ID2201 Distributed Systems, basic course

Chordy: A Distributed Hash Table

Johan Montelius, Vladimir Vlassov and Klas Segeljakt
Email: {johanmon,vladv,klasseg}@kth.se
June 28, 2024

Introduction

In this assignment, you will implement a distributed hash table following the Chord scheme. To understand what you're about to do, you should have a basic understanding of Chord and, preferably have read the original paper.

The first implementation will only maintain a ring structure; we will be able to add nodes in the ring but not add any elements to the store. Once we have a growing ring, we will introduce a store where key-value pairs can be added. Adding and searching for values will only introduce a few new messages and one parameter representing the store. When we have the distributed store, we can perform some benchmarks to see if the distribution gives us anything.

Moving forward, we will add failure detection to the system. Each node will keep track of the liveness of its successor and predecessor; if they fail, the ring must be repaired. If a predecessor dies, we don't do much, but if our successor dies, we have to contact the next in line. In the presentation of Chord, one will keep a list of potential successors, but to keep things simple, we will only keep track of one more successor.

Maintaining the ring in the face of failures is all well, but if a node dies, we will lose information. To solve this problem, we will have to replicate the values in the store. We will introduce a simple replication scheme that might solve some problems, but does it actually work?

The Chore architecture also defines a routing table, called fingers, for each node. The routing table will allow us to find any given key in $log(n)$ hops. This is, of course, important if we have a large ring. In a network of twenty nodes, it will, however be quite manageable to rely on the successor pointers only.

If you also want to implement mutable objects, you will face a challenge. How do you consistently update an object if it's replicated? Some objects or nodes might be unavailable during node insertion and failures. To solve this, you will have to do some more reading.

1 Building a ring

Start this project implementing a module `node1`. It will be our first implementation that only handles a growing ring. It could be good to keep this simple to see what functionality introduces new messages.

1.1 Keys

Chord uses hashing of names to create unique keys for objects. We will not use a hash function; instead, a random number is generated when a new key is generated. We will thus not have any "names" only keys. A node that wants to join the network will generate a random number, and we will hope that this is unique. This is ok for all of our purposes.

In a module `key`, implement two functions: `generate()` and `between(Key, From, To)`. The function `generate/0` will return a random number from 1 to 1.000.000.000 (30 bits) large enough for our test. Using a hash function such as SHA-1 would give us 160 bits and allow us to have human-readable names on objects but let's keep things simple. Use the Erlang module `random` to generate numbers.

The `between/3` function will check if a `Key` is between `From` and `To` or equal to `To`; this is called a partly closed interval and is denoted $(From, To]$.

Remember that we're dealing with a ring, so it could be that `From` is larger than `To`. What does that mean, and how do you handle it? Also, `From` could be equal to `To`, and we will interpret this as the full circle, i.e., anything is in between.

report -> https://claude.ai/chat/82396cc7-25ec-4746-904f-eda24266e522

1.2 The node

A node will have the following properties: a key, a predecessor, and a successor (remember that we will wait with the store until later). We need to know the key values of both the predecessor and the successor so these will be represented by a tuple `{Key, Pid}`.

The messages we need to maintain the ring are:

- `{key, Qref, Peer}` : a peer needs to know our key;
- `{notify, New}` : a new node informs us of its existence;
- `{request, Peer}` : a predecessor needs to know our predecessor;
- `{status, Pred}` : our successor informs us about its predecessor.

If you read the original paper that describes the Chord architecture, they, of course, describe it a bit differently. They have a pseudo-code description where they call functions on remote nodes. Since we need to build everything on message passing and handle the case where there is only one node in the ring pointing to itself, we need to make things asynchronous.

If we delay all the tricky decisions to sub-routines, the implementation of the node process could look like this:

```
node(Id, Predecessor, Successor) ->
    receive
        {key, Qref, Peer} ->
            Peer ! {Qref, Id},
            node(Id, Predecessor, Successor);

        {notify, New} ->
            Pred = notify(New, Id, Predecessor),
            node(Id, Pred, Successor);

        {request, Peer} ->
            request(Peer, Predecessor),
            node(Id, Predecessor, Successor);

        {status, Pred} ->
            Succ = stabilize(Pred, Id, Successor),
            node(Id, Predecessor, Succ);
   end.
```

You could also include handlers to print out some state information, terminate, and a catch-all clause in case strange messages are sent.

