ID2201 HT23 Distributed Systems, Basic Course (50929)

Report Homework 5 - Chordy - a distributed hash table

by Eugen Lucchiari Hartz

October 11, 2023

1 Introduction

In this homework a distributed hash table (DHT) following the Chord scheme was developed using Erlang. The objective here is to establish a well-organized network of nodes where each node only stores data it is responsible for. The focus is on efficiently locating nodes within the system. The Chord architecture includes mechanisms such as a ring structure, a key-value store, failure detection, replication for data resilience, and a routing table to optimize the process of finding nodes associated with specific keys in a large network. The ultimate goal is to create an effective and fault-tolerant distributed system for storing and retrieving information.
A Distributed Hash Table (DHT) is crucial in distributed systems because it provides an organized and efficient way to locate and manage data across multiple nodes. By distributing the responsibility of data storage and retrieval, DHTs enhance scalability, fault tolerance, and overall performance in large-scale distributed environments.

2 Main problems, solutions and evaluation

node1

This first implementation includes the key functionalities of a distributed hash table (DHT) using the Chord Scheme.

- Start nodes: each created node gets a unique identifier (ID)
- Connecting nodes: You can also connect nodes to form a ring. These connections are established by passing messages to set successors and predecessors.
- Periodic stabilization: There is an implemented stabilization mechanism to ensure the stability of the ring structure. Nodes periodically send messages to their successors to request information about their predecessor. Based on the received information, nodes may adjust their positions in the ring to maintain a stable network. So for example when a new node enters the ring it first sends a request message to its successor to get the predecessor of its

successor. Depending on the value of the predecessor a node could suggest to be the new predecessor of the successor. The successor will then receive a message and call the notify function to decide if the node should be the new predecessor or not.

- Handling messages for key-value operations: Nodes communicate through messages to perform various key-value operations. Each message carries specific information or instructions for the receiving node to execute the intended operation such as notifying nodes of their existence and requesting information about predecessors and successors.

```
4> Node1 = node1:start(1).
<0.103.0>
5> Node2 = node1:start(2, Node1).
<0.106.0>
6> Node3 = node1:start(3, Node2).
<0.109.0>
7> Node4 = node1:start(4, Node3).
<0.112.0>
8> Node1 ! state.
ID: 1
state
Predecessor: {4,<0.112.0>}, Successor: {2,<0.106.0>}
9> Node2 ! state.
ID: 2
state
Predecessor: {1,<0.103.0>}, Successor: {3,<0.109.0>}
10> Node3 ! state.
ID: 3
state
Predecessor: {2,<0.106.0>}, Successor: {4,<0.112.0>}
11> Node4 ! state.
ID: 4
state
Predecessor: {3,<0.109.0>}, Successor: {1,<0.103.0>}
```

```
5> Node1 ! probe.
Route: [1,2,3,4]
probe
Trip time: 31 micro
6> Node2 ! probe.
Route: [2,3,4,1]
probe
Trip time: 37 micro
7> Node3 ! probe.
Route: [3,4,1,2]
probe
Trip time: 38 micro
8> Node4 ! probe.
Route: [4,1,2,3]
probe
Trip time: 38 micro
```

node2

The second implementation of the DHT includes the storage feature.

- Includes a local key-value store and can perform operations like adding and looking up key-value pairs
- Each node is responsible for keys of the identifier of its predecessor and of the identifier of itself. When a node then receives an add operation, it first checks if it is responsible for the specified key. If it is, the node adds the key-value pair to its local store. However, if the node is not responsible for the key, it forwards the add operation, so sends an add message, to its successor for further processing.
- In the case of lookup operations, so to retrieve data which is associated with a specific key it is pretty similar. If the node is not responsible for the key, it forwards the lookup request to its successor for handling. This mechanism ensures that each node only deals with operations relevant to the keys for which it is responsible
- Furthermore there is also a new handover mechanism which ensures that a node that gets added to the ring takes over some responsibility and stores the key value pairs of its successor.

```
1> Node1 = node2:start(1).
<0.88.0>
2> Node2 = node2:start(2, Node1).
<0.91.0>
3> Node3 = node2:start(3, Node2).
<0.94.0>
4> Node4 = node2:start(4, Node3).
<0.97.0>
5> Node1 ! state.
ID: 1
state
Predecessor: {4,<0.97.0>}, Successor: {2,<0.91.0>}
Store: []
6> Data1 = test1:add(Node1, 10).
Total time to add 10 data in microseconds: 1838
[5922673,457924030,558255809,697140785,142108218,477121057,
 915656207,311326755,945816365,443584618]
7> Node1 ! state.
ID: 1
state
Predecessor: {4,<0.97.0>}, Successor: {2,<0.91.0>}
Store: [{5922673,562119769},
        {457924030,421398761},
        {558255809,214973049},
        {697140785,159811421},
        {142108218,209448557},
        {477121057,596510082},
        {915656207,666957294},
        {311326755,597447525},
        {945816365,501490715},
        {443584618,723040206}]
8> test1:lookup(Node1, Data1).
Total time to search for 10 elements in microseconds: 26
ok
```

```
4> Node1 = node2:start(1).
<0.103.0>
5> Node2 = node2:start(2, Node1).
<0.106.0>
6> Node3 = node2:start(3, Node2).
<0.109.0>
7> Node4 = node2:start(4, Node3).
<0.112.0>
8> test1:start(node2, 4).
[<0.115.0>,<0.117.0>,<0.118.0>,<0.119.0>]
9> Data = test1:add(Node1, 100).
Total time to add 100 data in microseconds: 309
[824062636,600784579,755193177,99422912,479668684,579315200,
 46587903,409650593,423000231,993583075,66580865,314326164,
 909000318,879152921,811870046,46807325,377449816,75227118,
 46391768,181403548,548798014,663112463,336201031,140468149,
 521769722,174207268,867833725,827307409,834748490|...]
10> test1:lookup(Node2, Data).
Total time to search for 100 elements in microseconds: 315
ok
```

