Improving DHT Routing Performance in Harmony using Client Caching

Allison Obourn

A senior thesis submitted in partial fulfillment of
the requirements for the degree of

Bachelor of Science
With Departmental Honors

Computer Science & Engineering
University of Washington
June 2011

Abstract

Harmony is a strictly consistent distributed key-value storage system. It is comprised of multiple groups which contain nodes in the system. Groups are arranged in a circle and know only about their adjacent groups. For a system of N groups up to O(N) groups may need to be queried to locate a target key. To achieve faster and more scalable lookups we have implemented a new client-side caching routing system for Harmony. This new routing system allows a client's entry point to be a node closer to the desired destination than previously possible. It adapts prior work in routing to use possibly inconsistent cached information about the system. Churn within Harmony causes the cached data to become incorrect. This is resolved in the client by updating its cached data on each access to a node. This is further improved by pinging frequently accessed nodes in the background. The new routing system scales better than the previous as it can access Harmony from multiple entry points. Our new method provides a 100 times decrease in lookup latency for the basic system and a 10 times decrease for the pinging configuration, across 95% of the lookups.

Presentation of work given on May 25, 2011

Thesis and presentation approved by:

Date:

Contents

1 Introduction

Fast access to distributed data is increasingly important. More and more data is being moved to cloud storage. For this to be workable the cloud must be able to store this data consistently and return it to a client requesting it quickly. Distributed systems are used to hold this data and keep it consistent. Routing is done to find the data and return it to a requesting client. As more data moves to web-based storage distributed systems will get bigger and bigger. The bigger they get the more important it is that we have smart efficient routing mechanisms to locate data.

A distributed system is a system of two or more machines networked together that appear to a client as a single entity. They provide more resources and greater redundancy than single computer systems. Greater resources allow them to serve more users at an increased speed. Redundancy allows them to recover from failure without losing data. Because of these desirable properties they are used very widely today.

Unfortunately using multiple machines also creates some new problems. According to Leslie Lamport, “A distributed system is one in which the failure of a computer you didnt even know existed can render your own computer unusable”. The more computers that you have in a system the higher the chances that one of them will fail. If a single computer fails you may lose data but you will at least know where the problem is and what needs to be fixed. In a distributed system some far away computer can fail and cause you to not be able to use the system.

DHTs are a class of highly-scalable distributed storage systems. DHTs store (key, value) pairs just like a regular hash table data structure. Keys are distributed among all the nodes of the system. To look up a piece of data in a DHT the data is hashed and the value is the value at the key in the system the data hashed to. The client must be able to find where this key is stored. This locating of the key is called routing and is what we will be discussing for the remainder of this paper.

Distributed systems must provide fast and consistent access to data. In order to get information in a distributed system, the nodes need to know each others addresses. Since nodes are constantly failing, distributed systems must be designed to cope with an ever changing topology. If nodes leave and/or join the system a lot it can be very cumbersome and time consuming for each of them to keep an up-to-date list of all other nodes currently in the system. As the system grows this list will become too big to manage efficiently. There is a trade off between the amount of routing data that should be stored so that data can be found and the overhead of the space for and upkeep of it. Storing some routing information about the system on the client can help alleviate this problem.

This thesis adds a routing layer to Harmony [1], a strictly consistent distributed system. This work augments the existing Harmony design to improve routing performance. It is completely client side so that it can be easily ported to different versions of Harmony and provides a set of routing hints that decrease lookup latency by 100 times for 95% of lookups.

In section 2 we will discuss other systems that influenced our design. In section 3 we will describe Harmony, the system that this routing system is built for. We will give a description of its structure and an explanation of how its old routing system worked. Section 4 contains a description of our new caching routing strategy and section 5 contains our cache maintenance strategy, an addition that we thought would improve our system. Section 6 contains the evaluation of this system, section 7 future work, section 8 what I have learned through this project and section 9 presents our conclusions.

2 Related Work

Previous work involved DHTs that store routing data on the nodes of the system itself. Storing routing data on the system means that these routing strategies need to touch the lower layers of the system and are heavily coupled with it. This section also overviews previous work on client-side routing. In our work, we use a combination of these techniques to create a quick and modular system.

2.1 Traversing the system using neighboring groups

The only connection to the rest of the system that Harmony nodes have is their links to their group members and to their neighboring groups. Another system with similar knowledge is CAN [5]. CAN is a DHT with a d dimensional toroidal space. Each node owns the keyspace of a particular hypercube and has a link to each of its immediate neighbors. Routing works by forwarding the request to the neighbor closest to the key. Each node has $O(d)$ neighbors and path lengths are $O(d\ n^{1/d})$. This setup is resilient as even if the closest node to where a node needs to route fails there are other nodes that it can route through. Harmony groups behave very similarly to nodes in CAN. Harmony groups only have two direct connections so it is similar to CAN where $d = 2$. Increasing Harmony's dimensionality, while very interesting, would not create a modular system that could be moved between versions. Extending Harmony to have a variable d is left as further work and traversing the system using previous and next pointers is borrowed.

2.2 Which routing data is stored

Some routing systems have chosen to store incomplete routing information. One of these is Chord [6]. It stores routing information in the form of a finger table which contains the addresses of the nodes spaced 2^k around the circle. This speeds up routing by allowing the node to access nodes as far away as the opposite side of the circle, quickly. Each node also stores pointers to its r closest successor nodes. This helps ensure correctness if the direct $r - 1$ successors leave the system. Routing to a node is done by routing to the closest node before the desired node. If the node routed to is not the desired node then this is repeated. Chord stores this routing data on the nodes in the system. Harmony does not have routing data stored on the nodes currently and adding it would not allow the system to be modular. The client could have a routing table similar to Chord's, storing addresses spaced 2^k around the circle. This does not make as much sense for the client as for a node in the system. The client's start node could be $O(log\ n)$ away from all of the data that it frequently accessed.

