KTH H16P01 Distributed Systems Basic Course: Chordy

Thorsteinn Thorri Sigurdsson (ttsi@kth.se)

October 13, 2016

1 Introduction

In this assignment a distributed hash table was implemented, a variant of the Chord protocol. Three implementations were made. The first implementation only allows for arranging nodes in a ring. The second implementation introduces a key-value store to the system, with put and get operations. The third implementation introduces fault tolerance to the system. The ring gets repaired if nodes leave the system, but the data contained in those nodes will be lost. The next step would be to add replication of data to the system to recover the data contained in the lost nodes. Note that a routing mechanism (finger tables) is not implemented, all messages must circle the ring to find their destination.

2 First implementation: A ring

The first implementation just allows us to start nodes that will connect to each other and form a ring.

Each node that wants to join the ring gets a random number as a key. These keys will hopefully be unique within the system. In the full implementation of Chordy nodes have names which are then hashed to generate the keys. We use this simpler approach.

The nodes will be arranged in a ring in increasing key order. The ring is doubly linked, i.e. each node keeps track of its predecessor and successor. When the first node joins the ring it considers itself as its successor, i.e. the node's successor is the node itself. When other nodes join the ring they are of course aware of some other node that is already in the ring (the node's address, or in our case, it's PID). Let's call this node the peer. They send a `key` message to the peer and the peer will reply with its key. The peer is then set as the node's successor on the ring. The node's predecessor is set to `nil` in both cases, when a node is the first node in a ring and when it is joining an existing ring.

This ring is of course not ordered correctly. In fact it isn't even a ring. Newly joined nodes just point to the node that they knew about in the ring. To fix this we periodically run a stabilization algorithm. The node sends a `request` message to its successor, which will reply with a `status` message that contains information about the successor's predecessor. Upon receiving this message, the node will check if something needs to be done. The following list shows what has to be done in each case. We use the word *myself* to indicate the node.

1. My successor doesn't have a predecessor. Send a `notify` message to it.
2. My successor's predecessor is myself. Everything is OK.
3. My successor's predecessor is itself (i.e. the successor itself). Send a `notify` message to it.
4. My successor's predecessor is some other node. This node is between myself and my successor. I adopt this node as my new successor and run `stabilize` again.
5. My successor's predecessor is some other node. I am between this node and my successor. Send a `notify` message to my successor.

The `notify` message the node sends to its successor notifies the successor of the node's existence, suggesting that it might be the successor's predecessor. The successor then checks what needs to be done with this information. If it doesn't have a predecessor, it adopts the node as its predecessor. If it already has a predecessor, it checks if the node is between its current predecessor and itself. If it is, the successor adopts the node as its new predecessor. If it isn't, the successor does nothing and keeps its current predecessor.

Through this stabilization algorithm, the ring will be created with the nodes in the ring ordered in increasing key order. This algorithm is run at set intervals, as well as every time a node finds a new successor. The node that got the new successor then runs stabilization again, this time sending a `request` message to its new successor, which will reply with a `status` message containing info on its current predecessor, starting the stabilization process again but now with the new successor and its current predecessor. Concurrent iterations of this algorithm will result in a correct ring with nodes ordered in increasing key order.

(The stabilization process is described as *asking ahead*).

2.1 Between-check considerations

When we check if a key is between two other keys (`from`, `to`) we say that it is if it is larger than `from` or less than or equal to `to` (i.e. in the range

(`from, to]`). This is different from the normal Chord implementation.

This does not matter in our case. First of all, we are assuming (hoping) that our (random) keys are unique. Second of all, when we check if our successor's predecessor's key (`Xkey`) is between our key (`Id`) and our successor's key (`Skey`), then technically we would allow `Xkey` to be equal to `Skey` and still say that `Xkey` is between `Id` and `Skey`. This is OK, since this would mean that our successor's predecessor is our successor (i.e. it is pointing to itself), assuming unique keys. In our implementation, this case is already handled correctly before this between-check is done.

2.2 Stabilization considerations

Running the stabilization algorithm more frequently (i.e. the automatic, perdiodic running of the algorithm) will result in nodes being integrated into the ring more quickly. However, this will also mean more messages are sent through the system. This might be unnecessary if peer churn (actually just joins in our case) is low.

If we don't schedule the stabilization procedure things would not work. When a node joins the ring its first successor will be the node that it knew about in the ring (and used to join the ring). This is most likely not the correct successor for the node. The only way for a node to get a new successor is through the stabilize algorithm, and the only time the stabilize algorithm is run is when it is scheduled, or when we get a new successor. Since the only way to get a new successor is through running the stabilization algorithm, it is apparent that the stabilization algorithm must be scheduled for things to work. Otherwise the ring will never form.

