

# CMSC 23310 Final Project: Distributed Hash Table

Josh Feingold

Henry Riordan

Jake Whitaker

Spring 2014

## 1 Project Overview

For this project we have decided on implementing a distributed hash table for our keystore. We focused on producing a key-value store that fits in the BASE model.[2] Our design of the distributed hash table came from Chord [4], Pastry [3], and Dynamo [1]. Overall our design followed that of Dynamo the closest as we liked some of the simple design choices. Our DHT like the others is based on a circular keyspace on which we place our nodes. These nodes will be in control of all keys between it and its predecessor. Many DHTs use virtual nodes to load balance and distribute replicas more evenly. Virtual nodes are not implemented in this design to reduce complexity.

The DHT's routing table is a complete routing table. This removed the complexity of the various routing methods discussed in [4, 3]. This follows the design of Dynamo [1], it makes for a quicker response to requests due to the reduction to total messages required. Maintaining the routing table is slightly more complex as each node needs to keep track of all the other nodes and detect failures. 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 an interval. Nodes can trigger replication and merging events by joining, dependent on their position in the keyspace.

(  
All routing table entries have a timestamp that we compare to current time with a sweep function. Once the timeout period expires the node is removed from the routing table and is considered failed. To keep to all other nodes it sends a heartbeat message out to the entire network. When a node receives a heartbeat it 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 it is added to our routing table. Nodes joining can also trigger replication and merging events depending on their position in the keyspace.

)  
Our system uses replication for fault tolerance. Each node replicates the keyspaces of its two predecessors in its own keyspace. 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. A failure will trigger additional replication to ensure that there are two duplicates, but it will not reduce availability. 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.

(Our main fault tolerance is a set of two replicas. Each node replicates the keyspace of its two predecessors. The replica is stored in the same keystore as the nodes main data. This design allows for a node to fail and by simply having each node update its routing table the entire DHT is

completely functional. A failure will trigger more replication to ensure two duplicates, but it will not reduce availability. There might be some additional latency if you make a request right after a node fails as the routing table update will take some time, however a response will be returned.)

Our worst case scenario 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 retrieve them. The far side of the network will respond to get requests for those values with an error stating that the key could not be found. Our network 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.

(The worst case in for our fault tolerance is when a network partition occurs. If a node and its two replicas are in one half of the network, the other half will not know those values anymore. It will respond to get requests with an error stating that key is not set in the DHT. We will remain available with this method, just not consistent. When the network recovers from a partition a simple merge function is run and we choose the value that was set the latest using a timestamp that is stored with the 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 merge 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

## 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

```

1 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
6 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
11 set C 2
set D 3
set E 4
set F 5
set G 6
16 set H 7
set I 8
set J 9
set K 10
set L 11
21 set M 12
set N 13
set O 14
set P 15
set Q 16
26 set R 17
set S 18
set T 19
set U 20
set V 21
31 set W 22
set X 23
set Y 24
set Z 25
set a 26
36 set b 27
set c 28
set d 29
set e 30
set f 31
41 set g 32
set h 33
set i 34
set j 35

```

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

```

get Able c
get Baker d
get Charlie e
96 get Delta f
get Able g
get Baker h
get Charlie i
get Delta j
101 get Able k
get Baker l
get Charlie m
get Delta n
get Able o
106 get Baker p
get Charlie q
get Delta r
get Able s
get Baker t
111 get Charlie u
get Delta v
get Able w
get Baker x
get Charlie y
116 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
3 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
8 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
13 set Gamma c 13
set Omega d 14
set Alpha e 15
set Beta f 16
set Gamma g 17
18 set Omega h 18
set Able a 21

```



```

set Beta b 12
set Charlie c 23
set Delta d 24
23 set Able e 25
set Baker f 26
set Charlie g 27
set Delta h 28
join part1
28 after 200 {
get a
get b
get c
get d
33 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
3 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
8 start Delta --peer-names Alpha,Beta,Gamma,Omega,Able,Baker,Charlie
set A 0
set B 1
set C 2
set D 3
13 set E 4
set F 5
set G 6
set H 7
set I 8
18 set J 9
set K 10
set L 11
delay 50 by 10
set M 12
23 set N 13
set O 14
set P 15
set Q 16
set R 17

```

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

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

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

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

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