Report 5 - Chordy: A Distributed Hash Table

Lorenzo Deflorian

October 8, 2025

1 Introduction

The main goal of this assignment was to implement a distributed hash table (DHT) using the Chord protocol. The DHT supports basic operations such as adding and looking up key-value pairs, as well as handling the dynamic joining and leaving of nodes in the network. The implementation also includes replication of data to ensure that the ring remains fault tolerant under the assumption that at most one node can fail at a time.

2 Main problems and solutions

2.1 The Lord of Ring

After implementig the basic functionality of the DTH I created a basic test script that creates on one side 4 erlang nodes, where each node will host two DHT nodes, and on the other side 4 more erlang nodes, where each node will host a client that will interact with the DTH.

Each client will add 4000 key-value paris to the DTH and then we will check that all the keys are present in the DTH.

The ring.sh script will create our ring while the test.sh script will run the test.

2.2 First steps

Running the test with all the clients interacting with a single DTH node showed that the basic functionality of the DTH was working correctly. All the keys were added and then looked up correctly. The looking up time in milliseconds for looking up 1000 keys each was the following:

Client	Lookup Time (ms)
1	887
2	857
3	883
4	887

Table 1: Lookup times (ms) for 1000 keys per client

What if we chose a random node for each client? Runninf the same test but this time each client will interact with a random DTH node we get the following results:

Client	Lookup Time (ms)
1	786
2	736
3	826
4	790

Table 2: Lookup times (ms) for 1000 keys per client (random DHT node)

The average lookup time dropped from 878.5ms to 784.5ms! Not a huge difference but still a difference. This is probably due to the fact that with multiple nodes the requests are distributed among the nodes and this reduces the load on each node.

2.3 Fact checking

But are we actually gaining some performance? To answer this question I run the same client setup but this time all the clients were interacting with a single DTH node. SInce the DTH has to handle all the requests simultaneously the performance drops significanlty! The add operations are now in charge of a single node and while it would me buch faster just to forward the request to the node that is responsible for the key, the single node has to handle all the requests and this creates a bottleneck.

After taking more than 15 minutes to add all the keys and then perform the lookup up we can definitly say that we have a performance boost when using multiple nodes.

2.4 I am speed

Before moving on to the other functionalities let's do a final test were each client will try to add 10000 keys to the DTH. Also in this case client will interact with a random DTH node. The results are the following:

Client	Add Time (ms)
1	7733
2	6470
3	6418
4	6214

Table 3: Add times (ms) for 10000 keys per client (random DHT node)

Our DHT is pretty fast! The average time to add 10000 keys is 6708.75ms, which is pretty good considering that each client is adding 10000 keys.

2.5 Satisfied or Refunded

Our DTH has so many functionalitie: We can send a probe to see how many keys we have stored:

```
Probe completed in 1510 microseconds. Nodes in ring:
    [<9388.94.0>,<9388.95.0>,<9135.90.0>,<0.95.0>,<9135.89.0>,<9389.94.0>,<9389.95.0>,<0.9
Number of nodes: 8
Total number of keys in ring: 4000
Node: <9388.94.0>, Keys: 522
Node: <9388.95.0>, Keys: 568
Node: <9135.90.0>, Keys: 629
Node: <0.95.0>, Keys: 918
Node: <9135.89.0>, Keys: 181
Node: <9389.94.0>, Keys: 140
Node: <9389.95.0>, Keys: 101
Node: <0.94.0>, Keys: 941
```

We can also perform an handover when new nodes joins the ring: We add another two nodes grape1 and grape2 and we see the handover sending another probe.

```
Probe completed in 1783 microseconds. Nodes in ring:
    [<9388.94.0>,<9388.95.0>,<9135.90.0>,<0.95.0>,<9135.89.0>,<9389.122.0>,<9389.94.0>,<93
Number of nodes: 10
Total number of keys in ring: 4000
Node: <9388.94.0>, Keys: 522
Node: <9388.95.0>, Keys: 568
Node: <9135.90.0>, Keys: 629
Node: <0.95.0>, Keys: 918
Node: <9135.89.0>, Keys: 181
Node: <9389.122.0>, Keys: 67 <--- New node
Node: <9389.94.0>, Keys: 73
Node: <9389.95.0>, Keys: 101
Node: <9389.123.0>, Keys: 383 <--- New node
Node: <0.94.0>, Keys: 558
```

As we can see ¡9389.94.0¿ and ¡0.94.0¿ have handed over some of their keys to the new nodes ¡9389.122.0¿ and ¡9389.123.0¿.

2.6 Beyond death

Our DHT can now handle both node joins and unexpected node departures, ensuring the ring remains robust and connected. By maintaining knowledge of each node's successor and its successor's successor (stored as `Next`), the system can quickly recover from single node failures. The stabilization process has been enhanced to propagate this information, so every node is aware of its immediate backup route in the ring.

When a node detects that its successor has failed (using Erlang's process monitoring and the `'DOWN'` message), it promotes the `Next` node to be its new successor and immediately triggers stabilization to repair the ring structure. If a predecessor fails, the node resets its predecessor to `nil` and relies on stabilization to reconnect. This mechanism allows the DHT to maintain availability and consistency, even in the presence of failures, thus achieving the desired fault tolerance for the system.

