CMSC 23310 Final Project: Distributed Hash Table

Josh Feingold Henry Riordan Jake Whitaker

Spring 2014

1 Project Overview

For this project we chose to implement a distributed hash table. That is, a given client should be able to set or retrieve values from any node in a given network. We focused on producing a key-value store that conforms to the BASE consistency model.[2] The DHT's designed was inspired by past distributed keystores such as Chord [4], Pastry [3], and Dynamo [1]. As a whole, our design most closely followed that of Dynamo's. We favored many of Dynamo's simpler but effective design choices. Like its historical predecessors, our DHT is utilizes a circular keyspace. Nodes are dynamically assigned responsibility to portions of this keyspace as the system grows and shrinks. Any given node will maintain control of all the keys between it and its 'predecessor' in the keyspace ring.

To reduce message overhead and simplify request routing, each node in Our DHT's maintains a complete routing table. By having knowledge of complete available network topology, much fewer 'hops' are required to deliver any get or set request, reducing overall message count. A discussion of the complexity of various routing methods is discussed in [4, 3]. This decision closely follows the design of Dynamo [1], and makes for quicker response to requests due to the reduction in total messages required. Maintaining the routing table requires slightly more internal overhead and storage as each node needs to keep track of all the other nodes and detect failures. However, for networks that are not massively large this is outweighed by the streamlined request forwarding's speed. In order to track network topology, every node sends a heartbeat message to every other node at a regular interval, and the entries in the routing table contain the timestamp of the most recent heartbeat from each node. Every node regularly checks its routing table, and removes nodes from whom it hasn't received a heartbeat in a given interval.The exiting and entering of nodes to the network - due to fail-stops, partitions, or client choices - can trigger replication and merging events among other nodes in the system, dependent on their position in the keyspace.

All routing table entries have a timestamp that is compared to current system time at regular intervals with a sweeping function. Once the timeout period expires for a given node, it is removed from the routing table and is considered failed. To make others aware of its prescence, a node broadcasts a heartbeat message to all other nodes in the network. Upon receiving a heartbeat a node resets the timeout for the sender in the routing table. If a node receives a heartbeat from a node not in its routing table, it is considered a new node and is added to our routing table.

Our system's primary fault-tolerance schema is based upon replication strategies. Each node holds a replica of the keyspaces of its two ring predecessors within its own storage space. This allows for seamless rollover in the event of failure. In the event that a node fails, the rest of the nodes simply update their routing tables and send necessary replicas to new neighbors. A failure will trigger additional replication to ensure that there are two duplicates, but it will not reduce

overall availability of the network. There may be some latency if a request is made after a node fails but before the routing tables are updated, but the system will always return a response.

The worst case scenario for Our DHT is a partitioning of the network. If a node and its replicas are both on the same side of the partition, then nodes on the other side will be unable to contact them or retrieve messages from them. A partition can be thought of as spliting the keyspace ring into two smaller, less populated rings. Any side of the network partition will thus respond to get requests for those values contained on the other side with an error stating that the key could not be found. If a partition is resolved, Our DHT contains mechanisms for merging re-appropriated keys from old owners to new owners within a given timespan and message count dependant on system size. Our network thus remains available at the cost of consistency. When the network recovers from the partition, a simple merge function is run, and each key is set to the most recent value on either side of the partition. To facilitate this, the nodes store a timestamp with each key-value pair.

2 Implementation

The DHT node is written in Python using the ZMQ python bindings.

2.1 Set

When a node receives a set message from the broker, two possibilities exist: Either it is the owner of the key, or it is not. If it is the owner of the key, it adds or updates that value in its keyspace, and returns a set response to the broker, and sends replica messages to the next two nodes on the ring.

If it does not own the key, it forwards the get message to the correct node, adding a field that indicates itself as the source of the message. Additionally, to deal with the possibility of network partitions or fail-stops, the message is stored in a dictionary, with the key being the message's ID field. If the destination node is removed from the routing table before the sender receives a response, the message is resent to a new destination. When a response is received, the sender removes the message from the queue and sends a set response message back to the broker.

The destination node uses a similar method. The only difference are that the node sends a 'set relay' message to the source node instead of a response to the broker. Furthermore, in the rare event that the message must be forwarded a second time, the node does not update the source field or add it to the pending messages queue. This prevents duplicate responses.

2.2 Get

Get requests are processed almost identically to set requests, with the difference being that the node retrieves the value and returns it to the client, rather than updating it. Get messages are forwarded and stored in the same manner as set messages. The only difference is that, while a set request will always be successful, a get request may return an error in the event that the node cannot communicate with the owner of a key, or any of its replicas.