1.3 stabilize

The periodic stabilize procedure will consist of a node sending a {`request, self()`} message to its successor and then expecting a {`status, Pred`} in return. When it knows the predecessor of its successor, it can check if the ring is stable or if the successor needs to be notified about its existence through a {`notify,` {`Id, self()`} message.

Below is a skeleton for the `stabilize/3` procedure. The `Pred` argument is our successor's current predecessor. If this i `nil`, we should inform it about our existence. If it points back to us, we don't have to do anything. If it is pointing to itself, we should, of course, notify it about our existence.

If it's pointing to another node, we need to be careful. The question is whether we should slide in between the two nodes or place ourselves behind the predecessor. If the key of the predecessor of our successor (`Xkey`) is between us and our successor, we should, of course, adopt this node as our successor and run stabilization again. If we should be between the nodes, we inform our successor of our existence.

```
stabilize(Pred, Id, Successor) ->
    {Skey, Spid} = Successor,
    case Pred of
        nil ->
             :

        {Id, _} ->
             :
        {Skey, _} ->
             :

        {Xkey, Xpid} ->
            case key:between(Xkey, Id, Skey) of
                true ->
                     :
                false ->
                     :
            end
    end.
```

If you study the Chord paper, you will find that they explain things slightly differently. We use our `key:between/3` function that allows the key `Xkey` to be equal to `Skey`, not strictly according to the paper. Does this matter? What does it mean that `Xkey` is equal to `Skey`? Will it ever happen? report

1.4 request

The stabilize procedure must be done at regular intervals so that new nodes are quickly linked into the ring. This can be arranged by starting a timer that sends a `stabilize` message after a specific time.

The following function will set up a timer and send the request message to the successor after a predefined interval. In our scenario, we might set the interval to be 1000 ms in order to slowly trace what messages are sent.

```
schedule_stabilize() ->
    timer:send_interval(?Stabilize, self(), stabilize).
```

The procedure `schedule_stabilize/1` is called when a node is created. When the process receives a `stabilize` message, it will call `stabilize/1` procedure.

```
        stabilize ->
            stabilize(Successor),
            node(Id, Predecessor, Successor);
```

The `stabilize/1` procedure will then simply send a `request` message to its successor. We could have set up the timer so that it was responsible for sending the request message, but then the timer would have to keep track of which node was the current successor.

```
stabilize({_, Spid}) ->
    Spid ! {request, self()}.
```

A process picks up the request message, and the `request/2` procedure is called. We should, of course, only inform the peer that sent the request about our predecessor, as in the code below. The procedure is over complicated for now, but we will later extend it to be more complex.

```
request(Peer, Predecessor) ->
    case Predecessor of
        nil ->
            Peer ! {status, nil};
        {Pkey, Ppid} ->
            Peer ! {status, {Pkey, Ppid}}
    end.
```

What are the pros and cons of a more frequent stabilizing procedure? What is delayed if we don't do stabilizing that often? report

1.5 notify

Being notified of a node is a way for a node to make a friendly proposal that it might be our proper predecessor. We can not take their word for it, so we have to do our own investigation.

```
notify({Nkey, Npid}, Id, Predecessor) ->
    case Predecessor of
        nil ->
             :
        {Pkey,  _} ->
            case key:between(Nkey, Pkey, Id) of
                true ->
                         :
                false ->
                         :
            end
    end.
```

If our own predecessor is set to nil the case is closed but if we already have a predecessor we of course have to check if the new node actually should be our predecessor or not. Do we need a special case to detect that we're pointing to ourselves?