A better solution would be to store more routing data like Dynamo [2]. Dynamo is a highly available system designed by Amazon. It is a zero-hop DHT as each node has enough routing information to reach any other node. Because Dynamo is not strictly consistent, routing is a bit easier than in Harmony as a request does not necessarily have to be received by all nodes containing the data before it is processed. A client does not need to know all of the routing data in Harmony but can use a similar system to be more knowledgable about the parts of the system that it needs to access.

2.3 Pinging nodes to check if they are still in the system

It is important for the routing system to be able to detect whether or not a node it has cached is still in the system. Kademlia [3] uses pinging to find out this information. It stores lists of addresses 2^k around the circle, similar to Chord but keeping multiple addresses for each range. The amount of addresses it can store for each range is capped. When the cap is exceeded it must decide whether to replace its old cache contents with the new data. To do this it sends a ping to the old address to see if it is still alive. We use this pinging in a similar way to keep our cache up to date.

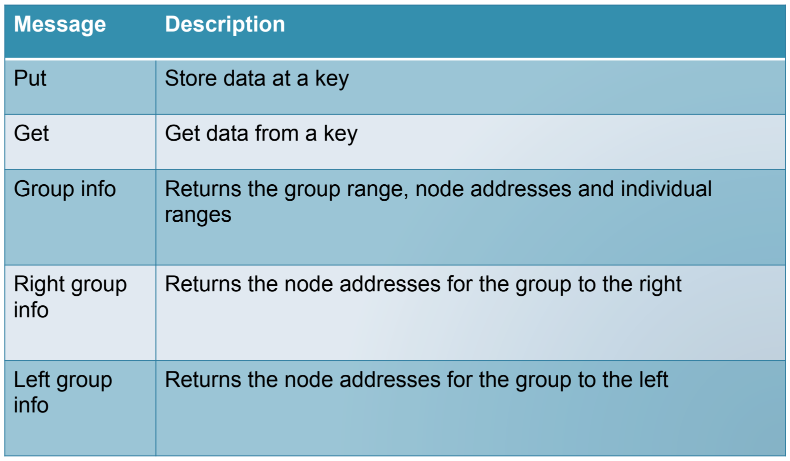
2.4 Client side caching

In order to maintain modularity but have caching we place the routing cache on the client. DNS [4] also uses client side caching. It translates domain names into IP addresses on the internet. When a client wants to connect to a webpage for the first time it looks up the IP address in DNS. It then caches this IP address so that it can access it again more quickly. Our system does the same thing by looking up the address the first time and then using it on repeated accesses.

3 Harmony

Harmony is a DHT which stores keys mapped to values. It ensures data consistency across the whole system.

3.1 Interface



Message	Description
Put	Store data at a key
Get	Get data from a key
Group info	Returns the group range, node addresses and individual ranges
Right group info	Returns the node addresses for the group to the right
Left group info	Returns the node addresses for the group to the left

Figure 1: Messges that can be sent to and from Harmony

The five message types in Figure 1 are built into Harmony. They are UDP messages so they are unreliable. Gets and puts are necessary for a user. For routing we will be sticking to using the *group info* message to learn about the system and *left group info* and *right group info* to learn adjacent groups addresses and move around the system.

3.2 Structure

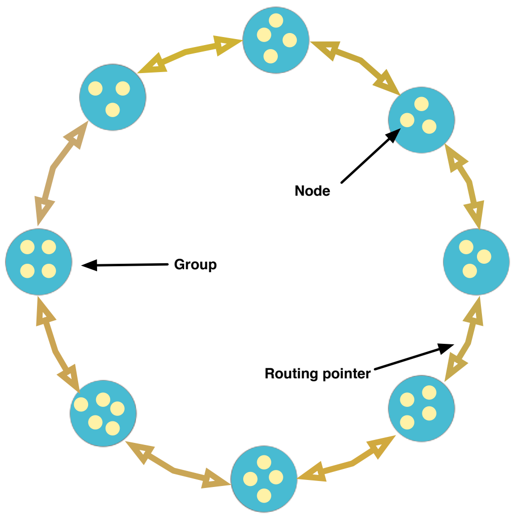


Figure 2: Parts and layout of Harmony

Nodes in Harmony are members of groups. These groups are arranged in a doubly linked circle where each group has a consistent link to the group to its right and the group to its left as shown in figure 2. Each group implements Paxos so the majority of nodes will know the correct group membership and neighbor membership but there is no guarantee that every node will know it. Since Harmony is consistent keys will always be able to be found as long as the system is not reconfiguring at the time of the request. Attempting to access a key during reconfiguration might cause the key to be temporarily unavailable. However, once the reconfiguration has finished the key will be available again.

Keys are distributed evenly among all of the nodes in the system upon startup. The key values increase to the right (clockwise in all of our diagrams). As nodes enter and leave the system keyranges need to be adjusted so that every keyrange is always covered by a node. Groups will also be adjusted if too many nodes enter or leave the system. Harmony will split groups that get too large and merge groups that get too small.

3.3 Linked-list routing

In order to use Harmony the client must know about some node in the system. We will refer to this node as the start node. Before this work, routing was started by routing to the start node and getting its group info. If the key being looked up was in the group then the client would know that it was at the node address that owned that key in the returned group info. Otherwise, the client would figure out whether the desired key was bigger or smaller than the current group's and ask for the group info for the group in the correct direction. It would then route to this group and continue until it got to the correct group. We refer to this type of routing as linked-list routing and have included the pseudocode below in figure 3.

```
define k      // key that the client is attempting to locate
define ip_r   // IP of the root node that the client contacts initially
let ip = ip_r

while (k not in current_group.kspace):
  if (k > current_group.kspace):
    current_group = find_right(current_group)
  else:
    current_group = find_left(current_group)
```