2.3 Heartbeats

Each node's routing table contains timestamps indicating the time of the last heartbeat from each node. Any nodes whose heartbeats were older than a chosen age are removed from the routing table. To refresh these timestamps, each node sends a heartbeat to the entire network on regular

intervals. These heartbeat messages get sent to every node in the network, rather than every node in the routing table. This full network broadcast allows for the "rediscovery" of nodes in the aftermath of a network partition or a failure. When a node A receives a heartbeat from a node B that is not in its routing table, B is added. If B is A's immediate predecessor, then A may send a merge message to B. If B is A's immediate, or secondary successor, then A must send replica messages to B.

2.4 Replication

Each node replicates the data of its two predecessors. This is achieved by having the owner of the key send the key, value, and timestamp of each item in its keyspace in a replica message. This message can contain a single key, or multiple keys depending on the circumstance in which the message was sent. Firstly, whenever a node sets a value in response to a set message, it sends out a replica message. Secondly, when a node's keystore is changed in response to a merge, it must send a replica of its entire keystore to ensure consistency between replicas.

2.5 Merging

When a node joins an existing network, either from a recovering partition, or just recovering from a fail-stop, a merge is required to integrate it into the network. When a node receives a heartbeat from a node not in its routing table, it checks if the new node is its immediate predecessor. If so, then the node checks if any of the items in its keystore should now belong to the new node, and sends any such items to the new node in a merge message. The new node resolves any conflicts by choosing the most recently set value, and then sends replication messages for the final values.

2.6 Deletion

A node cleans up its keystore when a new node enters the keyspace that it is replicating. This will happen after a mege with the node or a merge with the node predecessor. After this occurs the node then checks all keys in its keystore and removes those that are no longer owned by the nodes we are replicating.

3 Example Scripts and Discussion

3.1 Fail Stop

In this script, Listing 1, we set values across 8 nodes in a network, using 52 keys to ensure that they are distributed across all of the nodes. Then we simulate 4 nodes failing by creating a partition that we do not recover from. We only ask one set of 4 nodes for the values that were set. This script tests that our replication works as expected since we are falling back on the replicas when the nodes fail. We continue to be accessible and consistent in this script. In more dramatic cases we could lose consistency if all of the nodes containing the data were to fail at the exact same time. In real world applications, the nodes physical location could be a factor in what nodes replicate what data if failures of physical systems were a concern.

3.2 Partitioning and Merging

In our partitioning testing script, Listing 2, we start with a partitioned network. Then we set different values to the same keys to both halves of the network. Then we wait to ensure that

the values have propagated and then join the merged networks. Again we wait to ensure that everything has propagated before calling get on each key. This test focused on ensuring that we merged partitions correctly and that our conflict resolution algorithm worked as expected. As expected the values that were set more recently were those that were chosen after merging.

3.3 Multiple Failures

Our last script, Listing 3 was designed to have a multitude of faults. This script was designed to test if we could remain available in the face delays, dropped message, and partitions. While there are times that we have to return an error as the value of a key can be temporarily lost due to failures or partitions, we are always responding to the client and eventually we will have the value unless we lost all three replicas at the same time. Our system did not fail in the face of this intense set of faults which proves its robust nature.

4 Conclusion

Given the flexibility of the project specifications, we must define success for the project ourselves. Our choice of a distributed hash table creates a highly available system that we eventually make consistent. The design is created to be as flexible as possible so it can seamlessly react to and recover from faults with minimal to zero effect on the clients of the service. From our tests we show this very high availability, but we do see some weakness in consistency under extreme levels of faults.

4.1 Issues and Challenges

The primary issue we faced was the idea of eventual consistency, and what it means for the network to be eventually consistent. Prioritizing availability over consistency created a situation where a get request could return an old value, or even no value at all, and still be technically correct. As such, many of our design choices were designed to minimize the likelihood of returning a stale value or an error.

For example, the heartbeat system required some fine tuning, particularly the frequency with which messages were sent. We found that using too small a value caused heartbeats to overwhelm the network and starve out the rest of the messages, while using too great a value caused delays while waiting for nodes to time out.

We initially had the message queues on a separate timer from the heartbeat system. However, this design turned out to be overly complex, and we settled on the idea of sweeping the message queue only when the routing table changed, so as to minimize the number of moving parts.

4.2 Possible Extensions to the Design

A possible extension to this design is to utilize virtual nodes. This allow for greater load balancing as the larger number of nodes spread across the keyspace distributes the keys amount the physical machines more evenly. This allow ensures that when a physical machine fails that its keys are spread among the rest of the physical nodes instead of a physical machine as in our design.