Do we have to inform the new node about our decision? How will it know if we have discarded its friendly proposal? report

1.6 Starting a node

The only thing left is how to start a node. There are two possibilities: either we are the first node in the ring or we're connecting to an existing ring. We'll export two procedures, `start/1` and `start/2`, the former will simply call the latter with the second argument set to `nil`.

```
start(Id) ->
    start(Id, nil).

start(Id, Peer) ->
    timer:start(),
    spawn(fun() -> init(Id, Peer) end).
```

In the `init/2` procedure we set our predecessor to `nil`, connect to our successor, and schedule the stabilizing procedure; or rather, make sure that we send a `stabilize` message to ourselves. This also has to be done even if we are the only node in the system. We then call the `node/3` procedure that implements the message handling.

```
init(Id, Peer) ->
    Predecessor = nil,
    {ok, Successor} = connect(Id, Peer),
    schedule_stabilize(),
    node(Id, Predecessor, Successor).
```

The `connect/2` procedure is divided into two cases; are we the first node or trying to connect to an existing ring? In either case, we need to set our successor pointer. If we're all alone we are of course our own successors. If we're connecting to an existing ring we send a `key` message to the node that we have been given and wait for a reply. Below is the skeleton code for the `connect/2` procedure.

```
connect(Id, nil) ->
    {ok, .......};
connect(Id, Peer) ->
    Qref = make_ref(),
    Peer ! {key, Qref, self()},
```

```
receive
    {Qref, Skey} ->
            :
after ?Timeout ->
        io:format("Time out: no response~n",[])
end.
```

Notice how the unique reference is used to trap exactly the message we're looking for. It might be overkill in this implementation but it can be quite useful in other situations. Also note that if we for some reason do not receive a reply within some time limit (for example, 10s) we return an error.

What would happen if we didn't schedule the stabilize procedure? Would things still work?

The Chord system uses a procedure that quickly will bring us closer to our final destination, but this is not strictly needed. The stabilization procedure will eventually find the right position for us. report

1.7 does it work

Do some small experiments, starting with one Erlang machine but then in a network or machines. When connecting nodes on different platforms, remember to start Erlang i distributed mode (giving a -name argument) and make sure that you use the same cookie (-setcookie).

We can introduce a probe message to check if the ring is actually connected.

```
{probe, I, Nodes, Time}
```

If the second element, `I`, is equal to the `Id` of the node, we know that we sent it and can report the time it took to pass it around the ring. If it is not our probe we simply forward it to our successor but add our own process identifier to the list of nodes.

The timestamp is set when creating the probe, use `erlang:system_time(micro_seconds)` to get microsecond accuracy (this is local time so the timestamp does not mean anything on other nodes).

```
    probe ->
        create_probe(Id, Successor),
        node(Id, Predecessor, Successor);

    {probe, Id, Nodes, T} ->
        remove_probe(T, Nodes),
        node(Id, Predecessor, Successor);

    {probe, Ref, Nodes, T} ->
```

```
        forward_probe(Ref, T, Nodes, Id, Successor),
        node(Id, Predecessor, Successor);
```

If you run things distributed, you must, of course register the first node under a name, for example `node`. The remaining nodes will then connect to this node using, for example the address:

```
{node,  'chordy@192.168.1.32'}
```

The connection procedure will send a name to this registered node and get a proper process identifier of a node in the ring. If we had machines registered in a DNS server, we could make this even more robust and location independent.

2 Adding a store

We will now add a local store to each node and the possibility to add and search for key-value pairs. Create a new module, `node2`, from a copy of `node1`. We will now add and do only slight modifications to our existing code.

2.1 A local storage

The first thing we need to implement is local storage. This could easily be implemented as a list of tuples {`Key, Value`}, we can then use the key functions in the `lists` module to search for entries. Having a list is of course not optimal but will do for our experiments.

We need a module `storage` that implements the following functions:

- `create()`: create a new store
- `add(Key, Value, Store)`: add a key-value pair, return the updated store
- `lookup(Key, Store)`: return a tuple {Key, Value} or the atom false
- `split(From, To, Store)` return a tuple {Updated, Rest} where the updated store only contains the key-value pairs requested and the rest are found in a list of key-value pairs;
- `merge(Entries, Store)`: add a list of key-value pairs to a store

The split and merge functions will be used when a new node joins the ring and should take over part of the store.