Figure 3: Pseudocode for linked-list routing

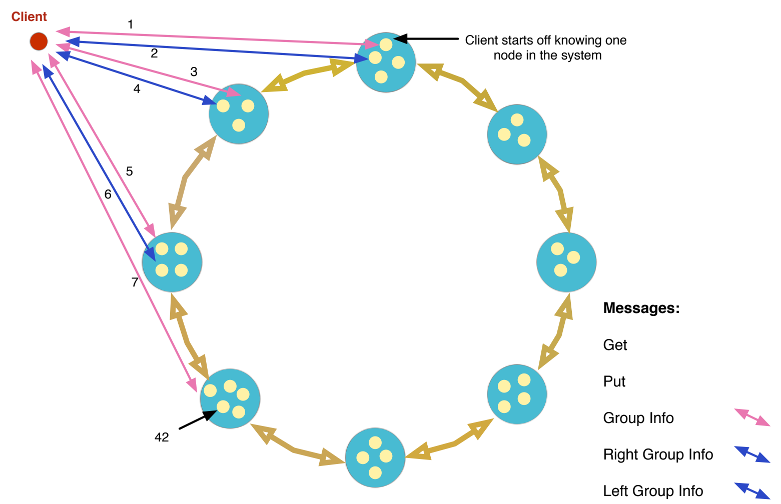


Figure 4: Executing a lookup for key 42 in a system with a keyspace of 64

Figure 4 illustrates a lookup for key 42 in a system with a keyspace of $[0 - 64]$. The start node is the node at the top of the diagram. This node is in the group that holds keys 0-7. The client contacts the start node with a $get\ info$ message. The response to this message will give the client all of the information it needs to determine if any nodes in the group have the key it is looking for. If a node in the group contains the sought after key than we have found our goal. Otherwise we need to keep searching. Because the keyspace is circular and we know the direction it increases in and its maximum size we can determine whether the sought after key is closer to our left or to our right. We can then either get $left\ group\ info$ or $right\ group\ info$. Since we are searching for 42 going left will be faster. The client will then repeat this series of $group\ info$ and $left\ group\ info$ messages until it reaches 42.

Although this method of routing will eventually get a client to the data it is requesting, it is not very efficient. Path lengths are linearly dependent on the size of the system ($O(n)$). This means that lookup latency will scale linearly with the size of the system. It also means that if a user does the same lookup twice in a row if must return to the starting node every time. This is very inefficient, especially as the system gets bigger.

4 Algorithm Overview

Routing state is accumulated on a client each time it accesses the system. On its first access the client has to traverse the system using only the previous and next group pointers as shown in the pseudocode in figure 5. It routes iteratively through groups in the ring until it reaches the node responsible for the key it is looking for, just like the initial linked-list routing scheme. By caching this information on the client our design has remained independent of the underlying system.

```
define k     // key that the client is attempting to locate
define ip_r  // IP of the root node that the client contacts initially
cache.update(ip_r.group_info)
let current_group = cache.get(k)
let ip = ip_r

while (k not in current_group.kspace):
  cache.update(current_group.group_info)
  current_group = cache.get(k)
  if (k > current_group.kspace):
    current_group = find_right(current_group)
  else:
    current_group = find_left(current_group)
```

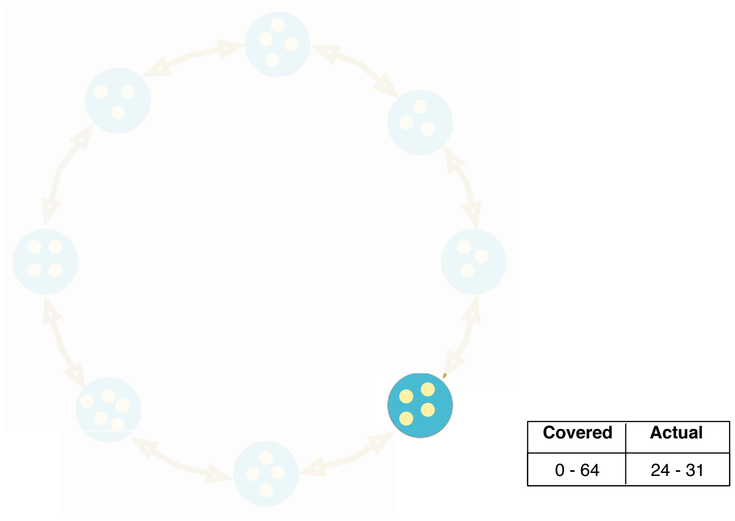
Figure 5: Pseudocode for cached routing

When the client needs to get information about a group or its neighbors it will choose a node at random to ask. We chose this method because every node in the group has the information that it needs and randomizing will even out the load. The only time a client contacts a specific node is when it thinks it has found the node it is looking for.

The difference between the two routing systems is that this new system caches the routing information that the client receives from $get\ info$ messages. This information contains the address range and IP address for each node in the group. Storing information for the entire group as opposed to just a single node is beneficial as it is equivalent to storing multiple pointers to the same thing. One node has a much greater chance of leaving the system than a group of nodes. If a client tries to access a node in the group and that access times out then it can try to send the same type of message to another node in the group. This is more efficient than sending to a node in a different group as then there would also have to be messages sent to route back to the target group. The client will always receive information about all group members anyway so there is no extra cost in accumulating this information.

