Chordy: a distributed hash table

Jordi Guitart

Adapted with permission from Johan Montelius (KTH)

November 17, 2012

Introduction

In this assignment you will implement a distributed hash table following the Chord scheme. In order to understand what you're about to do you should have a basic understanding of Chord and preferably read the article "Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications".

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 to represent the store. When we have the distributed store we can perform some benchmarks to see if the distribution actually 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 very much but if our successor dies we have to contact the next in line. In 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 of course 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.

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 very simple, to see what functionality introduces new messages.

1.1 Keys

Chord uses hashing of names to create unique keys for elements. We will not use a hash function, instead a random number generated is used 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 simply return a random number from 1 to 1.000.000.000 (30-bits), that will be large enough for our test. Using a hash function such as SHA-1 would give us 160 bits but let's keep things simple. Use the Erlang module `random` to generate numbers.

```
-module(key).
-export([generate/0, between/3]).

-define(N, 1000000000).

generate() ->
    random:uniform(?N).
```

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

```
between(Key, From, To) ->
    if
        From == To ->
            true;
        From < To ->
            (From < Key) and (Key =< To);
        From > To ->
            (From < Key) or (Key =< To)
    end.
```

1.2 Nodes

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 tuples {`Key, Pid`}.

The messages we need to maintain the ring are:

- {`key, Qref, Peer`} : a peer node wants to know our key
- {`notify, New`} : a new node informs us of its existence
- {`request, Peer`} : a potential predecessor needs to know our predecessor

- {`status, Pred`} : our successor informs us about its current predecessor

If we delay all the tricky decision to subroutines the implementation of the node process could look like this (insert this code in the module `node1`):

```
node(MyKey, Predecessor, Successor) ->
    receive
        {key, Qref, PeerPid} ->
            PeerPid ! {Qref, MyKey},
            node(MyKey, Predecessor, Successor);
        {notify, New} ->
            Pred = notify(New, MyKey, Predecessor),
            node(MyKey, Pred, Successor);
        {request, Peer} ->
            request(Peer, Predecessor),
            node(MyKey, Predecessor, Successor);
        {status, Pred} ->
            Succ = stabilize(Pred, MyKey, Successor),
            node(MyKey, Predecessor, Succ)
   end.
```

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

1.3 `stabilize`

The periodic stabilization 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 (by means of the `stabilize/3` function) if the ring is stable or if the successor needs to be notified about the node existence through a {`notify, {Key, self()`} message.

Below is a skeleton for the `stabilize/3` procedure. The `Pred` argument is the current predecessor of our successor. If this is `nil` we should of course notify it about our existence. If it is pointing 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 are to slide in between the two nodes or we should 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 (by sending ourselves a `stabilize` message). If we should be in between the nodes we notify our successor of our existence. At the end, this function must return our updated successor.

```
stabilize(Pred, MyKey, Successor) ->
  {Skey, Spid} = Successor,
  case Pred of
      nil ->
          %% TODO: ADD SOME CODE
          Successor;
      {MyKey, _} ->
          Successor;
      {Skey, _} ->
          %% TODO: ADD SOME CODE
          Successor;
      {Xkey, Xpid} ->
            case key:between(Xkey, MyKey, Skey) of
                true ->
                    %% TODO: ADD SOME CODE
                    %% TODO: ADD SOME CODE
                false ->
                    %% TODO: ADD SOME CODE
                    Successor
            end
    end.
```

1.4 request

The stabilization procedure must be done with 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 predefined interval. In our scenario we might set the interval to be 1000 ms in order to slowly trace what messages are sent. When the `stabilize` message is received, the process will send a REQUEST message to the successor.

```
-define(Stabilize, 1000).

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

The procedure `schedule_stabilize/0` is called when a node is created. When the node receives a `stabilize` message it will call `stabilize/1` procedure.

```
%% TO BE ADDED TO FUNCTION node/3
stabilize ->
    stabilize(Successor),
    node(MyKey, 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()}.
```

When the `request` message is picked up by a process, it calls the `request/2` procedure. We should of course only inform the peer that sent the request about our predecessor as in the code below.

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

Open Questions. What are the pros and cons of a more frequent stabilizing procedure?

1.5 `notify`

Sending a notification is a way for a node to make a friendly proposal that it might be our proper predecessor. We cannot take their word for it, so we have to do our own investigation.