A second extension would be an improved timestamp/ clock design. In our design we rely on the nodes having synchronized clocks and we are not enforcing this in our system at all. While it is possible to use an external service to maintain this synchronization, incorporating it into our own DHT would allow for a stronger guarantee on eventual consistency.

Another extension would be alternative routing designs. If this was to be expanded to more than a hundred or so nodes, a complete routing table becomes cumbersome and less effective. A method similar to Chord [4] or Pastry's [3] more sparse tables that run in $O(log(n))$ due to their design would save memory at the cost of higher message volume.

References

[1] Giuseppe DeCandia et al. "Dynamo: Amazon's Highly Available Key-value Store". In: *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*. SOSP '07. Stevenson, Washington, USA: ACM, 2007, pp. 205–220. ISBN: 978-1-59593-591-5. DOI: 10.1145/1294261.1294281. URL: http://doi.acm.org/10.1145/1294261.1294281.

[2] Armando Fox et al. "Cluster-based Scalable Network Services". In: *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*. SOSP '97. Saint Malo, France: ACM, 1997, pp. 78–91. ISBN: 0-89791-916-5. DOI: 10.1145/268998.266662. URL: http://doi.acm.org/10.1145/268998.266662.

[3] Antony I. T. Rowstron and Peter Druschel. "Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems". In: *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*. Middleware '01. London, UK, UK: Springer-Verlag, 2001, pp. 329–350. ISBN: 3-540-42800-3. URL: http://dl.acm.org/citation.cfm?id=646591.697650.

[4] Ion Stoica et al. "Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications". In: *IEEE/ACM Trans. Netw.* 11.1 (Feb. 2003), pp. 17–32. ISSN: 1063-6692. DOI: 10.1109/TNET.2002.808407. URL: http://dx.doi.org/10.1109/TNET.2002.808407.

A Test Scripts Source

Listing 1: Fail Stop Test Script

```
start Alpha --peer-names Beta,Gamma,Omega,Able,Baker,Charlie,Delta
start Beta --peer-names Alpha,Gamma,Omega,Able,Baker,Charlie,Delta
start Gamma --peer-names Alpha,Beta,Omega,Able,Baker,Charlie,Delta
start Omega --peer-names Alpha,Beta,Gamma,Able,Baker,Charlie,Delta
start Able --peer-names Alpha,Beta,Gamma,Omega,Baker,Charlie,Delta
start Baker --peer-names Alpha,Beta,Gamma,Omega,Able,Charlie,Delta
start Charlie --peer-names Alpha,Beta,Gamma,Omega,Able,Baker,Delta
start Delta --peer-names Alpha,Beta,Gamma,Omega,Able,Baker,Charlie
set A 0
set B 1
set C 2
set D 3
set E 4
set F 5
set G 6
set H 7
set I 8
set J 9
set K 10
set L 11
set M 12
set N 13
set O 14
set P 15
set Q 16
set R 17
set S 18
set T 19
set U 20
set V 21
set W 22
set X 23
set Y 24
set Z 25
set a 26
set b 27
set c 28
set d 29
set e 30
set f 31
set g 32
set h 33
set i 34
set j 35
```

```
set k 36
set l 37
set m 38
set n 39
set o 40
set p 41
set q 42
set r 43
set s 44
set t 45
set u 46
set v 47
set w 48
set x 49
set y 50
set z 51
after 300{
split part1 Able,Baker,Charlie,Delta
}
after 1000{
get Able A
get Baker B
get Charlie C
get Delta D
get Able E
get Baker F
get Charlie G
get Delta H
get Able I
get Baker J
get Charlie K
get Delta L
get Able M
get Baker N
get Charlie O
get Delta P
get Able Q
get Baker R
get Charlie S
get Delta T
get Able U
get Baker V
get Charlie W
get Delta X
get Able Y
get Baker Z
get Charlie a
get Delta b
```

```
get Able c
get Baker d
get Charlie e
get Delta f
get Able g
get Baker h
get Charlie i
get Delta j
get Able k
get Baker l
get Charlie m
get Delta n
get Able o
get Baker p
get Charlie q
get Delta r
get Able s
get Baker t
get Charlie u
get Delta v
get Able w
get Baker x
get Charlie y
get Delta z
}
```