4.1 Routing Cache Structure

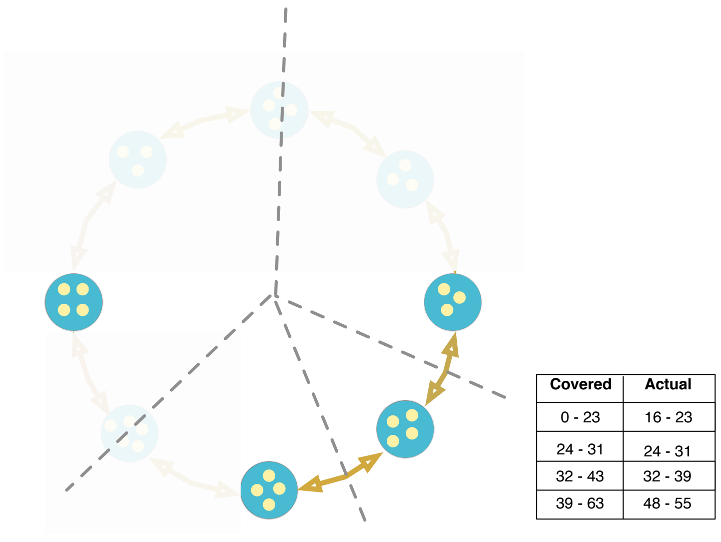
The routing cache contains the keyrange of an entire group mapped to a list of the node IP, port and keyranges. This keyrange is referred to as the actual keyrange as it is the keyrange that the group actually contains. It is likely that we will have information about some groups and not about others. If a client wishes to find a key that is not in the cache than it should access a node in the group in the cache that has the closest keyrange to the desired key. In order to do this we have created covered keyranges. A covered keyrange is the range of keys for which the entry point into the system should be the nodes in the group with an actual keyrange.



Covered	Actual
0 - 64	24 - 31

Figure 6: The routing cache of a client for a system with a keyspace of 64 when it knows only about the one group

At system startup the covered keyrange will be the entire keyrange of the system as shown in figure 6 as the client only know about one node.



Covered	Actual
0 - 23	16 - 23
24 - 31	24 - 31
32 - 43	32 - 39
39 - 63	48 - 55

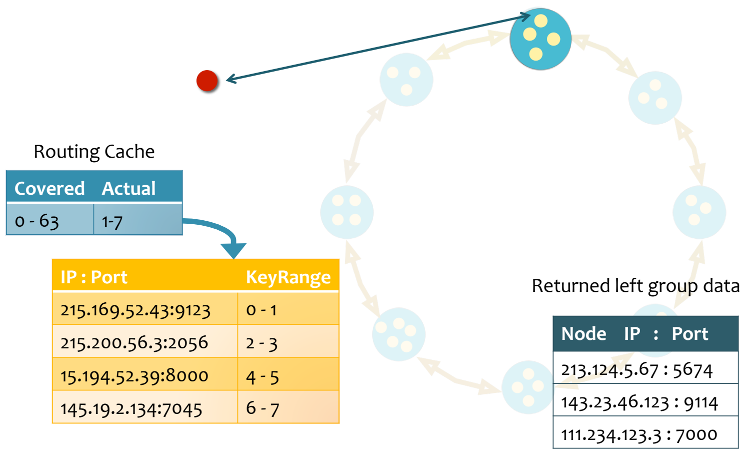
Figure 7: The routing cache of a client for a system with a keyspace of 64 when it knows partial information about the groups in the system

As the client learns about more groups it will adjust its covered keyranges as shown in figure 7. The covered keyrange for a group must always cover its actual keyrange as routing to a group directly will always be faster than routing through additional groups. The remainder of the keyrange that is not contained in actual groups will be split evenly between the groups that are closest to it. This means that there will likely be covered ranges of vastly different sizes.

4.2 Example Lookups

In this section we will give some concrete examples of how the cache works.

4.2.1 Filling the Cache



Covered	Actual
0 - 63	1-7

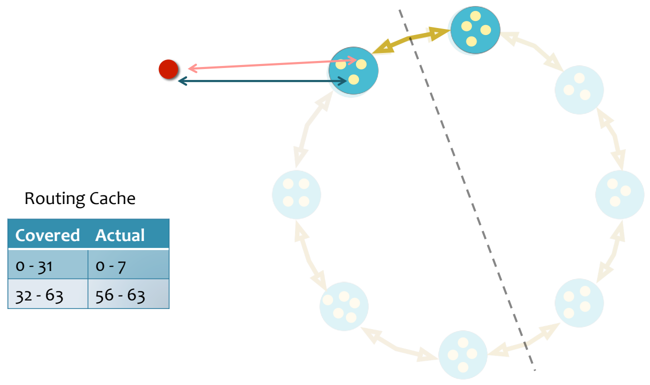
IP : Port	KeyRange
215.169.52.43:9123	0 - 1
215.200.56.3:2056	2 - 3
15.194.52.39:8000	4 - 5
145.19.2.134:7045	6 - 7

Node IP : Port
213.124.5.67 : 5674
143.23.46.123 : 9114
111.234.123.3 : 7000

Figure 8: Executing a lookup for key 42 in a system with a keyspace of 64 (step 1)

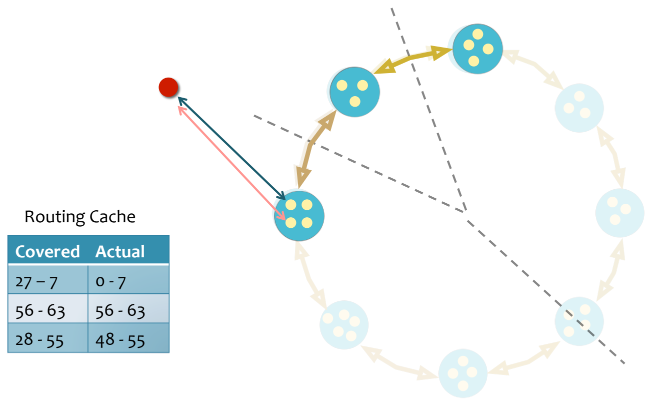
First let us lookup the key 42 as we did in section 3.3 for the original linked-list system. When the client first connects to the system it will send a *group info* message to the only node it knows about as illustrated by figure 8. This will return the addresses and keyranges of the nodes in the group which the client will then add to its cache. After sending a *group info* message the client will find that the returned actual range does not have 42 in it. It will then figure out which direction from this information is closest to 42. For the system pictured above the closest direction will be to the left. The client will then send a *left group info* message to a random node in the group. This node will return the addresses of the nodes to the left. These addresses will not be added to the cache as the client does not yet know their keyranges.

Now that we know the addresses for another group we can repeat the same thing until we find a node with the keyspace we are looking for. On every *get info* we update the cache more. This process can be seen in figures 9, 10 and 11.