2.3 Notify considerations

After receiving a `notify` message from a node (that has us as its successor) we check if the sender of that message might be our predecessor. If we don't have a predecessor we automatically adopt the sender as our predecessor, but if we don't have a predecessor we have to check if the sender is between our current predecessor and ourselves. If it is, we adopt it as our new predecessor, if it isn't, we do nothing and keep our old predecessor.

We do not need a special case to check if we are pointing to ourselves (i.e. my predecessor is myself). We check if the sender is between our current predecessor and myself using the `key:between/3` function.

```
% key:between(Key, From, To) - called as:
key:between(SenderKey, CurrentPredecessorKey, MyKey)
```

This will return `true` if `SenderKey` is greater than `CurrentPredecessorKey` and less than or equal to `MyKey`. However, if `From` is equal to `To`, (i.e. `CurrentPredecessorKey` is equal to `MyKey`, and I am pointing to myself)

our function returns true. We say that everything on the ring is between `From` and `To` in this case.

This means that if we are pointing to ourselves, we will always adopt the sender as our new predecessor.

We do not have to inform the sender about our decision. If we discard the `notify` message (proposal from the sender that it might be our predecessor) we do not let the sender know. The next time the sender (that has us as its successor) runs the stabilization algorithm, it will send us a `request` message and we will respond by telling him our predecessor. The sender can then see if we discarded his previous `notify` message or not. It will then either send us a new `notify` message or adopt our predecessor as its successor.

2.4 Evaluation

Tests were done to verify that the ring was formed correctly. A probe message is sent a full circle through the ring, logging the nodes (and their keys) that it passes through on its way (as well as the time it takes to traverse the ring).

The tests show that a ring is formed with the nodes in increasing key order.

```
[<0.284.0>] Received my own probe -
Created time: 1476308658414404 -
Current time: 1476308658414424 -
Nodes: [
{753862312,<0.284.0>},
{798709172,<0.289.0>},
{941046172,<0.285.0>},
{945294532,<0.286.0>},
{51520215,<0.288.0>},
{196257481,<0.287.0>}]
```

The node with key `51520215` is the "first" node in the ring, with the smallest key. The rest of the nodes are ordered in increasing key order until wrapping around and getting to the first node again.

3 Second implementation: Storage

Next a key-value store was added to the system, with the possibility to add and search for key-value pairs. A node will store all key-value pairs where the key is larger than the key (ID) of its predecessor and smaller or equal to the key (ID) of itself.

When a node receives a request to add a key-value to its store, it checks if it is responsible for that key. If it is it adds it to its store, and if it isn't it forwards the request to its successor. The lookup requests are similar, if we are responsible for the requested key we do a lookup in our local store, otherwise we forward the lookup request to our successor.

When a new node joins the ring we need to give it the key-value pairs that it is supposed to take care of (and are already stored in the ring). We do this every time we accept a new predecessor. Then we figure out what part of our local key-value store the new predecessor should get, and send it to the new predecessor via a `handover` message. The new predecessor will then merge the elements we send him with his local key-value store.

3.1 Evaluation: Performance tests

All tests were done on one machine, simulating other machines by using concurrent processes to add data elements to the ring and do lookups.

3.2 1 node in ring, 4 workers add and lookup 1000 elements each

Sample test run:

```
33> test:performanceTest(1, 4, 1000).
Sleeping (to stabilize ring)
Running performance test - N: 1 - Testmachines: 4 - Elements: 1000
Starting test machine worker 4
Starting test machine worker 3
Starting test machine worker 2
Starting test machine worker 1
ok
1000 lookup operation in 21 ms
0 lookups failed, 0 caused a timeout
1000 lookup operation in 18 ms
0 lookups failed, 0 caused a timeout
1000 lookup operation in 19 ms
0 lookups failed, 0 caused a timeout
1000 lookup operation in 18 ms
0 lookups failed, 0 caused a timeout
```

5 test runs were done and the average time taken, for all workers for all test runs. The average lookup time was `19,6 ms`.

3.3 1 node in ring, 1 worker adds and lookups 4000 elements

Sample test run:

```
2> test:performanceTest(1, 1, 4000).
Sleeping (to stabilize ring)
Running performance test - N: 1 - Testmachines: 1 - Elements: 4000
Starting test machine worker 1
ok
4000 lookup operation in 18 ms
0 lookups failed, 0 caused a timeout
```

5 test runs were done and the average time taken, for all test runs. The average lookup time was `21,6 ms`.