2.2 New messages

If the ring was not growing, we would only have to add two new messages: {`add, Key, Value, Qref, Client`} and {`lookup, Key, Qref, Client`}. As before, we implement the handlers in separate procedures. The procedures will need information about the predecessor and successor in order to determine if the message is actually for us or if it should be forwarded.

```
        {add, Key, Value, Qref, Client} ->
            Added = add(Key, Value, Qref, Client,
                        Id, Predecessor, Successor, Store),
            node(Id, Predecessor, Successor, Added);

        {lookup, Key, Qref, Client} ->
            lookup(Key, Qref, Client, Id, Predecessor, Successor, Store),
            node(Id, Predecessor, Successor, Store);
```

The `Qref` parameters will be used to tag the return message to the `Client`. This allows the client to identify the reply message and makes it easier to implement the client.

2.3 Adding an element

To add a new key value, we must first determine if our node is the node that should take care of the key. A node will take care of all keys from (but not including) the identifier of its predecessor to (and including) the identifier of itself. If we are not responsible, we send an add message to our successor.

```
add(Key, Value, Qref, Client, Id, {Pkey, _}, {_, Spid}, Store) ->
    case ......................  of
        true ->
              Client ! {Qref, ok},
              :
        false ->
              :
              :
    end.
```

2.4 Lookup procedure

The lookup procedure is very similar; we must do the same test to determine if we are responsible for the key. If so, we do a simple lookup in the local store and then send the reply to the requester. If it is not our responsibility, we will forward the request.

```
lookup(Key, Qref, Client, Id, {Pkey, _}, Successor, Store) ->
    case ................. of
        true ->
            Result = storage:lookup(Key, Store),
            Client ! {Qref, Result};
        false ->
            {_, Spid} = Successor,
              :
    end.
```

2.5 responsibility

Things are slightly complicated because new nodes might join the ring. A new node should, of course, take over part of the responsibility and must then, of course, also take over already added elements. We introduce one more message to the node, `{handover, Elements}`, that will be used to delegate responsibility.

```
        {handover, Elements} ->
            Merged = storage:merge(Store, Elements),
            node(Id, Predecessor, Successor, Merged);
```

When should this message be sent? It's a message from a node that has accepted us as its predecessor. This is only done when a node receives and handles a `notify` message. Go back to the implementation of the `notify/3` procedure. Handling a notify message could mean that we have to give part of a store away; we need to pass the store as an argument and return a tuple `{Predecessor, Store}`. The procedure `notify/4` could look like follows:

```
notify({Nkey, Npid}, Id, Predecessor, Store) ->
    case Predecessor of
        nil ->
            Keep = handover(Id, Store, Nkey, Npid),
                 :
        {Pkey,  _} ->
            case key:between(Nkey, Pkey, Id) of
                true ->
                      :
                      :
                false ->
                      :
            end
    end.
```

So, what's left is to implement the `handover/4` procedure. What should be done: split our `Store` based on the `NKey`. Which part should be kept, and which part should be handed over to the new predecessor? You have to check how your split function works; remember that a store contains the range $(Pkey, Id]$, that is from (not including $Pkey$ to (including) Id. What part should be handed over to our new predecessor?

```
handover(Id, Store, Nkey, Npid) ->
    {...., ....} = storage:split(Id, Nkey, Store),
    Npid ! {handover, Rest},
    Keep.
```

2.6 Performance

If we now have a distributed store that can handle new nodes added to the ring, we might try some performance testing. You need to be in a group with several machines to do this. Assume that we have eight machines and that we will use four in building the ring and four in testing the performance.

As a first test, we can have only one node in the ring and let the four test machines add 1000 elements to the ring and then do a lookup of the elements. Does it take longer for one machine to handle 4000 elements than four machines that do 1000 elements each? What is the limiting factor?

