used, nor can the existing hosts start new Distributed Hash Table nodes because the nodes cannot contact the central control hub to get information about nodes they could use to join the Distributed Hash Table network. All hosts that are already registered remain active, and once a Distributed Hash Table node has started it runs fully independently of the central control hub application. One could allow hosts to start new nodes by directly telling them about other known hosts they can connect to, but that functionality is currently not implemented.

The experiments used to evaluate the system are also initiated by the central control hub through a web interface (figure 3.7) that allows me to set the rate at which requests should be issued and the duration of the experiment. The requests themselves are issued individually by each host running the Distributed Hash Table nodes.

Initially fully automated experiments were supported as outlined in my project briefing, but later, while evaluating my system, I removed the feature as it was impractical and didn't give me enough flexibility to run the experiments I needed.

3.6 Supervision for fault tolerance

One of Erlang's main strengths is process supervision. The supervision behaviour which is a part of the standard library makes it trivial to write processes super-



Version: 11, Hosts: 87

8

Status: logs_aquired

Rate/s: 1, Time in min: 5.

```
--- Experimental run at fixed rate ---
Telling hosts to clear their logs
Telling hosts to start logging
Telling nodes to start requests
Setting rate
Waiting for alloted time to pass
Experiment finished
Telling nodes to stop
--- Experiment done ---
Waiting to let logs finish requests
Stopping logging
Getting logs
Experiment done. Thanks!
```

```
130.206.158.138  chord - 8 nodes  version 31
131.114.53.187  chord - 8 nodes  version 31
131.123.34.36  chord - 8 nodes  version 31
131.188.44.100  chord - 8 nodes  version 31
131.193.34.21  chord - 8 nodes  version 31
131.247.2.248  chord - 8 nodes  version 31
132.170.3.32  chord - 8 nodes  version 31
133.11.240.56  chord - 8 nodes  version 31
133.68.253.242  chord - 8 nodes  version 31
136.159.220.42  chord - 8 nodes  version 31
137.165.1.113  chord - 8 nodes  version 31
```

Figure 3.7: The Hub Application interface running on the central control hub. Shown here is the web interface that allows me to start and stop nodes, change between using Chord and Pastry, and start and stop experiments

vising others. These processes are responsible for the lifetime behaviour of the processes they supervise, starting them when needed, restarting them if they terminate prematurely and stopping them when the application terminates. All supervisors with the exception of the root supervisor are themselves also supervised.

Process supervision is an important feature as it encourages designing minimal and isolated modules with individual life cycle control that can fail independently without bringing down the system. As failing modules are automatically restarted without impact on other components, one can easily build systems with high availability. Another direct benefit also encouraged by Erlang and it's designers is to write optimistic code handling what you as a designer think of as the optimal case, rather than practising defensive coding where you constantly have to check for malformed inputs. This is possible as malformed input which would cause a process to crash does not affect the systems availability since the crashed process is automatically and immediately restarted. The direct benefit of this approach is that the code base becomes smaller and much easier to read and maintain.

Figure 3.8 shows the supervisors and the servers used in the main friend search application I developed.

The root supervisor, represented by the box in the middle of figure 3.8, supervises a wide range of other supervisors, all of which, with the exception of the Distributed Hash Table supervisors, supervise servers that are started at the same time as the application. The Chord and Pastry supervisors are a special kind of supervisor. They do not initially start any children, but can be signalled to start and maintain an arbitrary amount of children. This feature allows my system to at runtime dynamically adjust the number of Distributed Hash Table nodes that are run.

3.7 Hot code reloading

The Erlang virtual machine supports hot code reloading. I built a code distribution service on top of this functionality, that automatically updates the binary code of all running nodes whenever I pushed code changes to my code repository.

This greatly reduced the time it took to change small bugs and experiment with changes after I had deployed my code to a large network of machines.

In this chapter we have looked at how I developed my system. We looked at high level components and went into details of how routing differs between Chord and Pastry. We then looked at the other central components of my project, the central controlling hub and the search server. Finally we had a brief look at Erlang's supervisors and hot code reloading.

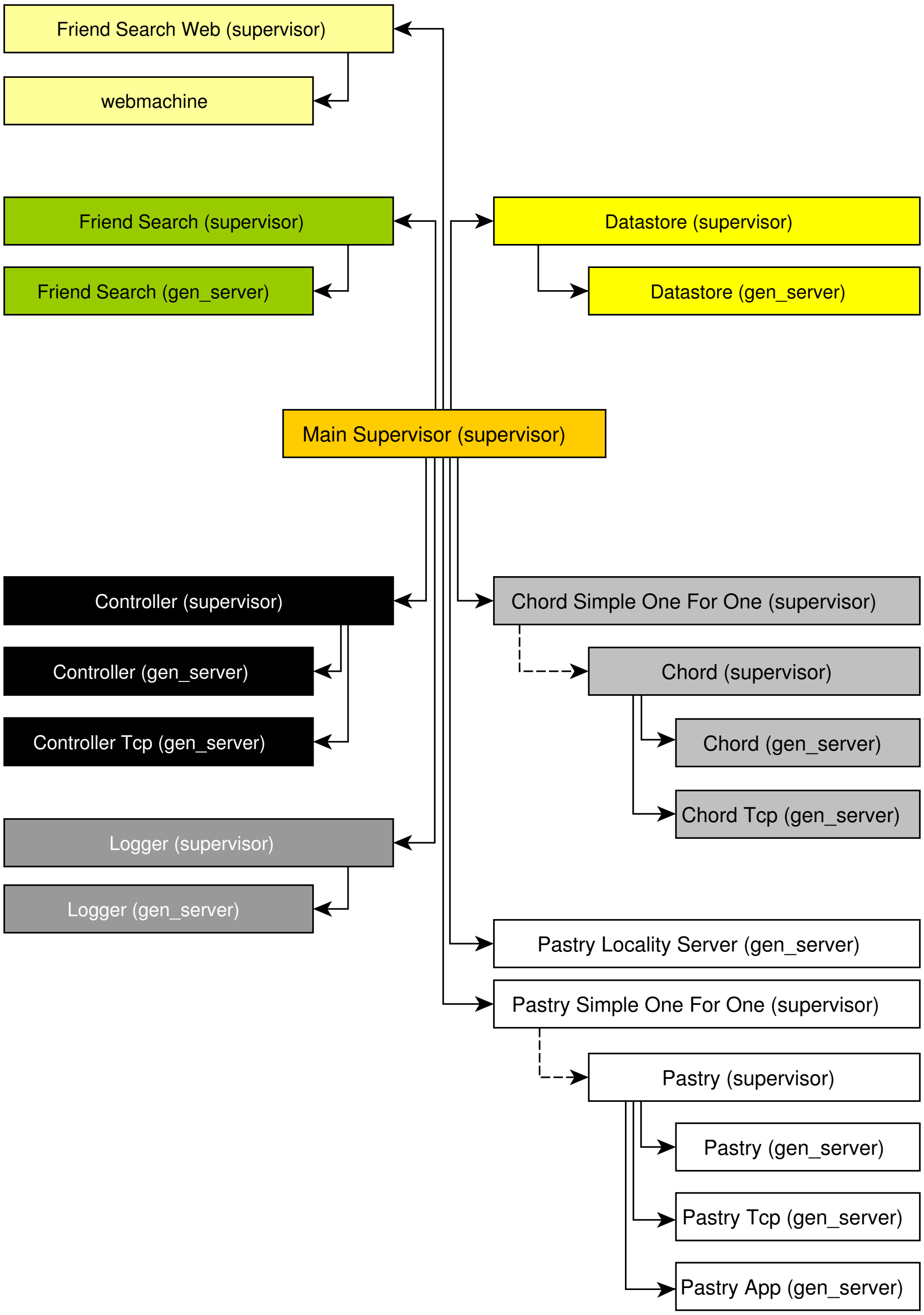


Figure 3.8: This figure shows the hierarchy of supervisors and the servers they supervise.

4. Evaluation

In this chapter I will look at if Distributed Hash Table are suitable back end data stores for distributed search engines of the very specific kind I set out to build. We will furthermore look at how the performance of Chord and Pastry compares in terms of latency and also briefly discuss the benefits of the proximity heuristic used in routing by Pastry.

4.1 What to evaluate

My main concern in this evaluation is if Distributed Hash Tables can be used as data stores for my distributed search engine. The determining factor will be the latency between issuing a key lookup request and receiving its value. Therefore the only dependent variable I will look at in depth is latency. The latency directly affects the response time as it is seen by the end users using the search engine and also directly affects to what extent predictive searches are possible or not as there are latency requirements associated with something being perceived as real-time. Latency also affects if my proposed *link record* scheme can be used or not, and by extension if predictive and fuzzy searches on top of Distributed Hash Tables can be done since link records in my implementation is a prerequisite for that to be possible. I also want to evaluate the success rate of requests as failed requests would need to be repeated, and therefore directly affect the latency. It is also interesting to look at how the performance, again in terms of latency, is affected by the number of nodes in the Distributed Hash Table. In evaluating if Distributed Hash Tables are viable data stores for my search engine it is also interesting to compare the performance of Chord and Pastry to see if one is a better choice than the other.

I will not look at the overhead of maintaining routing state as it is negligible for a well tuned Distributed Hash Table [5] and does not affect my dependent variable.