Performance Tests

Test how long it takes for one machine to handle 4000 elements

```
4> RingNode = node2:start(1).
<0.103.0>
5> Data = test1:add(RingNode, 4000).
Total time to add 4000 data in microseconds: 8734
[660442466,384648173,771275756,572168058,788576311,
 889638420,298562388,66992907,169004843,396022537,530305525,
 921576523,109759741,965910187,970972344,282497650,563888898,
 293478259,204618471,411368207,633359248,803582820,253216824,
 763028676,937422979,482556368,461505015,358876188,355689951|...]
6> test1:lookup(RingNode, Data).
Total time to search for 4000 elements in microseconds: 26722
ok
```

Test how long it takes for four machines that do 1000 elements each

```
1> RingNode = node2:start(1).
<0.88.0>
2> Node2 = node2:start(2, RingNode).
<0.91.0>
3> Node3 = node2:start(3, RingNode).
<0.94.0>
4> Node4 = node2:start(4, RingNode).
<0.97.0>
5> Node5 = node2:start(5, RingNode).
<0.100.0>
6> DataNode2 = test1:add(Node2, 1000).
Total time to add 1000 data in microseconds: 5860
[59726834,293472065,206155967,522694773,438097849,617223761,
 115194951,761050990,35129670,25282717,197044603,924389650,
 202519600,316459363,85947706,450737191,815889427,82447524,
 500904190,186659620,475922503,817210468,40278861,823561604,
 412414079,585237729,264341031,645723865,261049347|...]
7> DataNode3 = test1:add(Node3, 1000).
Total time to add 1000 data in microseconds: 3707
[729354554,513094350,185506849,28267120,170311720,658481084,
 198786168,297969306,297166997,13991752,123777001,923705114,
 322906102,492499247,555498033,546068400,993559354,332054583,
 281149992,255267708,209160564,979533846,440528842,318825335,
 13137427,266117990,56187451,647746223,761065086|...]
8> DataNode4 = test1:add(Node4, 1000).
Total time to add 1000 data in microseconds: 3265
[669302363,237468226,663323190,641481534,282857632,
 585124250,385052856,180022634,165792595,323830268,568694548,
 466875841,974483904,670800518,777265386,92110384,352505869,
 338991908,907513242,679900383,876073259,871250356,533879008,
 303302454,809719226,611556576,660076251,562416571,886451343|...]
9> DataNode5 = test1:add(Node5, 1000).
Total time to add 1000 data in microseconds: 2855
[660442466,384648173,771275756,572168058,788576311,
 889638420,298562388,66992907,169004843,396022537,530305525,
 921576523,109759741,965910187,970972344,282497650,563888898,
 293478259,204618471,411368207,633359248,803582820,253216824,
 763028676,937422979,482556368,461505015,358876188,355689951|...]
10> test1:lookup(Node2, DataNode2).
Total time to search for 1000 elements in microseconds: 13086
ok
11> test1:lookup(Node3, DataNode3).
Total time to search for 1000 elements in microseconds: 9381
ok
12> test1:lookup(Node4, DataNode4).
Total time to search for 1000 elements in microseconds: 6249
ok
13> test1:lookup(Node5, DataNode5).
Total time to search for 1000 elements in microseconds: 2627
ok
```

For one machine:
- Total time to add 4000 elements: 8734 microseconds
- Total time to search for 4000 elements: 26722 microseconds

For four machines:
- Total time to add 4000 elements: 15687 microseconds
- Total time to search for 4000 elements: 31343 microseconds

Accordingly it takes longer for four machines to handle 1000 elements each than for one machine to handle 4000 elements.
However the limiting factor in this scenario is the processing capacity of a single machine. When you have one machine handling all 4000 elements, it has to sequentially process each element, and the total time is determined by the processing speed of that single machine.