Listing 2: Partition Merge Test Script

```
start Alpha --peer-names Beta,Gamma,Omega,Able,Baker,Charlie,Delta
start Beta --peer-names Alpha,Gamma,Omega,Able,Baker,Charlie,Delta
start Gamma --peer-names Alpha,Beta,Omega,Able,Baker,Charlie,Delta
start Omega --peer-names Alpha,Beta,Gamma,Able,Baker,Charlie,Delta
start Able --peer-names Alpha,Beta,Gamma,Omega,Baker,Charlie,Delta
start Baker --peer-names Alpha,Beta,Gamma,Omega,Able,Charlie,Delta
start Charlie --peer-names Alpha,Beta,Gamma,Omega,Able,Baker,Delta
start Delta --peer-names Alpha,Beta,Gamma,Omega,Able,Baker,Charlie
split part1 Alpha,Beta,Gamma,Delta
#split part2 Able,Baker,Charlie,Omega # set the values on one half
   of the network
set Alpha a 11
set Baker b 22
set Gamma c 13
set Omega d 14
set Alpha e 15
set Beta f 16
set Gamma g 17
set Omega h 18
set Able a 21
```

```
set Beta b 12
set Charlie c 23
set Delta d 24
set Able e 25
set Baker f 26
set Charlie g 27
set Delta h 28
join part1
after 200 {
get a
get b
get c
get d
get e
get f
get g
get h
}
```

Listing 3: Multiple Failures Test Script

```
start Alpha --peer-names Beta,Gamma,Omega,Able,Baker,Charlie,Delta
start Beta --peer-names Alpha,Gamma,Omega,Able,Baker,Charlie,Delta
start Gamma --peer-names Alpha,Beta,Omega,Able,Baker,Charlie,Delta
start Omega --peer-names Alpha,Beta,Gamma,Able,Baker,Charlie,Delta
start Able --peer-names Alpha,Beta,Gamma,Omega,Baker,Charlie,Delta
start Baker --peer-names Alpha,Beta,Gamma,Omega,Able,Charlie,Delta
start Charlie --peer-names Alpha,Beta,Gamma,Omega,Able,Baker,Delta
start Delta --peer-names Alpha,Beta,Gamma,Omega,Able,Baker,Charlie
set A 0
set B 1
set C 2
set D 3
set E 4
set F 5
set G 6
set H 7
set I 8
set J 9
set K 10
set L 11
delay 50 by 10
set M 12
set N 13
set O 14
set P 15
set Q 16
set R 17
```

```
set S 18
set T 19
set U 20
set V 21
drop 400 Baker
set W 22
set X 23
set Y 24
set Z 25
set a 26
set b 27
set c 28
set d 29
set e 30
delay 10 by 40
set f 31
set g 32
set h 33
set i 34
set j 35
set k 36
set l 37
set m 38
set n 39
set o 40
drop 250 Gamma
set p 41
set q 42
set r 43
set s 44
set t 45
delay 19 by 9
set u 46
set v 47
set w 48
set x 49
set y 50
set z 51
split part1 Able,Baker,Charlie,Delta
get A
get B
get C
get D
get E
get F
get G
get H
get I
```

```
get J
delay 26 by 11
get K
get L
get M
get N
get O
drop 300 Able
get P
get Q
get R
get S
get T
get U
get V
get W
get X
get Y
get Z
get a
drop 400 Alpha
get b
get c
get d
get e
get f
get g
get h
get i
get j
get k
get l
delay 10 by 30
get m
get n
get o
drop 300 Beta
get p
get q
get r
get s
get t
get u
get v
get w
get x
get y
get z
```

```
set AA 100
set AB 101
set AC 102
set AD 103
set AE 104
set AF 105
set AG 106
set AH 107
set AI 108
delay 5 by 15
set AJ 109
set AK 110
set AL 111
set AM 112
set AN 113
set AO 114
set AP 115
set AQ 116
set AR 117
set AS 118
set AT 119
set AU 120
set AV 121
set AW 122
set AX 123
set AY 124
set AZ 125
join part1
get A
get B
get C
get D
get E
get F
get G
get H
get I
get J
delay 26 by 11
get K
get L
get M
get N
get O
drop 300 Able
get P
get Q
get R
```

```
get S
get T
get U
get V
get W
get X
get Y
get Z
get a
drop 400 Alpha
get b
get c
get d
get e
get f
get g
get h
get i
get j
get k
get l
delay 10 by 30
get m
get n
get o
drop 300 Beta
get p
get q
get r
get s
get t
get u
get v
get w
get x
get y
get z
get AA
get AB
get AC
get AD
get AE
get AF
get AG
get AH
get AI
delay 5 by 15
get AJ
```

```
get AK
get AL
get AM
get AN
get AO
get AP
get AQ
get AR
get AS
get AT
get AU
get AV
get AW
get AX
get AY
get AZ
```