4.2 Experimental design

I ran my Distributed Hash Tables on machines in the Planet-Lab network. With that in mind, the variables I was able to control were the number of machines participating in the system, the number of Distributed Hash Table nodes running on each machine, and the rate of requests in the network.

In my tests I used all the machines available to me to give some scale to the experiments, and limited the experimental factors to the number of nodes run on each machine and the rate at which requests were issued.

My experiments follow a factorial design where both node count and rate varied between 1 and 16 in powers of two. Since the theory behind Distributed Hash Tables indicate that the number of routing hops is proportional to the logarithm of the number of nodes in the system, and the number of hops directly affect our dependent variable latency, increasing the number of hops in powers of two is reasonable. Doing the same for the rate allows us higher resolution into the lower rates while still giving us data from a wide range of rates to evaluate. Each configuration was run at three different occasions for both Chord and Pastry, and the log files for all three runs with the same configurations were then used to give the results I am about to present.

4.2.1 Experimental pattern

Each experimental run followed a predefined pattern:

1. I would ensure that the correct number of nodes was running and allow 3 minutes for routing state to adjusted itself if the node count was not correct. 3 minutes was chosen as it gave me high confidence that the routing state would be up-to-date, given the high rate at which the nodes were configured to update their routing state.

2. Each machine would clear old logs and start logging.

3. The experiment would start and each machine in the distributed search engine, each hosting the same number of Distributed Hash Table nodes, would issue requests to its locally hosted nodes at a rate specified for that particular experimental run.

4. After a successfully completed run, there would be a cool down period with continued logging where requests taking slightly longer than permitted would be allowed to finish. Logging would then stop and the logs from the individual machines would be collected at the central hub node.

5. The logs were then downloaded from the central hub to my personal machine for safekeeping and analysis.

6. The process was then repeated for different experimental parameters.

4.3 Effects of Planet-Lab

As mentioned my experiments ran on top of machines provided by Planet-Lab. I was allowed to use up to 100 machines, but never managed to get quite that many up and running at the same time.

Using Planet-Lab as a test bed gave me several benefits: the machines at my disposal had a high geographical spread and varied greatly in what computational resources they provided. Depending on the time of day the computational load on the machines would also vary and not only that, but I would have a different subset of the machines available for testing from day to day. This provided an excellent foundation for testing Distributed Hash Tables which are meant to be able to cope with high node churn and machines with widely different capabilities and additionally abstracts away the specific topology used for the experiments.

While Planet-Lab provides an interesting test bed, the randomness with which computational resources are available makes it impossible to accurately repeat experiments. Rerunning an experiment would invariably give a slightly different result. To work with Planet-Lab's strengths and not against it, and at the same time get data I could have confidence in, I ran each experiment multiple times, shifted in time. This way I ensured I would sample different subsets of the topology. I believe this gave me results that while having higher variance, also more closely resemble the real world scenario Distributed Hash Tables are likely to find themselves in.

4.3.1 Experimental cleanliness and validity of results

Performing experiments over longer periods of time showed that the latency (figure 4.1) and success rate (figure 4.2) for all practical purposes stay constant over time. This allowed me to cut down the length of individual experimental runs without lowering the quality of my results. Figure 4.1 show the latencies of individual requests plotted against time for an experiment with 65 hosts running 1 Distributed Hash Table node each issued 8 requests per second. Figure 4.2 show the instantaneous rate of requests per second network wide against time and the success rate against time for the same experimental run used to generate the latency data in figure 4.1.

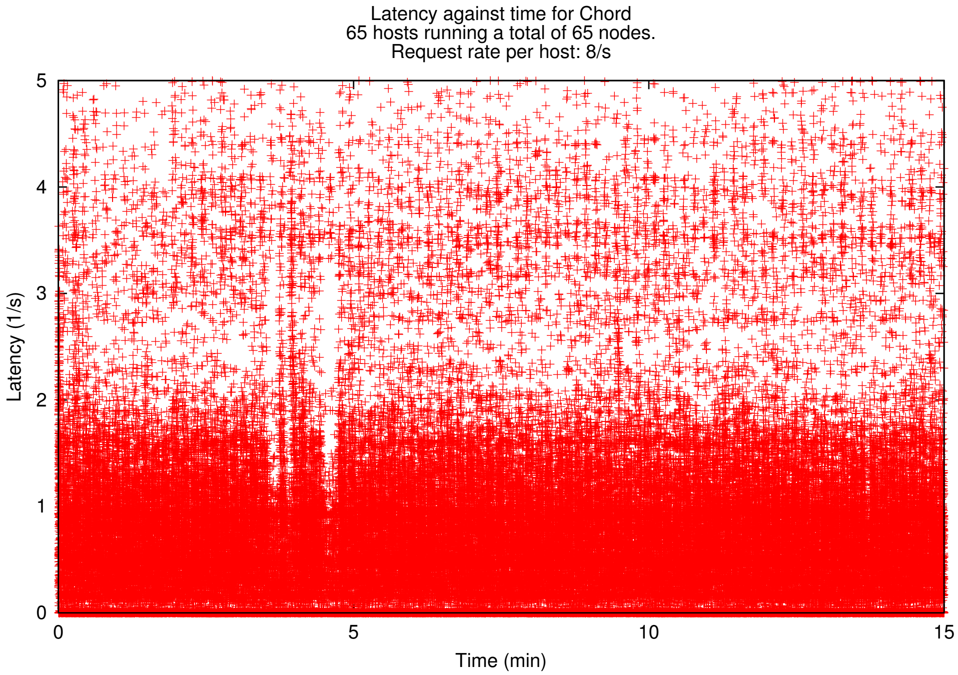


Figure 4.1: This graph shows how latency vary with time in a 15 minute experimental run of Chord.

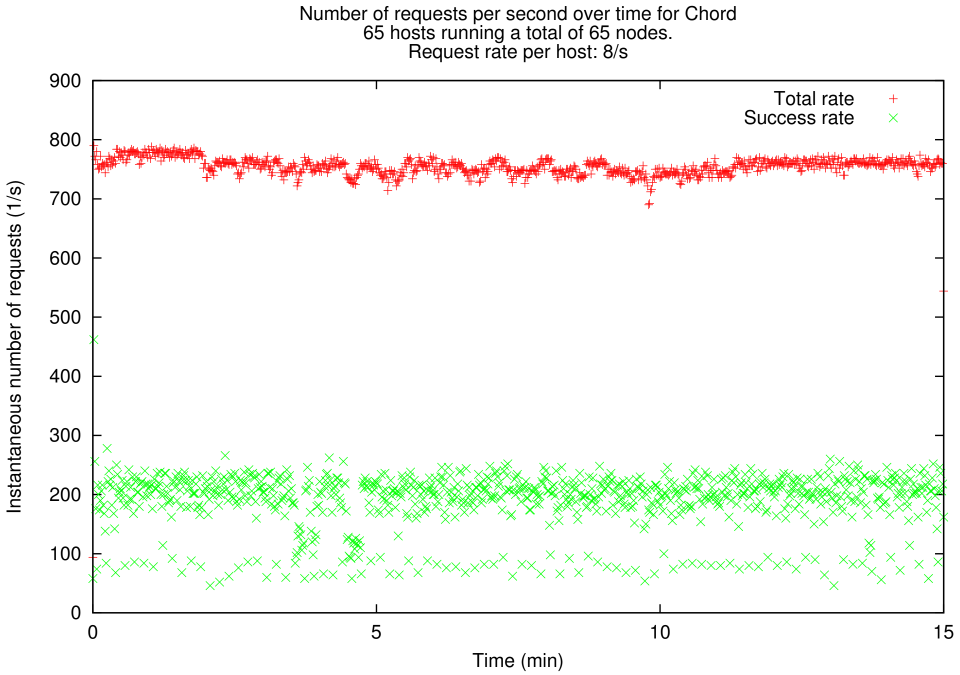


Figure 4.2: This graph shows how the request rate and success rate varies over a 15 minute experimental run of Chord.

During experimental runs I would turn off all functionality not directly needed for serving the requests. This would include turning off the mechanisms that kept the nodes routing state up to date. If a node disappeared during an experiment, nodes contacting it directly would take note of it, but other nodes wouldn't have the node's disappearance reflected in their routing state like they would under normal operation. This might seem a somewhat unnatural choice, but makes sense for my experiments for the following reasons: In my experimental builds of Chord and Pastry I had turned up the rate at which routing information was dissipated through the network to an extreme to cut down the waiting time between experimental runs. Since what I wanted to test was the latency of routing requests through the network rather than ancillaries like maintaining the routing state, removing the overhead of keeping the routing state up to date is the right thing to do. This overhead differs between Chord and Pastry, and depends on parameters that would have to be tweaked before using this system in any real world deployment, and therefore would unfavourably affect the test results.

To spread the load as evenly as possible across the network of nodes, each search engine host, hosting Distributed Hash Table nodes, would issue requests to the network through it's locally hosted nodes. The keys used in the requests were randomly generated by hashing a combination of the IP-address of the machine issuing the request and a time stamp. This approach gives keys with a nice spread, but results in keys that differ from experimental run to experimental run. While this makes doing repeatable experiments impossible, it evenly spreads the load between the machines.