Bonus 1

node3
Handling failure

The goal here is to make it possible that the ring repairs itself if a node crashes. To detect node failures and initiate self repair the built in monitor function is used. When a successor crashes it updates its successor to the next in line, so to the successor of its successor which is called Next. Accordingly, when a successor crashes the node seamlessly adopts the next node as the new successor. The system then monitors the new successor and executes the stabilization procedure to ensure the continued connectivity and integrity of the ring structure. In case a node's predecessor crashes the predecessor is set to nil which allows the node to wait for a new predecessor to eventually join the ring.

```
2> Node1 = node3:start(1).
<0.93.0>
3> Node2 = node3:start(2, Node1).
<0.96.0>
4> Node3 = node3:start(3, Node2).
<0.99.0>
5> Node4 = node3:start(4, Node3).
<0.102.0>
6> Node2 ! state.
ID: 2
state
Predecessor: {1,#Ref<0.4130334531.2167406593.119430>,<0.93.0>}, Successor: {3,
                                                                             #Ref<0.4130334531.2167406593.119552>,
                                                                             <0.99.0>}
Store: []
7> test:kill(Node3).
Successor of #Ref<0.4130334531.2167406593.119552> died
true
8> Node2 ! state.
ID: 2
state
Predecessor: {1,#Ref<0.4130334531.2167406593.119430>,<0.93.0>}, Successor: {4,
                                                                             #Ref<0.4130334531.2167406593.122106>,
                                                                             <0.102.0>}
Store: []
9> Node2 ! probe.
Route: [2,4,1]
probe
Trip time: 31 micro
10> Node4 ! state.
ID: 4
state
Predecessor: {2,#Ref<0.4130334531.2167406593.122110>,<0.96.0>}, Successor: {1,
                                                                             #Ref<0.4130334531.2167406593.119708>,
                                                                             <0.93.0>}
Store: []
11> Node1 ! state.
ID: 1
state
Predecessor: {4,#Ref<0.4130334531.2167406593.119709>,<0.102.0>}, Successor: {2,
                                                                              #Ref<0.4130334531.2167406593.119429>,
                                                                              <0.96.0>}
Store: []
```

As we can see after killing Node3 the DHT managed to detect the crash and to update the ring.

Bonus 2

node4
Replication

The goal here is to implement a replication strategy to ensure the availability and consistency of data in case of node failures.
When a node is intialized the primary store and the replica store are created as empty lists.
When a node then receives an add or replicate request to add a key-value pair to its store, it forwards a replication request to its successor. The successor node then receives the replication request and adds the key-value pair to its own store. It then replies with an acknowledgment to the original requester, ensuring that the addition is properly performed.

After receiving a replication acknowledgment from the successor, the node merges the updated key-value pair into its Replica store. This ensures that the Replica maintains a duplicate copy of the successor's store.
The replication strategy ensures data consistency by making sure that an acknowledgment is sent only when an element has been successfully added to both the primary store and the Replica.
In case of a node failure the successor takes over responsibility for the store held by the failed node. The Replica store of the successor is then merged with its own store to ensure that it has a comprehensive view of the data.
This replication strategy which includes replicating data at the successor node and ensuring proper acknowledgment ensures availability and consistency of data in case of node failures.

```
5> Node1 = node4:start(1).
<0.108.0>
6> Node2 = node4:start(2, Node1).
<0.111.0>
7> Node3 = node4:start(3, Node2).
<0.114.0>
8> Node4 = node4:start(4, Node3).
<0.117.0>
9> Node1 ! state.
ID: 1
state
Predecessor: {4,#Ref<0.3661614167.1736179713.236679>,<0.117.0>}, Successor: {2,
                                                                              #Ref<0.3661614167.1736179713.236355>,
                                                                              <0.111.0>}
Store: []
10> Data1 = test1:add(Node1, 10).
Total time to add 10 data in microseconds: 4029
[5922673,457924030,558255809,697140785,142108218,477121057,
 915656207,311326755,945816365,443584618]
11> Node1 ! state.
ID: 1
state
Predecessor: {4,#Ref<0.3661614167.1736179713.236679>,<0.117.0>}, Successor: {2,
                                                                              #Ref<0.3661614167.1736179713.236355>,
                                                                              <0.111.0>}
Store: [{5922673,562119769},
        {457924030,421398761},
        {558255809,214973049},
        {697140785,159811421},
        {142108218,209448557},
        {477121057,596510082},
        {915656207,666957294},
        {311326755,597447525},
        {945816365,501490715},
        {443584618,723040206}]
12> test1:lookup(Node1, Data1).
Total time to search for 10 elements in microseconds: 22
ok
13> test1:lookup(Node2, Data1).
Total time to search for 10 elements in microseconds: 38
ok
14> test:kill(Node1).
Successor of #Ref<0.3661614167.1736179713.236678> died
true
15> test1:lookup(Node2, Data1).
Total time to search for 10 elements in microseconds: 18
ok
```

3 Conclusion

In summary, the assignment was very useful to discover how a distributed hash table (DHT) following the Chord scheme works. It was very interesting to implement such a DHT, because it gives you a better understanding of how DHTs work and why they enable efficient data storage and retrieval across a network. It was also informative to run different experiments to discover changes and to do investigations about failure handling and replication.