```
notify({Nkey, Npid}, MyKey, Predecessor) ->
    case Predecessor of
        nil ->
            %% TODO: ADD SOME CODE
        {Pkey,  _} ->
            case key:between(Nkey, Pkey, MyKey) of
                true ->
                    %% TODO: ADD SOME CODE
                false ->
                    Predecessor
            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.

Open Questions. 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?

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 later with the second argument set to `nil`.

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

start(MyKey, PeerPid) ->
    timer:start(),
    spawn(fun() -> init(MyKey, PeerPid) end).
```

In the `init/2` procedure we set our predecessor to `nil`, connect to our successor and schedule the stabilizing procedure. 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(MyKey, PeerPid) ->
    Predecessor = nil,
    {ok, Successor} = connect(MyKey, PeerPid),
    schedule_stabilize(),
    node(MyKey, Predecessor, Successor).
```

The `connect/2` procedure is divided into two cases, depending on whether we are 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 successor. If we're connecting to an existing ring we send a `key` message to the peer node that we have been given (we start with this node as our successor) and wait for a reply. Below is the skeleton code for the `connect/2` procedure.

```
-define(Timeout, 5000).

connect(MyKey, nil) ->
    {ok, {... , ...}};    %% TODO: ADD SOME CODE
connect(_, PeerPid) ->
    Qref = make_ref(),
    PeerPid ! {key, Qref, self()},
    receive
```

```
        {Qref, Skey} ->
            {ok, {... , ...}}    %% TODO: ADD SOME CODE
    after ?Timeout ->
        io:format("Timeout: no response from ~w~n", [PeerPid])
    end.
```

Notice how the unique reference `Qref` is used to trap exactly the message we're looking for. It might be over-kill 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 5s) we return an error.

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

1.7 Testing

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

```
%% TO BE ADDED TO FUNCTION node/3
probe ->
    create_probe(MyKey, Successor),
    node(MyKey, Predecessor, Successor);
{probe, MyKey, Nodes, T} ->
    remove_probe(MyKey, Nodes, T),
    node(MyKey, Predecessor, Successor);
{probe, RefKey, Nodes, T} ->
    forward_probe(RefKey, [MyKey|Nodes], T, Successor),
    node(MyKey, Predecessor, Successor);
```

To check if the ring is actually connected we can introduce a {`probe, Probe, Nodes, Time`} message. If the `Probe` is equal to the `MyKey` 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:now()` to get microsecond accuracy (this is local time so the timestamp does not mean anything on other nodes). There is also a function `timer:now_diff/2` that can come in handy.

```
create_probe(MyKey, {_, Spid}) ->
    Spid ! {probe, MyKey, [MyKey], erlang:now()},
    io:format("Create probe ~w!~n", [MyKey]).

remove_probe(MyKey, Nodes, T) ->
```

```
    Time = timer:now_diff(erlang:now(), T),
    io:format("Received probe ~w in ~w ms Ring: ~w~n", [MyKey, Time, Nodes]).

forward_probe(RefKey, Nodes, T, {_, Spid}) ->
    Spid ! {probe, RefKey, Nodes, T},
    io:format("Forward probe ~w!~n", [RefKey]).
```

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 by giving {`node, 'chordy@192.168.1.32'`} (or something similar). 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 Local storage

The first thing we need to implement is a local storage. This can 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 functions to: `create/0` a new store, `add/3` a key-value pair, do a `lookup/2` on a key, `split/3` the storage in two parts given a key and `merge/2` two stores. The split and merge functions will be used when a new node joins the ring and should take over part of the store.

```
-module(storage).
-export([create/0, add/3, lookup/2, merge/2, split/3]).

create() ->
  [].

add(Key, Value, Store) ->
  case lists:keysearch(Key, 1, Store) of
      {value, {_, _}} ->
          io:format("[Store:Add] Key ~w already exists!~n", [Key]),
```

```
          Store;
      false ->
          [{Key, Value}|Store]
  end.

lookup(Key, Store) ->
  case lists:keysearch(Key, 1, Store) of
    {value, {Key, Value}} ->
      Value;
    false ->
      io:format("[Store:Lookup] Key ~w does not exist!~n", [Key]),
      -1
  end.

merge(S1, S2) ->
  lists:keymerge(1, S1, S2).

split(Id, Key, Store) ->
  lists:partition(fun({K,_}) -> key:between(K, Key, Id) end, 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.