Data from experimental runs where a significant number of machines disappeared during the run would be discarded.

4.4 Results

Throughout my experiments, a valid request is defined as a request that completes within 5 seconds. By that definition, all other requests are invalid, regardless of if they fail to complete at all, or if they just complete after more than 5 seconds have passed.

4.4.1 Mean latency and success rate for Chord and Pastry

In figure 4.3 and 4.4 you see the latency for different rates and different number of nodes per machine for around 60 machines, shown with 95% confidence intervals.

It is noticeable how the latencies for Chord (figure 4.3) are higher than they are for Pastry (figure 4.4), and also how there is less of a correlation between different experimental configurations and the resulting latencies. For Chord there seems to be a trend showing that the latencies for a fixed number of nodes per host decreases for higher request rates.

In the case of Pastry (figure 4.4) we see how the latencies consistently grow larger for increasing total numbers of nodes. Intuitively one would think this is as it should be as the number of nodes involved in any given key-lookup on average should increase proportionally to the logarithm of the number of nodes in the Distributed Hash Table network, but for this set-up with Pastry nodes it is somewhat counter intuitive. The number of hosts the nodes are run on is constant resulting in there being more nodes per host, which one would expect should let Pastry's proximity heuristic reduce rather than increase the latency as more nodes running on the same physical hosts allow some of the message passes being between nodes with close to no latency. Unfortunately these results, where performance decreases when more nodes are run on the same hardware, agree with the results I see when evaluating the performance improvements given by running multiple nodes on the same host and using the Pastry proximity heuristic (figure 4.13) presented later in this chapter.

What we seem to see in figure 4.4 is that as the number of nodes in the network as a whole grows, the average number of nodes involved in each requests also increases resulting in the inter-node connection latency playing a greater role in routing messages.

The success rate is also dramatically different between Chord (figure 4.6) and Pastry (4.5). Chord seems to perform better when more than one node is run on the same machine, while Pastry's success rate slightly decreases the more nodes are run on the same physical hardware. It is also worth noting that even at its best, Chord can't compare with the worst case performance of Pastry.

Unfortunately I have no good explanation for the variations we see within the different configurations for Chord, but I can explain why Chord in general performs so much worse than Pastry, and I will do so shortly.

Before that I want to discuss how Chord and Pastry behave when pushed to their limits. The cumulative distribution functions for Chord (figure 4.7) and Pastry (figure 4.8) show the fraction of requests succeeding within a certain time. The experiments are run on roughly 60 machines hosting 1 node each. For Chord the success rate is only marginally worse for 128 and 64 requests per second per hosts than when there are 16 requests per second per machine. Also rather odd is how the rate of 2 requests per second performs significantly worse than what does the rate of 4 in the case of Chord. I have no good explanations for this

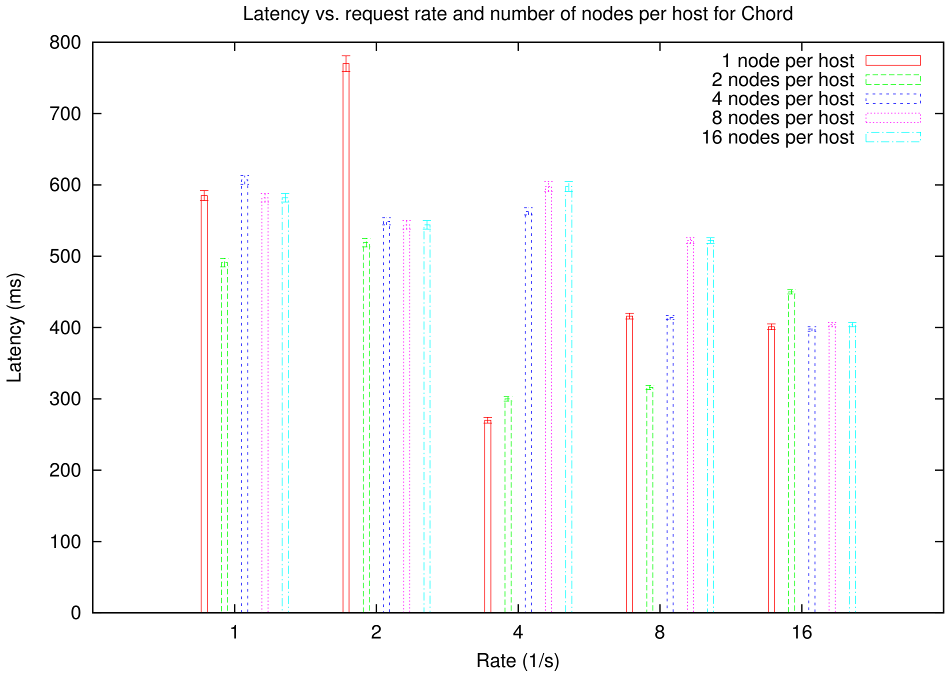


Figure 4.3: This graph shows latencies for Chord with 95% confidence intervals for setups of slightly above 60 nodes.

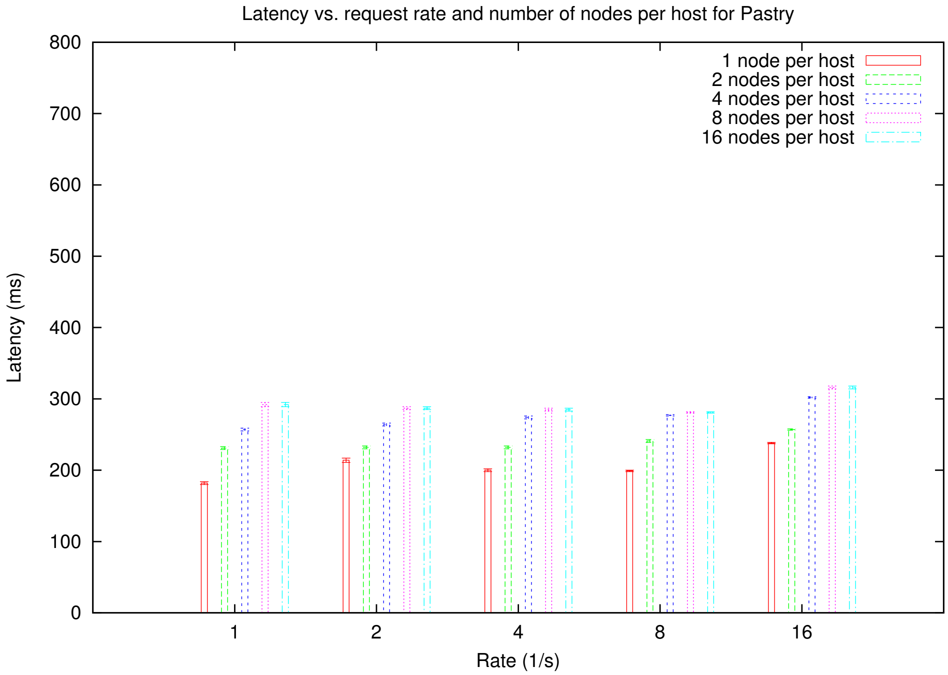


Figure 4.4: This graph shows latencies for Pastry with 95% confidence intervals for setups of slightly above 60 nodes.

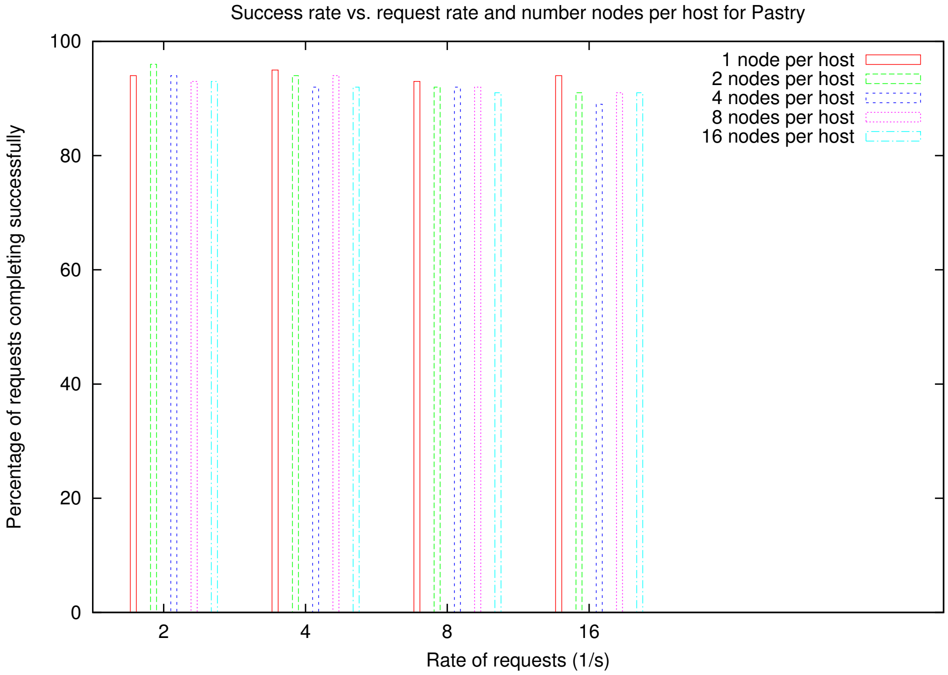


Figure 4.5: This graph shows the instantaneous rate of requests and the instantaneous rate of successfully completed requests sampled once per second for Pastry. An experimental setup with slightly more than 60 nodes was used.

