HW5 : Chordy - A Distributed hash table

[redacted]

October 9, 2024

1 Introduction

This project involved implementing a distributed hash table (DHT) based on the Chord protocol. Through this process, we learned about distributed systems, message-passing mechanisms, and ensuring consistency across nodes in a distributed ring. The main idea is that each node is responsible for a specific range of keys, and a key is stored at its successor node (the next node on the ring that is responsible for that key).

2 Key functions

Stabilize Function

The stabilize function maintains the ring's consistency. Each node periodically checks if its successor has a different predecessor, which might indicate a new node joining the ring. The node adjusts its successor pointer if needed, ensuring that the ring structure remains correct. This prevents data from being lost or routed incorrectly when new nodes join.
The snapshot shows the logic:

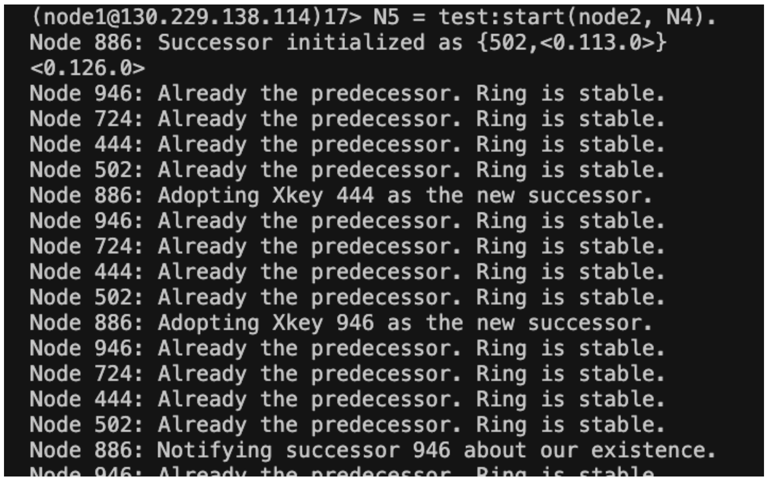
Node 886: Successor initialized as {502,<0.113.0>} — Node 886 has joined the ring and has initialized its successor as Node 502.

Node 886: Adopting Xkey 444 as the new successor. — Node 886 recognizes that Node 444 should be its successor based on their positions in the identifier space.

Node 886: Adopting Xkey 946 as the new successor. — because Node 946 is closer in the identifier space or due to stabilization logic.

The repeated messages like Node 444: Already the predecessor. Ring is stable indicate that the nodes are verifying their positions in the ring.

Node 886: Notifying successor 946 about our existence. — This shows that Node 886 is notifying its successor (Node 946) to ensure that Node 946 updates its view of the ring.



```
(node1@130.229.138.114)17> N5 = test:start(node2, N4).
Node 886: Successor initialized as {502,<0.113.0>}
<0.126.0>
Node 946: Already the predecessor. Ring is stable.
Node 724: Already the predecessor. Ring is stable.
Node 444: Already the predecessor. Ring is stable.
Node 502: Already the predecessor. Ring is stable.
Node 886: Adopting Xkey 444 as the new successor.
Node 946: Already the predecessor. Ring is stable.
Node 724: Already the predecessor. Ring is stable.
Node 444: Already the predecessor. Ring is stable.
Node 502: Already the predecessor. Ring is stable.
Node 886: Adopting Xkey 946 as the new successor.
Node 946: Already the predecessor. Ring is stable.
Node 724: Already the predecessor. Ring is stable.
Node 444: Already the predecessor. Ring is stable.
Node 502: Already the predecessor. Ring is stable.
Node 886: Notifying successor 946 about our existence.
Node 946: Already the predecessor. Ring is stable.
```

<u>Message Handling</u>

The system relies on message passing for communication between nodes. Ex, nodes send messages to request information, update their views, or transfer data. Erlang's pattern matching makes it easier to handle different types of messages.

```
        % A peer asks for our predecessor
        {request, Peer} ->
            request(Peer, Predecessor),
            node(Id, Predecessor, Successor, Store);
```

This snippet handles a request message by sending the predecessor information back to the requesting peer.

<u>Handover</u>

When a new node joins, it may take over responsibility for certain keys previously managed by another node. The handover function transfers these keys to the new node to ensure data consistency. This guarantees that each key is managed by the correct node based on the ring structure.