Implement a test procedure that adds several random key-value pairs into the system and keeps the keys in a list. You should then be able to do a lookup of all the keys and measure the time it takes. The lookup test should be given the name of a node to contact.

Now what happens if we add another node to the ring? How does the performance change? Does it matter if all test machines access the same node? Add two more nodes to the ring; any changes? How will things change if we have ten thousand elements?

3 First optional task for extra bonus: Handling failures

To handle failures, we need to detect if a node has failed. Both the successor and the predecessor need to detect this, and we will use the Erlang built-in procedure to `monitor` the health of a node. Start a new module `node3` and copy what we have from `node2`. As you will see, we will not have to make significant changes to what we have.

3.1 Successor of our successor

If our successor dies, we need a way to repair the ring. A simple strategy is to keep track of our successor's successor; we will call this node the `Next` node. A more general scheme is to keep track of a list of successors to make the system even more fault tolerant. We can survive from one node crashing at a time, but if two nodes in a row crash, we're doomed. That's ok; it makes life a bit simpler.

Extend the `node/4` procedure to a `node/5` procedure, including a parameter for the `Next` node. The `Next` node will not change unless our successor informs us about a change. Our successor should do so in the `status` message, so we extend this to {`status, Pred, Nx`}. The procedure `stabilize/3` must now be replaced by a `stabilize/4` procedure that also takes the new `Nx` node and returns not only the new successor but also the new next node.

```
{status, Pred, Nx} ->
    {Succ, Nxt} = stabilize(Pred, Nx, Id, Successor),
    node(Id, Predecessor, Succ, Nxt, Store);
```

Now `stabilize/4` need to do some more thinking. If our successor does not change, we should, of course, adopt the successor of our successor as our next node. However, if we detect that a new node has sneaked in between us and our former successor, then ... yes, then what? Do the necessary changes to `stabilize/4` so that it returns the correct successor and next node.

Almost done, but who sent the `status` message? This is sent by the `request/2` procedure. This procedure must be changed in order to send the correct message, a small change, and you are done.

3.2 Failure detection

We will use the `erlang:monitor/2` procedures to detect failures. The procedure returns a unique reference that can be used to determine which `'DOWN'` message belongs to which process. Since we need to keep track of both our successor and our predecessor, we will extend the representation of these nodes to a tuple {`Key, Ref, Pid`} where the `Ref` is a reference produced by the monitor procedure. We add wrapper functions for the built-in monitor procedures to make the code more readable.

```
monitor(Pid) ->
    erlang:monitor(process, Pid).

drop(nil) ->
    ok;
drop(Ref) ->
    erlang:demonitor(Ref, [flush]).
```

Now go through the code and change the representation of the predecessor and successor to include also the monitor reference. In the messages between nodes, we still send only the two-element tuple {Key, Pid} since the receiving node has no use of the reference element of the sending node. When a new node is adopted as a successor or predecessor, we need to de-monitor the old node and monitor the new one.

There are only four places where we need to create a new monitor reference and only two places where we de-monitor a node.

3.3 Houston, we have a problem

Now to the actual handling of failures. When a process is detected as having crashed (the process terminated, the Erlang machine died, or the computer stopped replying on heartbeats), a message will be sent to a monitoring process. Since we now monitor both our predecessor and successor, we should be open to handle both messages. Let's follow our principle of keeping the main loop free of gory details and handling all decisions in a procedure. The extra message handler could then look like follows:

```
{'DOWN', Ref, process, _, _} ->
    {Pred, Succ, Nxt}  = down(Ref, Predecessor, Successor, Next),
    node(Id, Pred, Succ, Nxt, Store);
```

The Ref obtained in the 'DOWN' message must now be compared to the saved references of our successor and predecessor. For clarity, we break this up into two clauses.

```
down(Ref, {_, Ref, _}, Successor, Next) ->
          :
down(Ref, Predecessor, {_, Ref, _}, {Nkey, Npid}) ->
          :
          :
    {Predecessor, {Nkey, Nref, Npid}, nil}.
```

If our predecessor died things are quite simple. There is no way for us to find the predecessor of our predecessor but if we set our predecessor to nil someone will sooner or later knock on our door and present themselves as possible predecessor.