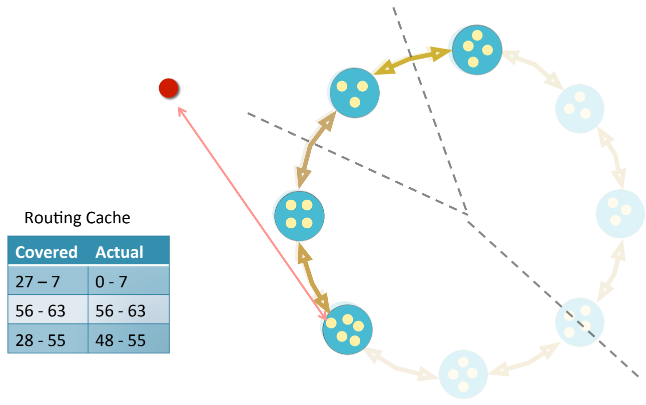
Covered	Actual
0 - 31	0 - 7
32 - 63	56 - 63

Figure 9: Executing a lookup for key 42 in a system with a keyspace of 64 (step 2)



Covered	Actual
27 – 7	0 - 7
56 - 63	56 - 63
28 - 55	48 - 55

Figure 10: Executing a lookup for key 42 in a system with a keyspace of 64 (step 3)

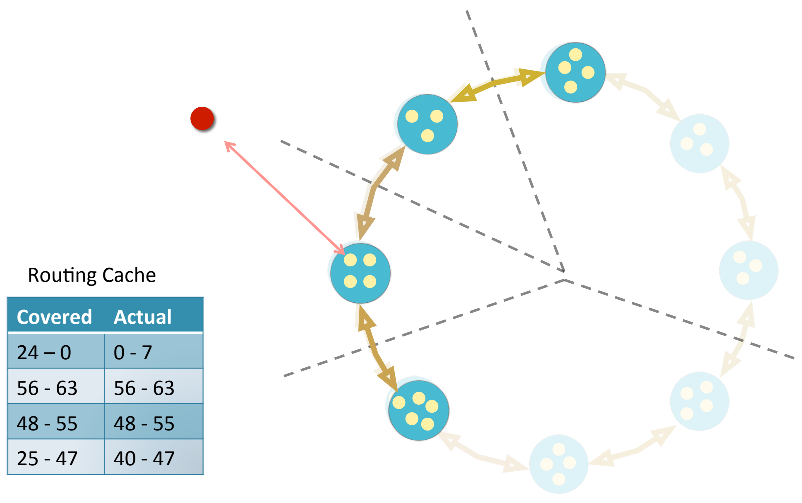


Covered	Actual
27 – 7	0 - 7
56 - 63	56 - 63
28 - 55	48 - 55

Figure 11: Executing a lookup for key 42 in a system with a keyspace of 64 (step 4)

4.2.2 Looking up a cached value

Once we have built up the cache the client executes a lookup just as before, by checking to see what the closest cache entry is to the key that it is looking for. If the client is looking up a key that is in the actual keyrange of a group that is in the routing cache than the client can route directly to the node that has this key. Figure 12 illustrates this situation.

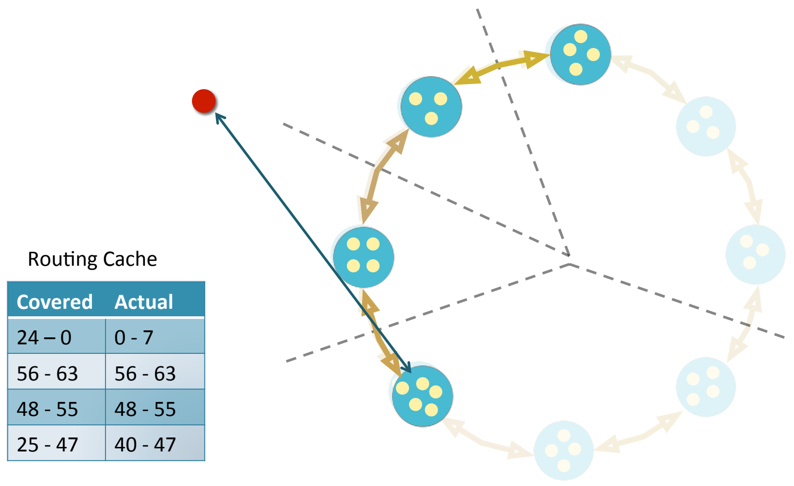


Covered	Actual
24 – 0	0 - 7
56 - 63	56 - 63
48 - 55	48 - 55
25 - 47	40 - 47

Figure 12: Executing a lookup for key 54 in a system with a keyspace of 64

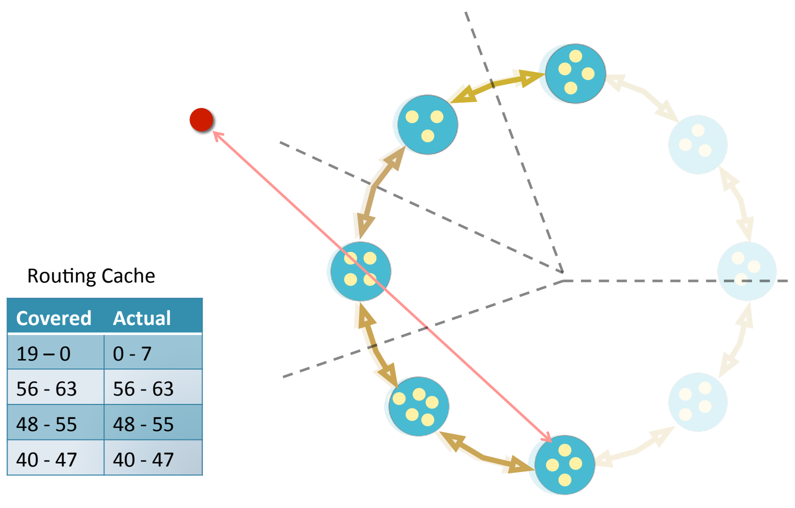
4.2.3 Looking up a non-cached value

The client described above would now like to lookup key 38. Upon looking up 38 in the cache it finds that it is in the covered range $25 - 47$ but not in the actual range for the node that this range maps to. The client can now send a $left\ group\ info$ message directly to this node instead of starting over at its initial connection point like the linked-list routing. As the client completes this lookup and learns about new groups the data will be added to the cache, just as before. Figures 13 and 14 illustrate this scenario.