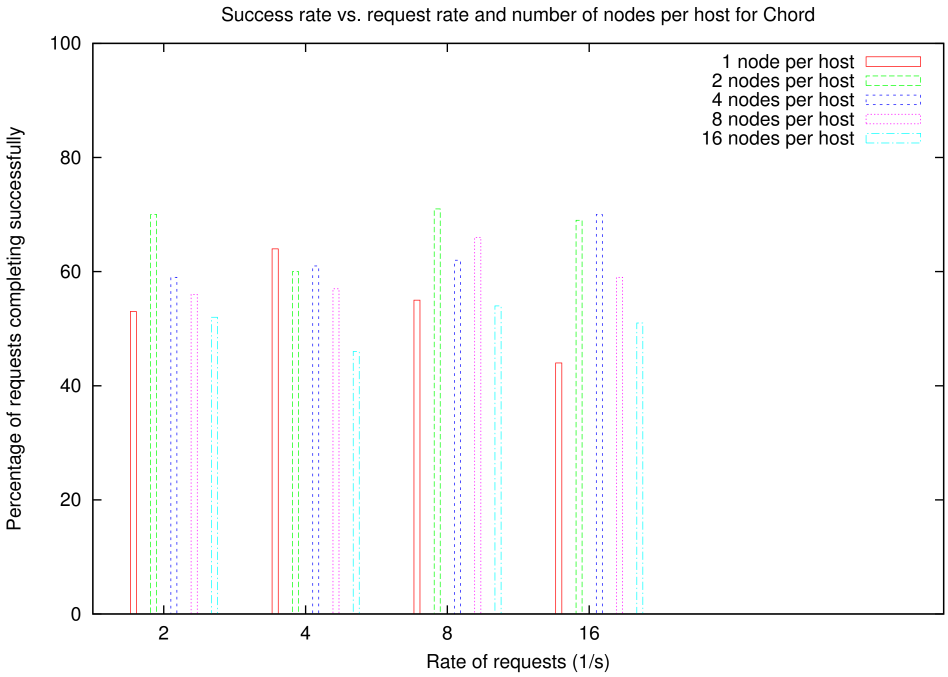


Figure 4.6: This graph shows the instantaneous rate of requests and the instantaneous rate of successfully completed requests sampled once per second for Chord. An experimental setup with slightly more than 60 nodes was used.

behaviour. For Pastry on the other hand we see how the success rate gracefully drops for extreme rates with Pastry.

Please also notice how the cumulative distribution function graph seems to plateau relatively quickly after around twice the average latency. What this seems to tell us is that if a request does not succeed within a relative short window of time, the probability of it still completing successfully is vanishingly small. This knowledge could be used to set sensible time-out values for lookups.

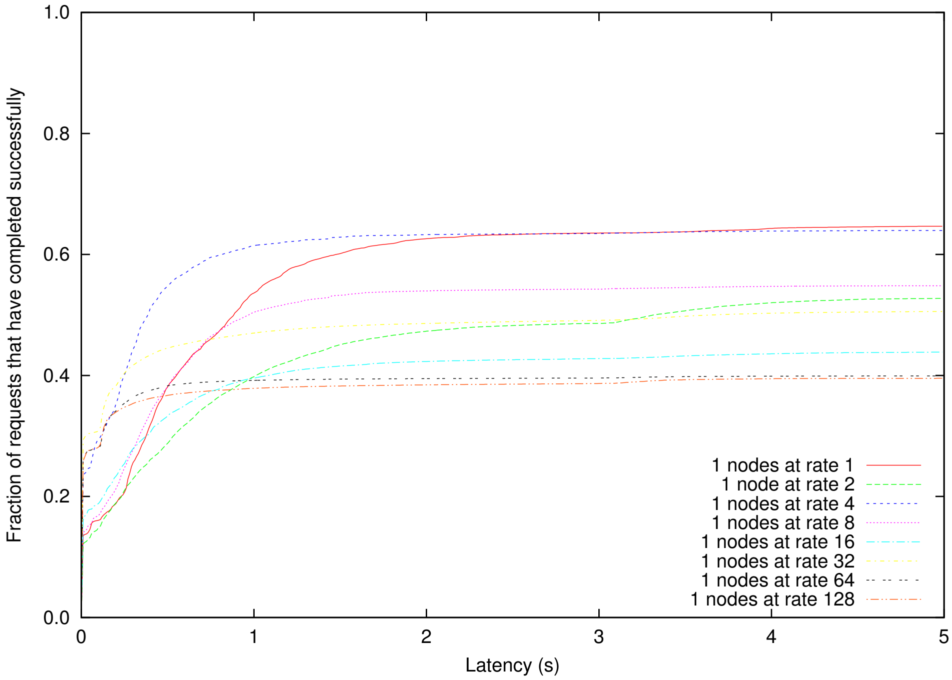


Figure 4.7: This graph shows the cumulative distribution function of a request being successful within the first 5 seconds for Chord.

4.4.2 Why Pastry performs better that Chord

The experimental evidence shown so far shows Pastry as superior to Chord both in terms of latency and in the success rate of requests. I will now try to explain in turn what the causes might be.

The higher latencies of Chord are the easiest to explain. In figure 4.9 and 4.10 you see how many nodes are involved in key lookups for Chord and Pastry respectively. It is clear from the graph that the average number of nodes involved in a key lookup are greater for Chord than for Pastry. If one multiplies the average

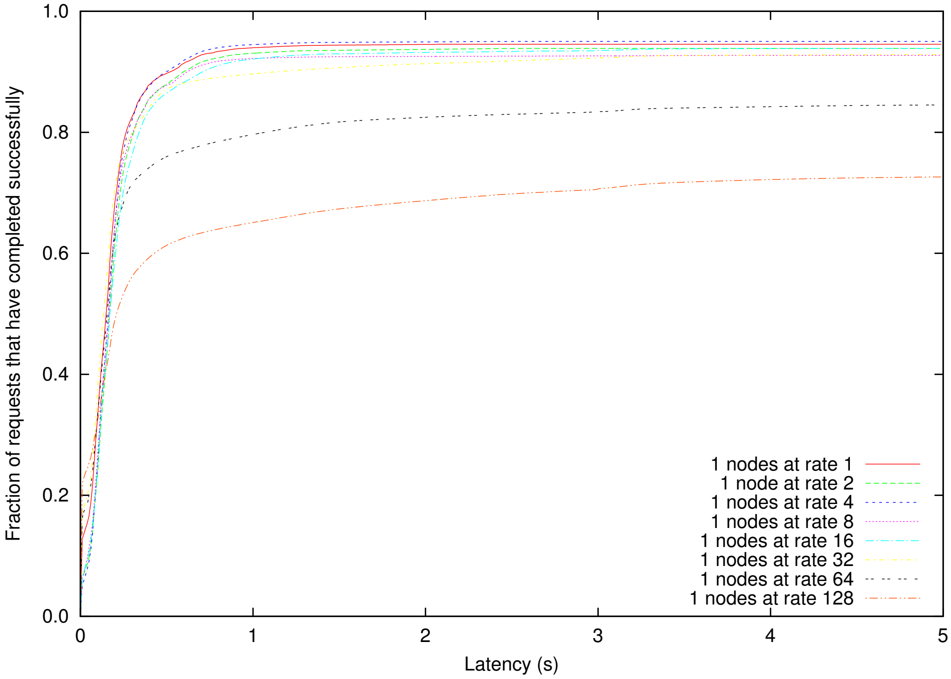


Figure 4.8: This graph shows the cumulative distribution function of a request being successful within the first 5 seconds for Pastry.

time it takes two randomly chosen nodes to connect with the number of nodes involved in a key lookup, it should be clear that a Chord lookup should take longer if the only thing one is accounting for is the latency of the routing. Additionally Pastry uses a heuristic when routing, favouring nodes closer by, which also affects the latency as each routing step is likely to take less time for Pastry than it does for Chord. I will shortly discuss the impact of the proximity heuristic in Pastry.

Whether the higher number of routing steps for Chord is a bug introduced by me or something inherent to the Chord algorithm is something I can't comment on. It is also possible that the difference would become less significant for larger networks of nodes. Unfortunately this is not something I was able to test given the infrastructure available to me.

