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 Get

When a node is sent a get message it first checks if it is the node that holds that key. If it holds that key, then it retrieves the value and returns a getResponse message. If it would hold that key, but the value is not set, an error message is returned. When a node receives a get message for a key it does not hold, it looks up the correct node using the complete routing table and then forwards the message to the correct node. The second node returns a value or error message to the first node who then responds to the client requesting the value.

2.2 Set

The set function is almost identical to that of the get except instead of just retrieving the value it sets the value and sends back a confirmation message to the client. Nodes forward messages in the same manner as in get when they receives messages that should go to different nodes.

2.3 Heartbeats

Each node's routing table contains timestamps that are used to remove nodes when the entries are too old. In order to refresh these timestamps each node sends a heartbeat message to the entire network of nodes. These get set to the entire network, not just the nodes in the its routing table. This full network broadcast allows this system to use heartbeats to detect when a node rejoins the network. When a node receives a heartbeat message from a node it does not have in its routing table, it adds the node and then checks if it needs to adjust its replication and if it needs to merge part of its keystore with the new nodes keystore.

2.4 Replication

Our nodes replicate the data of their two predecessors. The replication of data achieved by having the owner of the key send the key, value, and timestamp to the its two successors in a replica message. This message can contain a single key or multiple keys as replica messages are sent in two distinct cases. First, whenever a node sets a value due to a set message being received it sends out a replica message with that key's data to the nodes replicating it. The second case is when a nodes keystore is changed due to a merge of if its replicas have changed due to nodes entering or leaving the network. In this case we send the entire keystore in a message to our replicas to ensure that they are completely up to date with our information.

2.5 Merging

When a node joins an existing network either from a partition than recovers or just recovering from a fail stop, a merge is required to integrate it into the network. When a node receives a heart beat message from a node not in its routing table, it checks if the new nodes successor is itself. If this is true then the old node must merge some of its keystore with the new node. The node giving up part of its keyspace sends a message containing all the key value pairs that exist in that space to the new node. The new node compares those values with the ones already in its space, if there is a conflict, the value with the most recent time stamp is the value that is kept. After the merge is completed, the new node sends replication messages.

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

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

```
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
}
```