Covered	Actual
24 – 0	0 - 7
56 - 63	56 - 63
48 - 55	48 - 55
25 - 47	40 - 47

Figure 13: Executing a lookup for key 38 in a system with a keyspace of 64 (step 1)



Covered	Actual
19 – 0	0 - 7
56 - 63	56 - 63
48 - 55	48 - 55
40 - 47	40 - 47

Figure 14: Executing a lookup for key 38 in a system with a keyspace of 64 (step 2)

4.3 Incorrect Data in the Cache

After the initial access routing state will accumulate. To find a group, a client chooses the IP address in its table with the range nearest to the one of the group it is searching for and routes to it. If this is not the group the client is looking for then it follows the successor and predecessor pointers to the group with the range it wants. In the process it adds new groups IP addresses to its table. If the client does not receive a response from an IP address it tries to route to then it assumes that the IP address is no longer in the system. In this case the client will remove the IP address from its table and route to the next closest IP address. If the client succeeds in routing to a group and finds that this group does not have the address range that it thought the IP address had then it will need to route to another IP address. If the keyspace of this IP address is owned by the same group as the keyspace that it was labeled with in the routing table then the client will follow the successor and predecessor links from this group to the one it is looking for. Otherwise, the client will try routing to the next closest IP address in its table. In both cases it will update its routing table with the new keyrange belonging to the IP address.

Harmony guarantees that the majority of the nodes in each group know the correct information at all times. Therefore it is possible to get information from a node with old information about the system. This old information will be consistent with the state of the system at some previous time. Our system will still work fine with this incorrect data. It is very likely that at least some of the addresses and keyranges returned are correct. We can try nodes until we find one that works. We decided it was unnecessary to ping a majority of nodes and in the process learn the most current state as the routing information is only a cache of hints. We will still have enough information to be correct most of the time without getting the most up-to-date information. Our reasoning was sending $get\ info$, $get\ right\ info$ and $get\ left\ info$ messages to entire groups would put a lot more messages in the system. Sending messages to entire groups might actually speed up our performance as it would ensure we were always routing to the correct node. In our current system we can route to a node that is known to the majority of nodes in the group to be incorrect. Even if we made sure to get correct information, if there is a lot of churn it is likely that some of the data will be out of date by the time we use.

5 Cache Maintenance

Nodes in all distributed systems churn. Churn can happen for many reasons, nodes connecting and disconnecting from a P2P system, computers failing, adding new machines to a system and many other reasons. It is impossible to predict and avoid all of these. Regardless, distributed systems should remain quick and correct in this ever changing environment.

When there is churn in the system some of the client's cached addresses will become invalid. These failed node addresses need to be cleared out of the client's cache so that the client does not try to route to them and have to wait for a timeout. Some nodes may remain in the system for a long time so it would not be efficient to invalidate all routing state after a span of time. We have come up with a smarter system to preemptively detect and correct this out of date data.

In order to keep the cache up-to-date we have developed what we will refer to as a pinging strategy. This strategy periodically sends a $get\ info$ message to addresses in the cache to check whether they are still correct. It has its own thread and runs in the background. If it receives a response it replaces the cached information about the group with the response it receives back. This refreshes the information about the whole group, not just the node with the single key. If it times out it executes a lookup for the key starting from the closest node it knows about (other than the one that failed) to the key it is looking for. This will refresh the cache for all groups that must be routed through.

5.1 Who to ping

The ping messages are sent to the node with the key that is looked up the most frequently. This is because this node will provide the biggest speed benefit if it is kept up-to-date. This strategy is advantageous if the client mainly accesses a single or small range of keys clustered about a point in the keyspace. It is not so good if the client's accesses are distributed randomly over the keyspace. If the accesses are randomly distributed than the pinged key may not be accessed much more than any other key. With evenly distributed inputs it would be more beneficial to ping nodes spaced evenly around the keyspace and not concentrate on any particular node. We have assumed that most lookups can be modeled by a poisson distribution and so will mainly be clustered around a small range.

5.2 When to ping

If there is little churn in the system there is no reason to send a lot of pings as not much will be changing. In this case pinging will only add overhead and create congestion. On the other hand, if there is a lot of churn sending infrequent pings will not be near enough to keep the routing data up-to-date enough to be useful. Therefore the system we implemented dynamically adjusts the times it pings depending on the churn in the system. It calculates the churn by keeping track of the total number of lookups and the total number of timeouts since system startup. It uses these total numbers instead of just the last n so that fluctuations over time will not drastically skew the ratio. It would not be good for there to be 100 successes and then 10 failures and for the system to start pinging at a very high rate when it was only a bubble of errors with the error rate going very low again right after the 10 misses. The downside of this system is if there is no churn for the majority of the life of the system and then suddenly churn is introduced, it will be slower to adapt.

The ping time is lowered whenever there is a miss and increased whenever there is a success. The changes are made in .001 second steps. The highest it can increase to is 1 second and the lowest to 0.001 seconds. These numbers were chosen so that the maximum time between pings would never be longer than the maximum time between lookups. The minimum time was chosen arbitrarily so that pings would not clog the system but if there was a very high rate of churn there would be enough pings to keep the system up to date. If there were no cap and there was no churn for a while then it could get to a very long time between pings. Since the time shifts by a constant rate it would take much longer for it to shift back to a short time if there was suddenly a lot of churn.