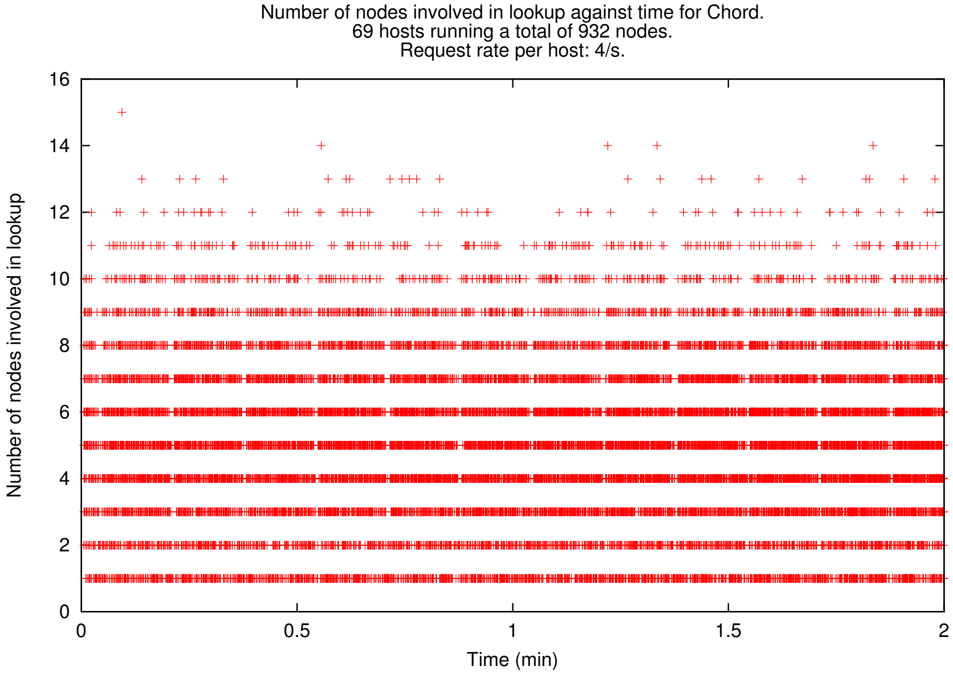


Figure 4.9: This graph shows the number of nodes involved in a key lookup in a particular experimental run for Chord.

Now that we have an idea of why the latency might be higher for Chord than for Pastry, let me try to explain why the failure rate is higher as well.

First let us recall how Chord and Pastry perform their routing. In Chord, the node that wants to lookup a key (the requester) asks the node closest to the key that it knows about if it knows about other nodes closer to the target key. It then asks the node it gets returned the same question, and the pattern continues

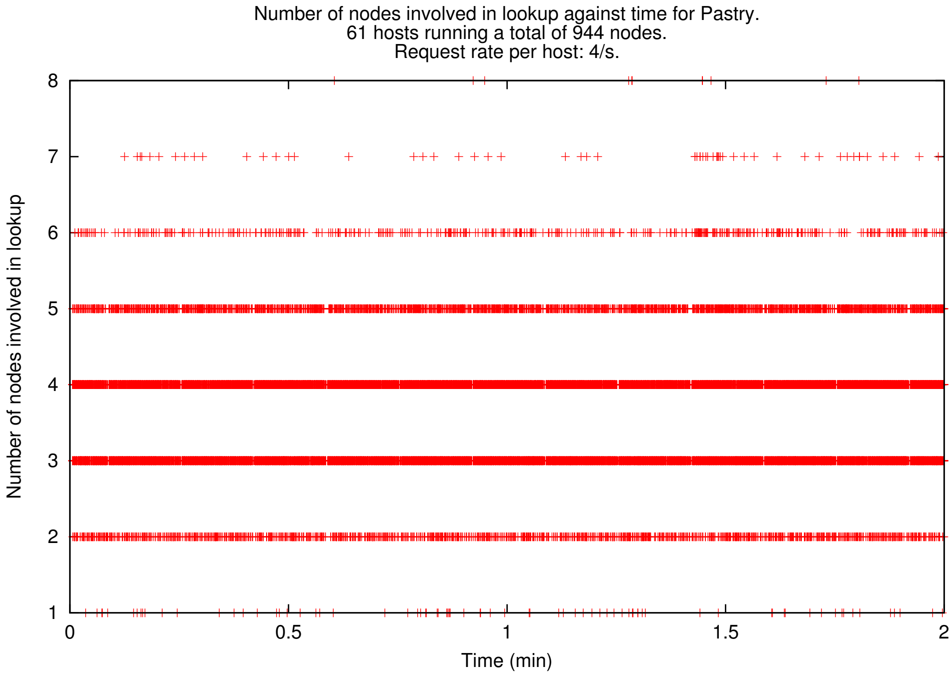


Figure 4.10: This graph shows the number of nodes involved in a key lookup in a particular experimental run for Chord.

until it finds the node immediately preceding the key. Its successor is the target node responsible for the key.

Pastry follows a different approach. When a requesting node sends a message addressed a key it forwards the message to the node it knows about that is closest to the message key. Where it differs from the Chord approach is that the requester hands over the responsibility for having the message forwarded onwards to its destination to the node it itself forwarded it to.

Now lets consider what happens when something goes wrong. In figure 4.11 we see what happens when Chord node A gets returned node D as the next hop node from B. D is not accessible and the lookup fails. If A retries the routing step, B again returns D, unless it in the meantime itself has noticed that D is inaccessible.

In the case of Pastry (figure 4.12) node B becomes responsible for routing the request forward towards the target node. Upon realising node D is unavailable it routes it to the next closest node it knows about, in this case C. Node C happens to know about a node closer to the target than D, and this way we managed to route around the unavailable node.

There are workarounds for improving Chord's routing performance. One would be for the requesting node A to inform node B about node D's death upon retrying to route the message. An alternative could also have been for node A to try to route the request through another intermediate node before D, but in this case we are not guaranteed that this intermediate node will not also return D. A third approach could be to look up D's predecessor, and then use that node in the same role previously filled by B. None of these methods have been used in my implementation as they would extend the core Chord routing algorithm and therefore skew the comparison in Chord's favour. For that matter, it might be that the solution is as simple as leaving the mechanism for maintaining routing state on for the duration of the experiments, but initial experiments run from before I decided to turn off the routing state maintenance during experiments do not seem to indicate this.

4.5 Effect of Pastry routing heuristics

In this section I will look at how the heuristics used in my implementation of Pastry affect the routing performance.

I have sampled the latencies from Pastry in a small set of different configurations with the heuristic enabled and then with the heuristic turned off. Please note that this is nothing but a very limited sample and by no means an exhaus-

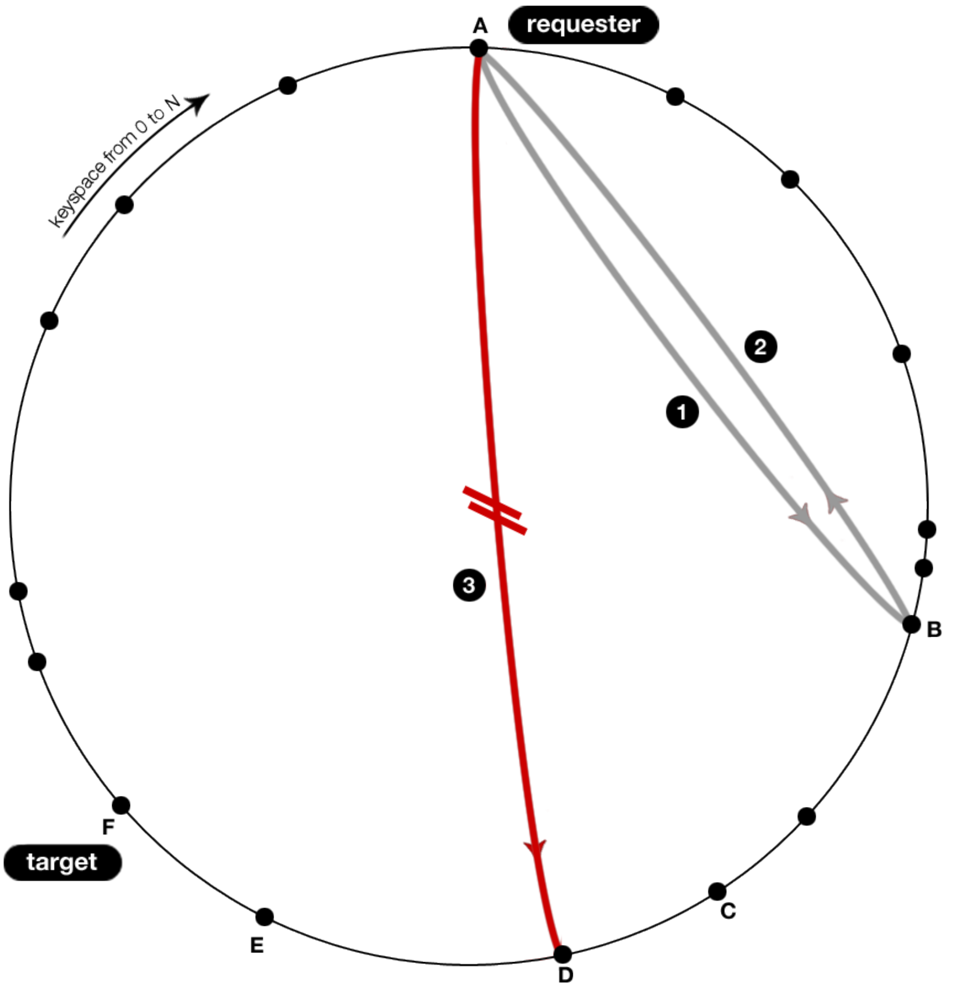


Figure 4.11: The illustration shows what happens when routing fails in Chord. We see how when node A gets the unavailable node D returned from B, it is stuck and the routing fails.

tive evaluation of the heuristic, or heuristics in general. One should not try to generalize the findings based on this data or read it as valid for anything but the sampled configurations.