If our successor dies, things are almost as simple. We will, of course, adopt our next node as our successor and then only have to remember two things: monitor the node and make sure that we run the stabilizing procedure.

You're done you have a fault-tolerant distributed storage...... well almost, if a node dies it will bring with it a part of the storage. If this is ok, we could stop here, if not we have to do some more work.

Another thing to ponder is what will happen if a node is falsely detected as being dead. What will happen if a node has only been temporally unavailable (and in the worst case, it might think that the rest of the network is gone)? How much would you gamble in trusting the `'DOWN'` message? report

4 Second optional task for extra bonus: Replication

The way to maintain the store in the face of dying nodes is of course to replicate information. How to replicate is a research area of its own so we will only do something simple that (almost) works.

4.1 Close, but no cigar

We can handle failures of one node at a time in our ring, so let's limit the replication scheme to be on the same level. If a node dies its local store should be replicated so its successor can take over the responsibility. Is there then a better place to replicate the store than at the successor? report

When we add a key-value element to our own store we also forward it to our successor as a {`replicate, Key, Value`} message. Each node will thus have a second store called the `Replica` where it can keep a duplicate of its predecessor's store. When a new node joins the ring it will as before takeover part of the store but also part of the replica.

If a node dies, its successor is, of course, responsible for the store held by the node. This means that the `Replica` should be merged with its own `Store`. Sounds simple does it not there are however some small details that make our solution less than perfect.

4.2 The devil in the detail

What does it mean that an element has been added to the store? Can we send a confirmation to the client and then replicate the element, what if we fail? If we want to handle failures we should make sure that a confirmation only is sent when an element has been properly added and replicated. This should not be too difficult to implement, who should send the confirmation?

A client that does not receive a confirmation could, of course, choose to re-send the message. What happens if the element was added the first time but the confirmation message was lost? Will we handle duplicates properly?

Another problem has to do with a joining node in combination with adding a new value. You have to think twice before realizing we have a problem. What would a solution look like?

Are there more devils? Will we have an implementation with no obvious faults or an implementation with obviously no faults? Take a copy of `node3`, call it `node4` and try to add a replication strategy.

5 Carrying on

There are more things that we could add or change and there is often not only one way of doing things. It becomes a trade-off between the properties that we want and efficiency in the implementation. Sometimes the properties are driving in opposite directions, such as high availability and consistency, and one has to make up one's mind about what property is actually needed most.

5.1 Routing

Routing is one thing that we have left out completely. If we only have twenty nodes it is less of a problem but if we have hundred nodes it does become important.

If network latency is high (think global Internet distances), then we need to do something. Should we even try to take network distances into account and route to nodes that are network-wise close to us (remember that the ring is an overlay and does not say anything about network distance)?

5.2 Replication

Is one replica enough or should we have two or three replicas? On what does this depend? It is of course related to how reliable nodes are and what reliability we need to provide, is it also dependent on the number of nodes in the ring?

Can we use the replicas for read operations and thus make use of the redundancy? How are replicas found, and can we distribute them in the ring to avoid hot spots?

5.3 Mutable object

As long as we only have one copy of an object, things are simple, but what if we want to update objects and some replicas are not updated? Do we have to use a two-phase-commit protocol to update all replicas in a consistent way? Could we have a trade-off between the expense of read and write operations? How relaxed can we be, and how does this relate to a shopping cart?

6 Conclusions

If you have followed this tutorial implementation, you should have a better understanding of how distributed hash tables work and how they are implemented. As you have seen, it's not that hard to maintain a ring structure, even in the face of failures. A distributed store also seems easy to implement, and replication could probably be solved. Consistency is a problem; can we guarantee that added values never are lost (given a maximum number of failed nodes)?

When things get complicated to implement, the performance might suffer. What is the advantage of the distributed store? Is it performance or fault tolerance? What should we optimize if we have to choose between them?