3.4 Comparison

It takes longer to handle 4000 elements from one machine (worker process) than 4 different machines sending 1000 elements each. When the worker process adds an element to the store, it waits for an `ok` reply before moving on to the next element. If we only have one machine (worker process) sending 4000 elements, they all get sent sequentially. To add one element requires us to send a message, wait for the node to process it (no messages sent within ring since there is only one node in it), and wait for an `ok` message. Then we can prepare the next `add` message to send, send it, wait for `ok`, etc. However, if we have 4 machines (worker processes) sending 1000 messages each, they will all send `add` messages to the node at the same time (concurrently). Once the node has processed a message, it will send an `ok` message to the corresponding worker process. Meanwhile, the other `add` messages from the other worker processes are waiting in the pipeline. The node can then take the next `add` message from the pipeline and start processing it, and so on. So by having 4 worker processes we are saving ourselves the processing time and message sending time in the worker process (i.e. going to the next element, send it, wait for message to reach node) by having the workers run concurrently.

The same applies for the `lookup` messages (we also wait for a reply message there).

What if we have more nodes in the ring?

All `add` messages get sent to the same node in the ring. The `add` message then traverses the ring getting sent to the next successor until the data element ends up in the node that is responsible for it. Only then is the `ok` message sent. So in the worst case scenario, the add message may be sent N-1 times between the nodes in the ring before the `ok` message is sent to the worker process and it can send the next `add` message carrying the next data element.

However, if we have 4 different machines (worker processes) they can send these `add` messages concurrently. The first node to receive them in the ring can then send each `add` message on its way to find its correct home

in the ring, and immediately receive the next `add` message from another worker process. This way we can have multiple `add` messages circling the ring looking for their new home concurrently, giving us better performance.

The same applies for the `lookup` messages (they also have to circle the ring to find the correct home of the data element, and we also wait for a reply message there before moving on to the next lookup message).

3.5 More tests

We add another node to the ring. Now we have 2 nodes in the ring and 4 worker processes sending 1000 elements each. The average lookup time is `18,15 ms`.

Then we add two more nodes to the ring. Now we have 4 nodes in the ring and 4 worker processes sending 1000 elementes each. The average lookup time is again `18,15 ms`.

It is a coincidence that both cases give the same result. There is an outlier in the test data form the (2,4,1000) test (28 ms while most of the other measurements are <= 20 ms), which is affecting the average (making it higher). By adding more nodes to the ring I would expect the lookup times to increase, since more message passing is required within the ring to find the right value.

One more test was done with 4 nodes in the ring and 4 worker processes, each sending 10000 elements. The average lookup time is `573,45 ms`.

This is high, lookup time does not increase linearly with the number of elements. We are just using randomly generated keys for both the node keys and the data element keys, between 1 and 1.000.000.000. We only have 4 nodes in the ring (4 samples from the uniform distribution that the keys are taken from), so there is no guarantee that they data elements will be evenly distributed among the nodes. One node might have a small key, e.g. 10.000.000, and the next node in the ring might have key e.g. 100.000.000. This means that the first node can take up to 10 million data elements, while the second one can have up to 99 million data elements.

Now we are distributing more elements among the nodes, 40000 in total, so it is possible that the different local key-value stores can end up with a very different number of elements. As we get more elements in a local key-value store, the lookup times within the local key-value store could start to matter.

Does it matter if all test machines access the same node?

Yes, I believe that it matters. If all `add` and `lookup` messages go through the same node, they will be processed one by one. Get message, send it around the ring, get next message, send it around the ring, etc. Furthermore, they will all be sent the same way around the ring, so the successor of the access node will receive the stream of messages from the access node, then

send it to his successor etc. until the message reaches the correct node in the ring (note that there are no finger tables in our implementation).

If we had multiple access nodes, we would be able to receive messages and send them around the ring concurrently, which is better.

The messages would also begin their circulation of the ring at different places, instead of all going the same path from successor to successor. If there is only one access node, the successor of the access node would receive almost all messages (those that the access node couldn't handle itself), and then the successor of the successor would receive a little bit fewer messages (those that the access node and its successor couldn't handle), then the next successor gets even fewer messages and so on. This goes on until the last node in the ring (the access node's predecessor), which should only receive messages that were destined for it anyway (all other messages have been filtered out as they go around the ring). This leads to uneven load on the ring. Therefore it would be more efficient to have the messages begin their circulation of the ring at different places.