The following configurations were sampled:

- 10 machines running 1 node each
- 10 machines running 5 nodes each
- 50 machines running 1 node each

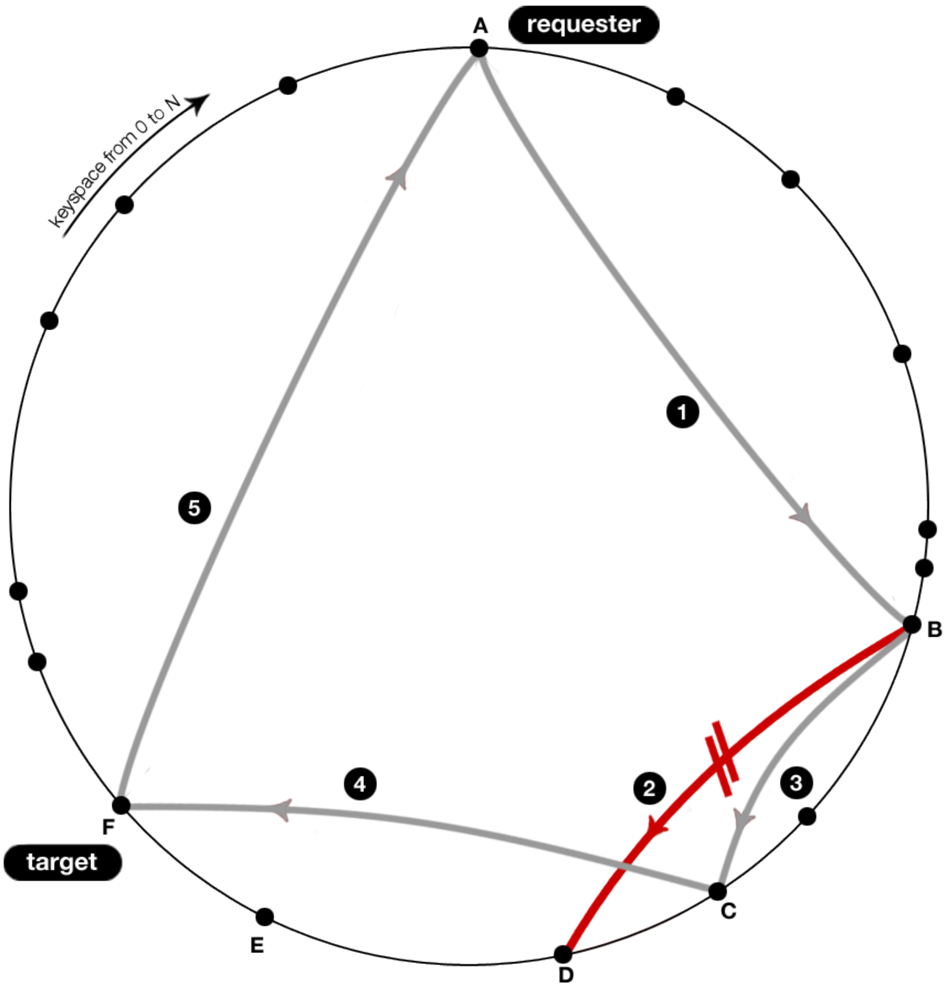


Figure 4.12: The illustration shows how Pastry deals with node failure during message routing. The unavailable node D is routed around by node B to get to the target node.

- 50 machines running 16 nodes each

The choice of configurations was made for the following reason: I wanted to see how expanding the network by adding additional machines at new geographical locations affect the latency. I also wanted to see how running 50 nodes across a few machines compared to running 50 nodes on 50 different machines. Lastly I also wanted to see how the heuristic behaved in the more extreme case with a larger network with relatively large cliques of nodes close to each other.

The results seen in figure 4.13 partly show what one would expect, in other words that the heuristic improves the routing performance.

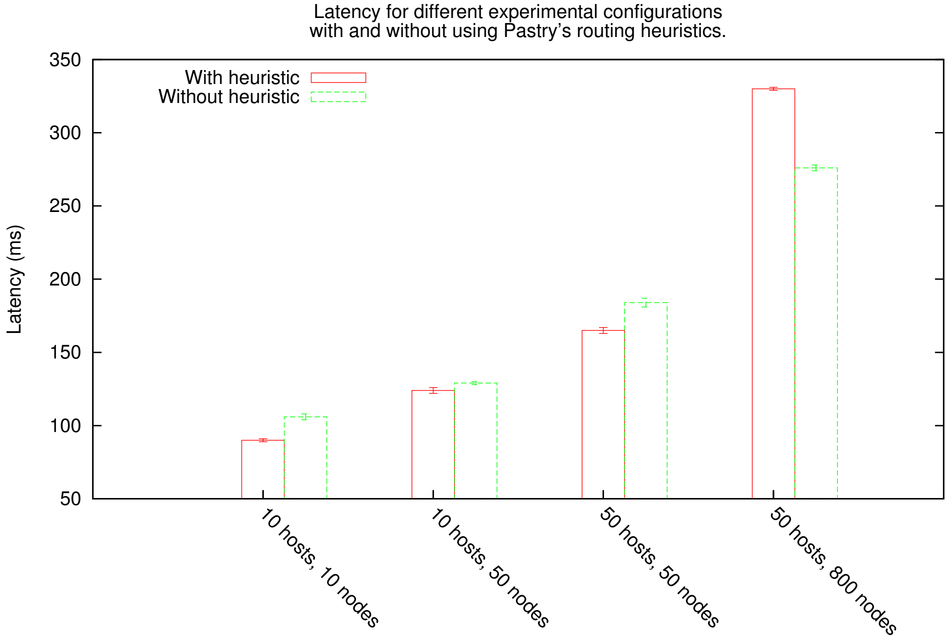


Figure 4.13: Latencies for Pastry with and without the routing proximity heuristic activated. Shown with 95% confidence intervals.

As one would expect, the heuristic allowing nodes to preferentially route to geographically closer nodes for the most part improved the performance substantially. In the two cases where only 1 node was run per host we see a 15% and 10% gain for the 10 machine and 50 machine cases respectively. Quite surprisingly, the cases where multiple nodes are located on the same machine, scenarios where I would have expected the heuristic to shine, perform less well. In the case where 50 nodes are spread over 10 hosts we only see a 4% performance gain when using the heuristic, and in the case where 50 machines run 16 nodes each, the performance drops by 20% when the heuristic is being used.

The heuristic is only calculated when nodes are added to the routing table during the routing state stabilization phase, and should therefore not affect the routing performance locally within a node. A reason could be that while the Pastry paper [4] suggests nodes use an expanding ring multi-cast to find other Distributed Hash Table nodes to connect to, which would result in the node automatically getting to know all its nearby neighbour-nodes, I made all nodes contact the central control hub in order to get nodes to connect to. The central control hub does not take node proximity into account and as a result a node might not know about all the other nodes living on the same physical hardware.

The reasoning behind this design decision originally came from the insight that most nodes would live in completely separate physical networks and that multicast messages would never be able to get from one node to the next, but making that decision I failed to account for multiple hosts living on the same machine. This is a weakness in my implementation rather than in Pastry itself, and something that would be interesting to look into for the future. While multicast might still not be an ideal solution, it would be interesting for Pastry nodes that are not the first node on a machine to use other nodes hosted on the same hardware as their entryway into the Distributed Hash Table network. Yet this does not explain why the performance is better without the heuristic as the nodes join the network the same way in both cases.

It might be that the heuristic of hop count in the underlying network is just suboptimal and that a pure latency based heuristic should be used instead. These results highlight the importance of relying on metrics when making design decisions rather than blindly implementing performance enhancements in the faith that optimisations are always for the better.

Less interesting, but reassuring non the less, is that the latency grows with increasing number of nodes in the network, just like one would expect. Unfortunately this dataset is too limited to give a general measure of how the latency grows with the network size.

4.6 Cost of using link records

As introduced in the preparation chapter, my implementation uses *link records*, records pointing to full *profile records*, in order to provide predictive and fuzzy search.

I will now evaluate how this affects the storage requirements of the Distributed Hash Table network, and also the number of key lookups that need to be done for a particular search.

First, let us look at the how the storage requirements are affected. I have a sample of 170 million names taken from Facebook. This sample might not follow the distribution of names in the real world, but most certainly quite accurately models the kind of names people give themselves in online social networks.

Based on *link records* for 20 million names, I can with 99% confidence say that the social network population mean name would require 5.322 ± 0.001 link records. I can also say that the population mean average length of a name is 14.959 ± 0.002 characters.

If each link record consists of a user's full name, the key of the *profile record* and the key of the *link record* itself, then the storage requirement of each *link record* is roughly 55bytes, or 300bytes per *profile record*, assuming each character is stored using 8 bits. In my opinion this is not an issue.

Now, let us consider the number of extra lookups that would be required for a predictive search using link records. Table 4.1 and table 4.2 show the amount of new link records that have to be downloaded when the user adds an extra character to the name being searched for and how much space these link records take. The second and third leftmost columns show the amount of link records given the link records currently used, where link records are generated for each additional three characters in a user's name. The subsequent columns show how many link records there would be if link records were generated for each additional 4, 5 and 6 characters in a users name. The link records are generated from a sample of 60 million names evenly distributed in the bag of names taken off of Facebook.