Like in the previous paragraph let's send a probe before and after killing a node:

```
(pear@DORORO)2> Node <0.95.0>: Got a probe from 162419524
Probe completed in 1702 microseconds. Nodes in ring:
    [<9381.94.0>,<9382.95.0>,<9135.87.0>,<0.95.0>,<9381.95.0>,<9382.94.0>,<9135.88.0>,<0.9
Number of nodes: 8
Total number of keys in ring: 4000
Node: <9381.94.0>, Keys: 1206
Node: <9382.95.0>, Keys: 347
Node: <9135.87.0>, Keys: 136
Node: <0.95.0>, Keys: 115
Node: <9381.95.0>, Keys: 354
Node: <9382.94.0>, Keys: 362
Node: <9135.88.0>, Keys: 1348
Node: <0.94.0>, Keys: 132
```

Now let's kill one node:

```
ring:kill(pear1).
Node 162419524: I'm being forced to die
die
```

```
Probe completed in 1994 microseconds. Nodes in ring:
    [<9379.95.0>,<9378.95.0>,<9362.94.0>,<0.88.0>,<9378.94.0>,<9362.95.0>,<0.87.0>]
Number of nodes: 7
Total number of keys in ring: 3868
Node: <9379.95.0>, Keys: 115
Node: <9378.95.0>, Keys: 354
Node: <9362.94.0>, Keys: 362
Node: <0.88.0>, Keys: 1348
Node: <9378.94.0>, Keys: 1206
Node: <9362.95.0>, Keys: 347
Node: <0.87.0>, Keys: 136
```

The ring is still connected but we have lost some keys since the node that died was the only one storing them. We are gonna change that now!

2.7 The age of the replicants

To prevent data loss when nodes fail, we implemented a replication scheme where each node maintains two separate stores: its own `Store` containing the keys it is responsible for, and a `Replica` containing a copy of its predecessor's store. Whenever a node adds a key-value pair to its local store, it also forwards a {`replicate, Key, Value`} message to its successor, ensuring that every key is stored on at least two consecutive nodes in the ring. This simple yet effective approach allows the system to tolerate single node failures: when a node crashes, its successor automatically merges the replica into its own store and takes over responsibility for all the lost keys. The replication data is also properly maintained during node joins through the handover mechanism, ensuring that new nodes receive both the appropriate store partition and replica data from their neighbors.

We are gonna demonstrate the replication by doing the same test as before, but this time we'll see how the data won't be lost when a node fails.

```
Probe completed in 2440 microseconds. Nodes in ring:
    [<9388.95.0>,<9135.90.0>,<9135.89.0>,<0.94.0>,<9388.94.0>,<9389.94.0>,<9389.95.0>,<0.9
Number of nodes: 8
Total number of keys in ring: 4000
Total number of replica keys in ring: 4000
Node: <9388.95.0>, Keys: 53, Replica: 18
Node: <9135.90.0>, Keys: 252, Replica: 53
Node: <9135.89.0>, Keys: 247, Replica: 252
Node: <0.94.0>, Keys: 737, Replica: 247
Node: <9388.94.0>, Keys: 384, Replica: 737
Node: <9389.94.0>, Keys: 2046, Replica: 384
Node: <9389.95.0>, Keys: 263, Replica: 2046
Node: <0.95.0>, Keys: 18, Replica: 263
```

Now let's kill one node:

```
(pear@DORORO)4> Probe completed in 1615 microseconds. Nodes in
    ring:
    [<9388.94.0>,<9389.94.0>,<9389.95.0>,<9388.95.0>,<9135.90.0>,<9135.89.0>,<0.94.0>]
Number of nodes: 7
Total number of keys in ring: 4000
Total number of replica keys in ring: 4000
Node: <9388.94.0>, Keys: 384, Replica: 737
Node: <9389.94.0>, Keys: 2046, Replica: 384
Node: <9389.95.0>, Keys: 263, Replica: 2046
Node: <9388.95.0>, Keys: 71, Replica: 263
Node: <9135.90.0>, Keys: 252, Replica: 71
Node: <9135.89.0>, Keys: 247, Replica: 252
Node: <0.94.0>, Keys: 737, Replica: 247
```

As we can see the total number of keys in the ring is still 4000! No data loss!

3 Conclusions

This assignment successfully demonstrates the implementation of a distributed hash table using the Chord protocol, achieving scalability, fault tolerance, and efficient data distribution. The experimental results clearly show the benefits of distributing requests across multiple nodes—lookup times improved when clients accessed random nodes rather than a single bottlenecked node, and the system handled 40000 concurrent key additions in under 8 seconds. The handover mechanism correctly redistributed keys when new nodes joined, maintaining the total key count while rebalancing the load across the ring.

The implementation of replication, where each node maintains a copy of its predecessor's data, proved highly effective for fault tolerance. Without replication, node failures caused significant data loss (4000 keys dropped to 3868), but with replication enabled, all 4000 keys were preserved even after node crashes. The stabilization protocol, enhanced with the `Next` pointer tracking each node's successor's successor, enabled rapid recovery from failures through Erlang's process monitoring. Overall, this assignment provided valuable hands-on experience with fundamental distributed systems challenges: maintaining consistency during dynamic membership changes, achieving fault tolerance through replication, and balancing performance with reliability in a peer-to-peer environment.