6 Evaluation

To evaluate the routing protocols we ran the system we developed with and without pinging and compared it to the linked-list routing. All three were run on a simulator on a cluster of five machines all with 16GB RAM and 2* 4 core 2.0 Xeon HT processors. The simulator uses discrete time performing a single action every time step. When it receives requests for any node in the system it deals with them in a sequential first come first serve basis. It is single threaded and opens up a port for each node in Harmony. There is no extra latency in the system besides the processing of messages, and sending them over the sockets. Experiments were run with 600 nodes, with 8 per group. Harmony merges groups when they get too small and splits them when they get too big. The merge threshold was 6 and the split threshold was 10 for all tests. The time between churn events and the times between lookups were both from a uniform distribution. The keys chosen to lookup were on the uniform distribution for some tests and the poisson for others. The following graphs are compiled from the combined data points gathered from five separate runs of the system.

6.1 No Churn

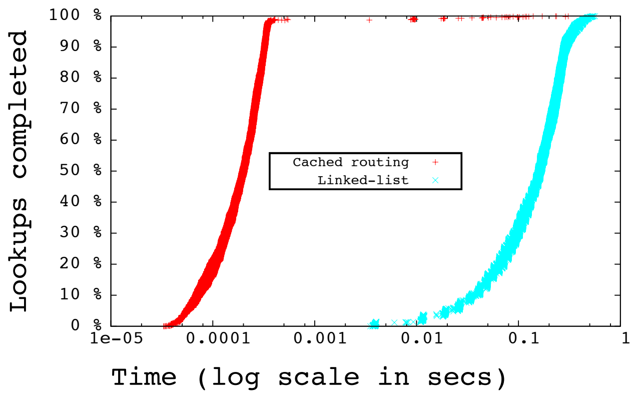


Figure 15: Running with no churn and input keys from the uniform distribution

Figure 15 shows the performance of our caching system as compared to the prior linked-list system. As can be seen from the graph, our system was 100 times faster than the linked-list system for 95% of lookups. This graph shows a clear improvement in performance. It does not contain evaluation for caching with pinging as pinging would not improve the system at all with no churn. The whole point of pinging is to account for churn.

6.2 Churn

The performance of our system with churn is still markedly better than that of the linked-list. However, we now have more extremely long lookups in the system without pinging. One peculiarity is that there are very few lookups between .1 and 10 seconds. This is likely explained by the fact that if more than one node in a group has stale data then it is likely that it has been a long time since that group has been refreshed and so it is likely that some of the other nodes in the group have stale data too.

As can be seen from both figure 16 and 17, the pinging strategy did not increase the performance as we hoped. Instead, contrary to our hypothesis it actually made performance worse for some of the time. Since the ping strategy always pings the most frequently accessed node it makes sense that it would not provide a large benefit when accessing uniformly distributed keys. It should, however, keep most of the required data up-to-date when accessing keys from a poisson distribution. It is apparent from the graph that our hypothesis that pinging would speed up performance for keys from a poisson distribution is incorrect.

We believe that the reason pinging slows our system down is that it adds a lot extra messages to the system. Since the simulator is single threaded it can only deal with one message at a

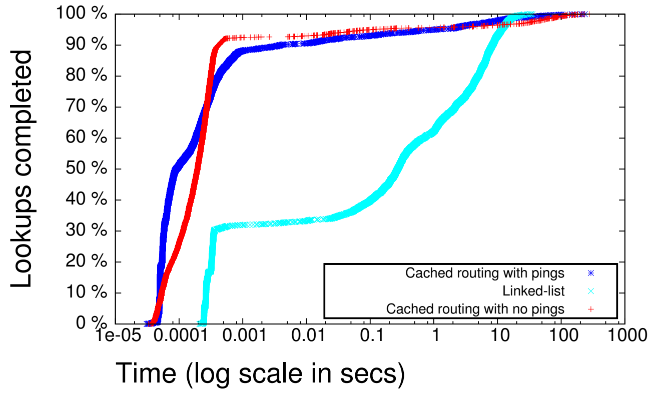


Figure 16: Ten times as much churn as lookups with input keys from the uniform distribution

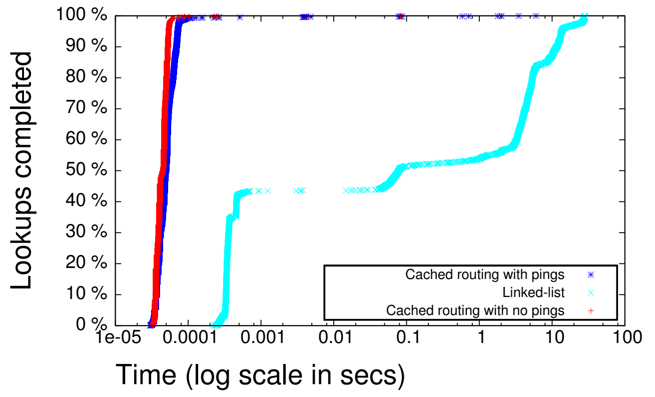


Figure 17: Ten times as much churn as lookups with input keys from the poisson distribution

time and all the other messages have to wait. Our messages have relatively short timeouts and so if they have to wait too long the client will time out and remove the node it sent the message to from its cache. It is possible that if our system were run on Harmony, instead of a simulator, that the ping strategy would provide a speedup since each machine would have its own separate process.

The pinging system improves and adds to the information in the cache, it never makes it worse. The cost of keeping the information up-to-date appears to outweigh the benefits it provides.

6.3 Scalability

Figure 18 displays the improvement in scalability that our routing system has made. Linked-list routing appears to scale at a rate greater than $O(n)$ whilst cached routing appears to scale linearly. Our testing indicates that using cached routing improves the scalability of the routing system when there is no churn.