Table 4.1: Number of link new link records per additional character in the search term Archaya Pruedsakaran and the size of the link records for the cases where link records are generate for each additional 3, 4, 5 and 6 characters in a name. Green cells indicate that one of the new link records points to the profile record we are looking for.

Atchaya Pruedsakaran

	3 char		4 char		5 char		6 char	
Query	# links	size /MB	# links	size /MB	# links	size /MB	# links	size /MB
A	163600	46.81	163600	46.81	163600	46.81	163600	46.81
At	4357	1.25	4357	1.25	4357	1.25	4357	1.25
Atc	3966	1.13	112	0.03	112	0.03	112	0.03
Atch	38	0.01	3716	1.06	38	0.01	38	0.01
Atcha	106	0.03	106	0.03	792	0.23	106	0.03
Atchay	24	0.01	2	0.00	2	0.00	24	0.01
Atchaya	15	0.00	15	0.00	15	0.00	15	0.00
Atchaya P	45741	13.09	45741	13.09	45741	13.09	45741	13.09
Atchaya Pr	175	0.05	175	0.05	175	0.05	175	0.05
Atchaya Pru	15091	4.32	55	0.02	55	0.02	55	0.02
Atchaya Prue	244	0.07	2372	0.68	244	0.07	244	0.07
Atchaya Prued	0	0.00	0	0.00	5	0.00	0	0.00
Atchaya Prueds	2	0.00	0	0.00	0	0.00	2	0.00
Atchaya Pruedsa	0	0.00	0	0.00	0	0.00	0	0.00
Atchaya Pruedsak	0	0.00	2	0.00	0	0.00	0	0.00
Atchaya Pruedsaka	2	0.00	0	0.00	0	0.00	0	0.00
Atchaya Pruedsakar	0	0.00	0	0.00	2	0.00	0	0.00
Atchaya Pruedsakara	0	0.00	0	0.00	0	0.00	0	0.00
Atchaya Pruedsakaran	2	0.00	2	0.00	2	0.00	2	0.00

Table 4.2: Number of link new link records per additional character in the search term Brad Catron and the size of the link records for the cases where link records are generate for each additional 3, 4, 5 and 6 characters in a name. Green cells indicate that one of the new link records points to the profile record we are looking for.

Brad Catron

	3 char		4 char		5 char		6 char	
Query	# links	size /MB	# links	size /MB	# links	size /MB	# links	size /MB
B	73400	21.00	73400	21.00	73400	21.00	73400	21.00
Br	434	0.12	434	0.12	434	0.12	434	0.12
Bra	956906	273.77	475	0.14	475	0.14	475	0.14
Brad	157479	45.06	305181	87.31	157479	45.06	157479	45.06
Brad C	91473	26.17	91473	26.17	91473	26.17	91473	26.17
Brad Ca	3800	1.09	3800	1.09	3800	1.09	3800	1.09
Brad Cat	4 63292	132.55	22172	6.34	22172	6.34	22172	6.34
Brad Catr	2	0.00	13414	3.84	2	0.00	2	0.00
Brad Catro	30	0.01	30	0.01	857	0.25	30	0.01
Brad Catron	748	0.21	703	0.20	703	0.20	748	0.21

What we see is that there are a lot of link records for 1 character search queries (163k for *a* and 73k for *b*) and in the case of link records generated for each 3 or 4 additional characters also when the query is as long as the length at which the first link record is generated. Short common names like Brad also generate a lot of link records.

What gives this approach some hope is that there are significantly fewer link records when longer name fragments are used for generating link records, and also in the cases where we generate link records for each 3 additional characters, there seem to be much less link records for the second link record generated for a name.

Caching link records for all character combinations up to 3 characters could be a possibility, but back of the envelope calculations based on the numbers presented indicate that this would require several gigabytes worth of local caches per host, and probably more than that if the system was also to accommodate non-latin character sets. While possible, this might be a bit too much to ask of the independent distributed online social networks.

Other promising performance enhancements like adding query length thresholds and compressing the link record data are discussed and evaluated in Appendix C.

4.7 Utility of Chord and Pastry as back end data stores for a search engine

In its current raw form, Chord is not really a viable data store for any application. Its high and variable latency, although undesirable, is something one could live with, if the key lookups would not have such a high failure rate. Pastry on the other hand proves a quite efficient key-value store, even in highly distributed environments. Its combination of high success rates and consistently low key lookup latencies and gradual degradation in performance under high load are all desirable properties.

If one uses a combination of caching link records at the search servers, add a query length threshold before which no requests are made to the search network, and make the search servers more capable of judging the relevance of link records, in addition to compressing and minimizing the size of link records, I think key-value stores could quite efficiently be used as the back end data store for a search engine. Especially considering the benefits Distributed Hash Tables bring like built in load balancing, fault tolerance and predictable and scalable performance.

In this chapter we have seen how my implementation of Pastry outperforms my implementation of Chord both in terms of latency and key lookup success rate. We have also discovered how the proximity heuristic used in routing by Pastry quite counter intuitively does not always yield performance improvements, and have understood the importance of relying on metrics rather than intuition when evaluating performance.

We also looked at the impact using link records has on the amount of data that needs to be stored in the network and the number of link records loaded during two sample searches.

Finally we concluded that with some optimizations in the search server, Distributed Hash Tables can work quite well as the data store for a Distributed Search engine of the particular kind I have proposed.

5. Conclusion

In my project I created a search engine to be run by independent and distributed online social networks. The search engine honours the fundamental ideal of independent online social networks of allowing their users control over what and how much data about them is made publicly available, and also solves the problem of allowing users of independent online distributed social networks to find and reconnect with their friends regardless of in which online social network they have their profile.

I built this service on top of Distributed Hash Tables and invented a scheme for enabling fuzzy and predictive searches across key-value stores. This scheme works well in its current form for installations with small to medium sized data sets, and could work well in larger installations if slightly tweaked.

As part of my project I implemented two Distributed Hash Tables. While both work correctly, my implementation of Pastry works significantly better than Chord, and gives quite impressive results.

My project was highly successful as a proof of concept, but would need further refinement before being deployed by real social networks, which is in line with what I set out to do.

While being a success in its own right, the project has also been an excellent learning experience. I had never used the programming language Erlang for anything but trivial “hello world” applications before the start of my project and now have a solid working knowledge of it and its libraries. I also got significant exposure to working with highly distributed systems and deploying software across up to 100 machines. The same project implemented again from the beginning would cost me significantly less time and effort considering all the knowledge I now possess, but that itself can also be considered a success, considering knowledge acquisition is a central component of what role I believe the Part II projects are designed to achieve.

As a whole the project solves a very real problem that needed addressing to help independent distributed social networks stand a fighting chance in gaining ground against the social networking gorilla Facebook. There is still a lot of work that needs doing, but my project serves an important role in highlighting an obstacle that needs addressing, and proposed a solution for how it can be tackled.

6. Acknowledgements

I want to thank Dr David Evans for supervising my project and for always being supportive and willing to help and give tips when I got stuck.

I want to thank Felix Bauer for spending countless hours helping me make sense of my experimental results and helping me out with LaTeX, Eugene Chan for being supportive and encouraging and Josh Ward for being so kind to proofread my dissertation.

I want to thank Dr David Eyers for all the help he gave me while I was planning my project and also for giving feedback on the project proposal.

And finally I want to thank Dr Robert Harle and Prof Jon Crowcroft for taking the time to read through drafts of my dissertation.

Bibliography

[1] Joe Armstrong, *Programming Erlang, Software for a Concurrent World.* The Pragmatic Programmers, Version: 2008-7-6.

[2] Francesco Cesarini and Simon Thompson, *Erlang programming.* O'Reilly Media, 2009.

[3] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan, *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications.* MIT Laboratory for Computer Science.

[4] Antony Rowstron and Peter Druschel, *Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems.* Microsoft Research Ltd, St. George House.

[5] Miguel Castro, Manuel Costa and Antony Rowstron, *Debunking some myths about structured and unstructured overlays.* Microsoft Research, 7 J J Thomson Avenue, Cambridge, UK.

[6] Peter Maymounkov and David Mazieres, *Kademilia: A Peer-to-peer Information System Based on the XOR Metric.* New York University.

[7] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, Member, IEEE, and John D. Kubiatowicz, Member, IEEE, *Tapestry: A Resilient Global-Scale Overlay for Service Deployment.*

[8] *Wikipedia: Distributed Hash Tables.* http://en.wikipedia.org/wiki/Distributed_hash_table.

[9] Dr David Evans *Lecture notes on Experimental Design.* CS 457/657, Winter 2005.

[10] Serge Aleynikov, *Building a Non-blocking TCP server using OTP principle.* TrapExit.org / http://www.trapexit.org/Building_a_Non-blocking_TCP_server_using_OTP_principles.

[11] Andreas Stenius, *gen_listener_tcp*. Github.org / https://github.com/kaos/gen_listener_tcp.

[12] Jason Wagner, *erlymock*. Github.org / https://github.com/nialscorva/erlymock/wiki.

[13] Jason Wagner, *Erlang, Testing and TDD*. http://erlcode.wordpress.com/.

[14] Basho, *Webmachine documentation*. http://webmachine.basho.com/.