4 Third implementation: Handle failed nodes

In the third implementation we introduce failure handling. The ring should survive if a node leaves the system.

To do this we make the nodes track their successor's successor (which we will call the `Next` node) as well, so each node points to the next two nodes on the ring. The extra pointer to `Next` is sometimes called a *safety pointer*. The full version of Chord has more safety pointers (each node points to more nodes in the ring), and in general, more safety pointers mean more failure tolerance, since we can survive if a couple of nodes in a row fail. With only one safety pointer we can only survive one node crashing at a time.

We make the `status` messages (that we send after having received a `request` message from a node - stabilization algorithm) now contain info about a node's successor as well as its predecessor. The node receiving the `status` message can then figure out both its next successor and its next `Next` node (successor's successor).

(If I get a new successor that means that the new successor pushed itself in between me and my old successor - then my old successor becomes my `Next` node. If I don't get a new successor, I keep my old one and adopt his successor as my `Next` node).

A node uses erlang's `monitor` function to keep track of both its predecessor and successor. When a new node is adopted as predecessor or successor, we stop monitoring the old one and start monitoring the new one.

When a node fails, the nodes that are monitoring it will receive a `DOWN` message. Once the node receives the `DOWN` message, it checks if it is from its predecessor or successor.

If our predecessor died we just set it as `nil` and carry on (we do not have safety pointers for predecessors).

If our successor died, we adopt our `Next` node (our old successor's successor) as our new successor, start monitoring it, and set our `Next` node to `nil`. We also run the stabilization algorithm.

Note that if a node dies, the key-values that it is storing will be lost. To get around this, we need to add replication of data to the system.

4.1 Unreliable failure detector considerations

What if a node is falsely reported as being dead? I.e. we receive a `DOWN` message for a node that isn't dead. Then we will either remove our predecessor-refernce to it, or adopt its successor as our new successor.

The node would then eventually run stabilization again, send a `request` message to what was its successor, he replies with a `status` message containing his predecessor, the node then sends a `notify` message to its successor, and so on. Basically, the node will be brought back into the fold through the stabilization algorithm.

If the node believes that the rest of the network is gone, it will have lost its predecessor and successor pointers. It then needs to rejoin the network (under a new key (ID)) once it figures out that the network didn't go anywhere.

4.2 Evaluation

Tests were done by starting a ring of nodes, then killing some of the nodes and see if the ring stabilizes. The tests showed that the ring does indeed stabilize. Sample test run:

```
27> test:testNode3().
[<0.235.0>] Received my own probe -
Created time: 1476291059352762 -
Current time: 1476291059352792 -
Nodes: [
{134660158,<0.235.0>},
{165857176,<0.236.0>},
{236005157,<0.237.0>},
{271543407,<0.241.0>},
{333638099,<0.240.0>},
{453243160,<0.242.0>},
{772208333,<0.239.0>},
{920282593,<0.238.0>}]

% We have a stable ring, that is ordered in increasing key order.
% Now let's kill some nodes
```

```
Stopping node 1 - PID: <0.235.0>
Stopping node 3 - PID: <0.237.0>
Stopping node 7 - PID: <0.241.0>

[<0.242.0>] Received my own probe -
Created time: 1476291074354916 -
Current time: 1476291074354997 -
Nodes: [
{453243160,<0.242.0>},
{772208333,<0.239.0>},
{920282593,<0.238.0>},
{165857176,<0.236.0>},
{333638099,<0.240.0>}]
probe

% The killed nodes have been removed from the ring
% and it has stabilized in the correct order again
```

5 Conclusions

What is the advantage of the distributed store, is it performance or fault tolerance?

We definitely get increased performance. First off, we have more machines serving requests concurrently, instead of all requests going to the same centralized storage. By using a distributed hash table we are also able to achieve scalability since we are able to distribute the data (hopefully) evenly among the participating nodes, instead of having a huge centralized storage. As the amount of data increases, we can add more nodes to spread it out more. If the load on the system is high (many requests), we can also add more nodes to spread the load among more machines.

We also gain some fault tolerance. If a node dies we do not lose all of the data in the system like we would do if this was a centralized storage solution. The other nodes will still be alive and ready to serve requests. To achieve real fault tolerance we would also need to implement replication of data among the nodes to stop us from losing any data when a node dies.

If we want to optimize for fault tolerance we should have a high degree of replication. This will impact performance since the replicas need to be kept consistent. If we optimize for performance we would use a lower degree of replication (if any).