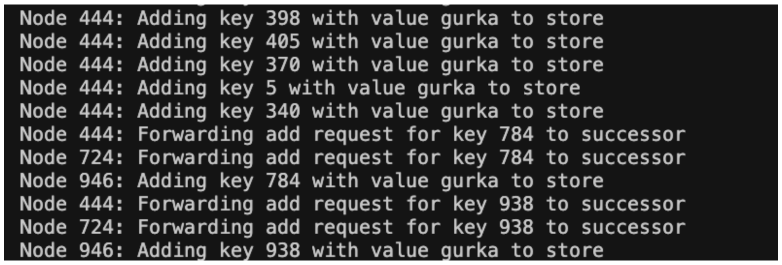
```
handover(Id, Store, Nkey, Npid) ->
   {Keep, Rest} = storage:split(Id, Nkey, Store),
   % Send the part of the store that should be handed over to the new predecessor
   Npid ! {handover, Rest},
   Keep.  % Return the part of the store that we keep
```

Here, a node receives a new (but **part** of an old) set of keys that it now needs to store after a new node takes over some of its previous responsibilities.

3 Testing

<u>Adding keys</u>

In the test, 1000 keys were added with values, and the logs showed how keys were either stored by a node directly or forwarded to a successor until they reached the node responsible for that key. This behavior is expected since each node is responsible for a specific range of keys based on their IDs (which is checked by a function in the key.erl module) and keys outside this range are forwarded.



```
Node 444: Adding key 398 with value gurka to store
Node 444: Adding key 405 with value gurka to store
Node 444: Adding key 370 with value gurka to store
Node 444: Adding key 5 with value gurka to store
Node 444: Adding key 340 with value gurka to store
Node 444: Forwarding add request for key 784 to successor
Node 724: Forwarding add request for key 784 to successor
Node 946: Adding key 784 with value gurka to store
Node 444: Forwarding add request for key 938 to successor
Node 724: Forwarding add request for key 938 to successor
Node 946: Adding key 938 with value gurka to store
```

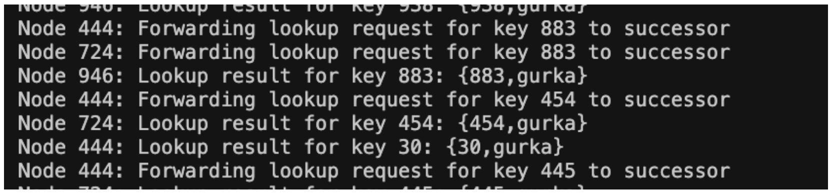
1000 keys being passed around the chord and stored by the node assigned to store them.

Which key is responsible? That is determined by this function:

```
between(Key, From, To) when From < To ->
   From < Key andalso Key =< To;
between(Key, From, To) when From > To ->
   Key =< To orelse From < Key;
between(Key, Id, Id) ->
   true.
```

Lookup

When performing lookups, the keys were traced through the network until they reached the node responsible for them, which then returned the associated value. The round-trip times were recorded, showing the time taken for each lookup operation.

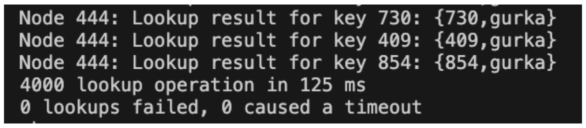


```
Node 946: Lookup result for key 938: {938,gurka}
Node 444: Forwarding lookup request for key 883 to successor
Node 724: Forwarding lookup request for key 883 to successor
Node 946: Lookup result for key 883: {883,gurka}
Node 444: Forwarding lookup request for key 454 to successor
Node 724: Lookup result for key 454: {454,gurka}
Node 444: Lookup result for key 30: {30,gurka}
Node 444: Forwarding lookup request for key 445 to successor
```

This confirms that keys are retrieved from the correct nodes based on their position in the ring.

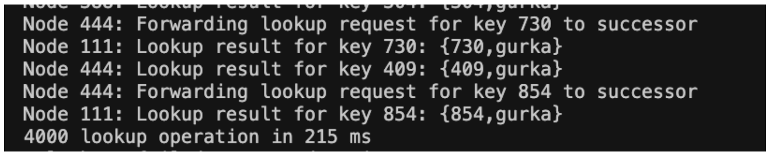
Check (time for lookup)

We can also use a check function to see how long it takes to look up the keys. This test is done by first handing all keys to one node. When we check for the keys, this node holds all keys, and the node will not forward the search to any other node in the chord. This test is with one node holding 4000 keys. The time i takes is 125 ms.



```
Node 444: Lookup result for key 730: {730,gurka}
Node 444: Lookup result for key 409: {409,gurka}
Node 444: Lookup result for key 854: {854,gurka}
4000 lookup operation in 125 ms
0 lookups failed, 0 caused a timeout
```

The same test is done after adding 3 more nodes, i.e. these nodes will get a fair share of keys each since the handover function gives them keys to be responsible for once they join. It takes 215 ms to look up the same keys when they are distributed in the chord ring.



```
Node 444: Forwarding lookup request for key 730 to successor
Node 111: Lookup result for key 730: {730,gurka}
Node 444: Lookup result for key 409: {409,gurka}
Node 444: Forwarding lookup request for key 854 to successor
Node 111: Lookup result for key 854: {854,gurka}
4000 lookup operation in 215 ms
```