A. Routing state as Erlang data structures

I use Erlang records for the Chord and Pastry routing state. When regular tuples are used to pass data-structures between functions, the amount of code that needs updating when the data structure evolves is often substantial. While you need to write out the full structure of a tuple to pattern match on one of its values, records give you direct access to named members of the structure, abstracting away the physical layout of the data. Under the hood, records are still compiled down to regular tuples, but this is handled transparently without the programmer needing to know about it or update code where the data-structure is being used.

A.0.1 Chord routing state

Code listing A.1 shows how I translated the Chord routing state described in section 3.3.1 on page 13 into Erlang records.

Listing A.1: Chord routing table as Erlang records

```
-record(node, {
    ip :: ip(),
    port :: port_number(),
    key :: key()
  }).
-record(finger_entry, {
    start :: key(),
    interval :: {key(), key()},
    node :: #node{}
  }).
-record(chord_state, {
    self :: #node{},
    predecessor :: #node{},
    fingers = array:new(160) :: array(),
    % ...
  }).
```

A.0.2 Pastry routing state

Code listing A.2 shows how I translated the Pastry routing state described in section 3.3.1 on page 14 into Erlang records. The distance field on line 6 in the node record in the listing is the numeric distance between a particular node and the node maintaining the routing state, as returned by the proximity heuristic used for routing.

Listing A.2: Pastry routing table as Erlang records

```
 1 -type(pastry_key() :: [integer()]).
 2 -record(node, {
 3     key :: pastry_key(),
 4     ip :: ip(),
 5     port :: integer(),
 6     distance :: number()
 7   }).
 8 -record(routing_table_entry, {
 9     value :: integer() | none,
10     nodes = [] :: [#node{}]
11   }).
12 -record(pastry_state, {
13     % The routing table contains log base 2^b rows of 2^b - 1
14     % entries each. Entries at row n share the first n digits
15     % with the current node, but differ in digit n + 1
16     routing_table :: [#routing_table_entry{}],
17
18     % The nieghborhood set contains the nodes closest to the
19     % given node according to the distance metric used.
20     % The size of the table is roughly 2^b
21     neighborhood_set = [] :: [#node{}],
22
23     % The leaf set usually contains up to 2^b nodes in total
24     leaf_set = {[], []} :: {[#node{}], [#node{}]},
25
26     % Information about the current node
27     self = #node{},
28
29     % ...
30   }).
```

B. Pastry routing algorithm in Erlang

This section compliments section 3.3.2 on page 21 and shows how I translated the Pastry pseudo code routing algorithm into Erlang.

My Erlang implementation of the Pastry routing algorithm, rather than being a direct translation of the pseudo code (listing 3.6 on page 23), takes the essence of what the Pastry routing algorithm tries to accomplish, and implements that.

The code in listing B.1 shows my main Erlang `route_msg` function. Just like the pseudo code in pseudo code listing 3.6 on page 23 it tries routing via the leaf set, then the routing table and if all else fails to any node that is closer to the target key.

Listing B.1: Routes a message towards a recipient

```
route_msg(Msg, Key, State) ->
  spawn(fun() ->
    route_to_leaf_set(Msg, Key, State) orelse
    route_to_node_in_routing_table(Msg, Key, State) orelse
    route_to_closer_node(Msg, Key, State)
  end).
```

In listing B.2 we see that if the match in the leaf set is the node itself, then on line 5 the message is delivered to the main pastry application. Otherwise the message is forwarded to a closer node if appropriate on line 7.

Listing B.2: Routes a message to the leaf set if applicable

```
1 route_to_leaf_set(Msg, Key, #pastry_state{self = Self, pastry_pid = PastryPid,
      pastry_app_pid = PastryAppPid} = State) ->
2   case node_in_leaf_set(Key, State) of
3     none -> false;
4     Node when Node =:= Self ->
5       pastry_app:deliver(PastryAppPid, Msg, Key),
6       true;
7     Node -> do_forward_msg(Msg, Key, Node, PastryPid)
8   end.
```

If routing to the leaf set fails (line 3 in listing B.2), the node tries routing the message to a node in the routing table sharing more digits in the key with the message key than what itself does. Again, just like when routing to the leaf set, if the best match is itself, the message is delivered. This function is shown in code listing B.3.

It is not quite as easy to see how this code listing corresponds to the pseudo code in listing 3.6 on page 23. For example, in the listing B.3 there is nothing that directly correspond to getting the length of the shared key path as we see on line 9 in the Pastry pseudo code in listing 3.6 on page 23. Instead what the Erlang implementation does is find all the nodes in the routing table that share as many key digits with the message key as the node itself does (line 2) in addition to the `PreferredKeyMatch` (also line 2) which is the shared key-segment plus the first digit in the message key that differs from the key of the node itself. On line 4 we look for a node in the list of nodes which shares all the digits in the `PreferredKeyMatch`. If there is one then the message is passed on to that node and if there is none, we return empty handed.

Listing B.3: Routes a message to a node in the routing table

```
 1 route_to_node_in_routing_table(Msg, Key, #pastry_state{pastry_pid = PastryPid,
       pastry_app_pid = PastryAppPid} = State) ->
 2   {#routing_table_entry{nodes = Nodes}, [none|PreferredKeyMatch]} =
 3       find_corresponding_routing_table(Key, State),
 4   case filter(fun(Node) -> is_valid_key_path(Node, PreferredKeyMatch) end, Nodes)
         of
 5     [] -> false;
 6     [Node] ->
 7       case Node =:= State#pastry_state.self of
 8         true -> pastry_app:deliver(PastryAppPid, Msg, Key);
 9         false -> do_forward_msg(Msg, Key, Node, PastryPid)
10       end
11   end.
```

If all other routing approaches fail, the last resort the Pastry node has is to find any node that is numerically closer to the key than what itself is. Code listing B.4 shows the Erlang implementation finding the key segment shared between the message key and the node's key (line 2) and then finding all the nodes it knows about that share this key segment (line 3). On line 7 the node numerically closest to the message key is found and the message either forwarded or delivered.

Listing B.4: Routes a message to any closer node

```
 1 route_to_closer_node(Msg, Key, #pastry_state{self = Self, b = B, pastry_app_pid =
       PAPid, pastry_pid = PastryPid} = State) ->
 2   SharedKeySegment = shared_key_segment(Self, Key),
 3   Nodes = filter(
 4     fun(N) -> is_valid_key_path(N, SharedKeySegment) end,
 5     all_known_nodes(State)
```

```
),
case foldl(fun(N, CurrentClosest) ->
        closer_node(Key, N, CurrentClosest, B) end, Self, Nodes) of
  Self -> pastry_app:deliver(PAPid, Msg, Key);
  Node -> do_forward_msg(Msg, Key, Node, PastryPid)
end.
```

C. Optimizing the link records approach

This appendix on optimizing the link record approach to search compliments section 4.6 on page 47 where I discuss the cost of using link records to enable predictive and fuzzy searches on top of Distributed Hash Tables.

An optimization that could be quite promising, but is not part of my current search server implementation, is to not start looking for link records unless the search term exceeds a minimum length. The immediate drawback of this approach is that it is less interactive, but this lack of interactivity could easily be camouflaged by having the search engine display matches taken from data cached from local caches resulting from previous searches, and adaptively load more results from the distributed search network as the query gets longer. The threshold has to be matched to the way the link records are generated. Say if we generated link records for each additional 5 characters and had set the threshold to 4, then no correct matches could ever be found for names longer than 5 characters before the 5th character in the search query has been entered. That is unless cached results already exist on the search engine node. The threshold value has to be set according to how long the average name is. There are a significant proportion of names 4 character or less in length, and by setting the threshold to a value higher than 4 these short names would never be found using the predictive search unless they had already been cached. One would also have to consider if the threshold should be a per name threshold, by which I mean that we do not look up link records for name fragments shorter than the threshold, or a per query threshold, in which case surnames would already trigger a link record lookup after their first character has been entered, which in the case of the letters *a* and *b* would trigger 47 and 21 MB downloads of link records respectively (from table 4.1 on page 48 and 4.2 on page 49 respectively). It could be a good idea to set the threshold on a per name basis, but continue to order and prioritize the already downloaded link records given the full name component they contain already from the very first character the user types in the subsequent names. I believe setting a search threshold of 2 or 3 characters combined with generating link records for each additional 4 characters could be a good compromise. It would allow us to find the short names interactively, and would still ensure a relative sparsity of link records.

The danger with generating link records for too long name fragments is that it makes fuzzy searching harder to achieve, as fuzzy searches rely on either already having found the correct link record before the spelling mistake is done, or find a correct link record for one of the person's other names and then give that a higher priority due to it being a close match for the misspelled full name.

Another optimisation to the link record worth considering is how one could compress the data they occupy in the search network and that needs to be transferred during search. In the current implementation link records quite unnecessarily store their own key. 160-bit keys also take up 192 bits of spate on 64-bit machines and the names themselves are not at all compressed. If one removed the self-referencing keys, used 128-bit keys and compressed the full name by a factor of 2, then a link record could take 125 bytes instead of 300 bytes per record, which would reduce the storage and transfer requirements by a factor of 2.4.

D. Source code

My project is fully open sourced. The source code can be found on my github account under: https://github.com/sebastian/Part-2-project

E. Project Proposal

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.[1] 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:

[1] 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[2]
- 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

[2] 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.[3] 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).

[3] 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.