We wanted to evaluate the scalability of the system with churn but were unable to do so

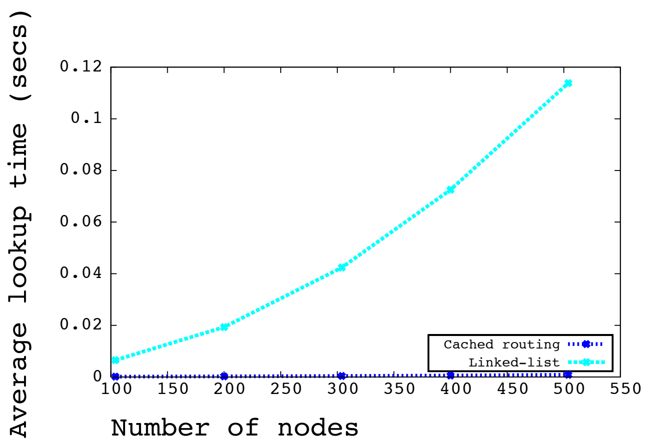


Figure 18: Scalability with no churn

accurately because of restrictions of the simulator. The simulator creates churn by generating a churn event at uniformly distributed intervals. The range can be specified per test. Since the amount of churn per time is specified if the number of nodes in the system changes the probability of each node churning changes. This means that a system of 100 nodes has a much higher churn rate than that of 800 nodes with the same time between churns.

7 Future work

A big disadvantage of of the previously outlined design is that each client has to individually build up its knowledge of the system. If there are multiple clients accessing the system at the same time it would be advantageous for them to be able to help each other by sharing their information about the system topology. A collaborative routing scheme would lead to better routing for all clients.

With an initial setup that is the same as previously described, all routing data is stored on the clients. Allowing the clients to collaborate means that the clients would need to somehow access each other's cached routing data. It does not make sense for them to contact each other directly. In order for clients to contact each other directly they would need to know each other's addresses.

7.1 Collaborative routing by storing routing data in the system

A method we could choose to allow clients to collaborate while preserving their anonymity is to store routing data in the system. On startup a fixed number of keys could be reserved for routing data. When any of these keys are looked up it would return cached routing data, giving the client who accessed it up-to-date information about the system.

When a client first connects to the system it would know about only one node, the node it is connecting to. It would also know a key to lookup to find out routing information for the system. The first thing it could do after connecting is execute a lookup on this key. It would execute this lookup in the same manner as in our current system, accumulating information about the system as it searches for the requested key. When the desired key is found a message would be sent back to the client with the routing data stored at the key. The client would now have data that it discovered by traversing nodes in order to find the key and the routing data that was stored at the key. Some of this data may conflict because of churn that has taken place in between when the addresses were accessed.

To resolve conflicting data it is important to know which piece of data is the newest. Therefore each piece of routing information would be time-stamped. This way the data in the routing cache and data accumulated by a node could be compared and merged, allowing newer data to overwrite older data. Time-stamping may not be entirely accurate as the clocks on each node would not be synced perfectly. This would not matter because the cached data is just a hint for the system, the system would still function correctly without it. If clocks are slightly out of sync the data cached would still be pretty recent even if the cache updates do not happen in quite the right order.

Clients would repeat these steps for every lookup. If a node gets accessed frequently it is likely to have correct routing data in the cache. Therefore the client should be able to access the keys that hold the routing cache in a very small number of hops after the initial connection to the system.

7.1.1 Reading the routing cache

If a client repeatedly accessed the same key it may have more up-to-date routing information about that key than the routing cache. Therefore it would only slow the client down if the client had to repeatedly check the routing cache. A client could only bother to check the routing data in the system if its own data for a value it needed to look up was older than a certain amount. If the data were older and there would be a higher probability that the routing data was stale than it would check the routing data stored in Harmony to try to find a new hint. If the client tried to use their local copy and it failed they could then lookup the routing cache.

7.1.2 Updating the routing cache

When the client merges its cache with the cache stored at a routing key it needs to generate a new cache with the most up-to-date values. This is simple if both caches have the same keys in them. If the caches have different keys then the client could add the both the keys it is missing and the keys the cache is missing to its local cache. This is advantageous as it would give both the stored cache and the client more information. It could be a disadvantage if the client's cache was small and adding new keys forced some old keys out and the client only needed to access the old keys. Another option would be to not add in any new keys. The disadvantage of this would be that it would take clients longer to access new keys.

8 Lessons Learned

I came into this project with no knowledge of distributed systems, the research process, Python, concurrent programming or network programming. Although I was unable to accomplish as much as I would have liked I learned a huge amount from what I was able to finish. I got a lot of practice debugging multithreaded network code. I used UDP ports and came to understand first-hand why people prefer to program with TCP. I also learned a bit about the differences in programming in a statically typed language versus a dynamically typed one. There were type bugs that I needed to learn a new programming mindset to find. I also learned a lot about reading papers and interpreting data. Prior to this project I had little experience with either.

9 Conclusion

We discovered that the ping strategy is not beneficial the way we had assumed. We thought that by adding more information to the cache we would surely speed up our system. It turns out that pinging nodes to keep the cache up-to-date creates so much overhead that it only provides a 10 times speedup for 95% of keys generated on a uniform distribution while caching without pings provides a 100 times speedup. Caching definitely improved system performance.

Acknowledgements

I would like to thank Ivan Beschastnikh, Lisa Glendenning, Arvind Krishnamurthy, Tom Anderson, Daniel Powell, Jeff Rasley and Colin Scott for their feedback and support that greatly helped me complete this thesis.

References

[1] I. Beschastnikh, L. Glendenning, A. Krishnamurthy, and T. Anderson. Harmony: Consistency at scale. Technical Report UW-CSE-10-09-04, `http://www.cs.washington.edu/homes/arvind/harmony.pdf`, Univ. of Washington, 2010.

[2] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *Proc. of SOSP*, Stevenson, Washington, USA, 2007.

[3] P. Maymounkov and D. Mazières. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 53–65, London, UK, 2002. Springer-Verlag.

[4] P. V. Mockapetris. Domain names - concepts and facilities, 1987.

[5] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *Proc. of SIGCOMM*, 2001.

[6] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM TON*, 11(1), 2003.