```
%% TO BE ADDED TO FUNCTION node/4
{add, Key, Value, Qref, Client} ->
    Added = add(Key, Value, Qref, Client,
                MyKey, Predecessor, Successor, Store),
    node(MyKey, Predecessor, Successor, Added);
{lookup, Key, Qref, Client} ->
    lookup(Key, Qref, Client, MyKey, Predecessor, Successor, Store),
    node(MyKey, 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 pair 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 so we add the key-value pair in the local store (using the function `add` from the `storage` module). If we are not responsible we simply send an `add` message to our successor.

```
add(Key, Value, Qref, Client, MyKey, {Pkey, _}, {_, Spid}, Store) ->
    case key:between(... , ... , ...)  of    %% TODO: ADD SOME CODE
        true ->
              Added = ... ,    %% TODO: ADD SOME CODE
              Client ! {Qref, ok},
              Added;
        false ->
              %% TODO: ADD SOME CODE
              Store
    end.
```

2.4 Looking up an element

The lookup procedure is very similar, we need to do the same test to determine if we are responsible for the key. If so we do a simple lookup in the local store (using the function `lookup` from the `storage` module) and then send the reply to the requester. If it is not our responsibility we simply forward the request.

```
lookup(Key, Qref, Client, MyKey, {Pkey, _}, {_, Spid}, Store) ->
    case key:between(... , ... , ...)  of    %% TODO: ADD SOME CODE
        true ->
            Result = ... ,    %% TODO: ADD SOME CODE
            Client ! {Qref, Result};
        false ->
            %% TODO: ADD SOME CODE
    end.
```

2.5 Elements responsibility

Things are slightly more complicate by the fact that 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.

```
%% TO BE ADDED TO FUNCTION node/4
{handover, Elements} ->
    Merged = storage:merge(Store, Elements),
    node(MyKey, Predecessor, Successor, Merged);
```

When should this message be sent? It's a message from a node that has accepted us as their 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 of 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 also return a tuple {`Predecessor, Store`}. The procedure `notify/4` could look like follows:

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

So, what's left is simply 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 the split function works, remember that a store contains the range `(Pkey, MyKey]`, that is from (not including) `Pkey` to (including) `MyKey`.

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

2.6 Performance

Open Questions. If we now have a distributed store that can handle new nodes that are added to the ring we might try some performance testing. You need 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 one node only in the ring and let the four test machines add a number (e.g., 1000) of random elements (key-value pairs) to the ring and then do a lookup of the elements, measuring the time it takes. You can use the test procedure given in 'Appendix A', which should

be given with the name of the node to contact. Does it take longer for one machine to handle 4000 elements rather than four machines that do 1000 elements each? What is the limiting factor?

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 a ten thousand elements?

3 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 procedures 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 do large 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 will be able to survive from one node crashing at a time but if two nodes in a row crash we're doomed. That's ok for now, 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`}. This is sent by the `request/2` procedure. This procedure, which is now `request/3`, must be adapted in order to send the correct message.

```
%% TO BE UPDATED IN FUNCTION node/5
{request, Peer} ->
    request(Peer, Predecessor, Successor),
    node(MyKey, Predecessor, Successor, Next, Store);

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

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.

```
%% TO BE UPDATED IN FUNCTION node/5
{status, Pred, Nx} ->
    {Succ, Nxt} = stabilize(Pred, Nx, MyKey, Successor),
    node(MyKey, 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 you must do the necessary changes to `stabilize/4` so that it returns the correct successor and next node.

3.2 Failure detection

We will use the `erlang:monitor/2` procedure to detect failures. The procedure returns a unique reference that can be used to determine which `'DOWN'` message belong 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. To make the code more readable we add wrapper functions for the built-in monitor procedures.

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

demonit(nil) ->
    ok;
demonit(MonitorRef) ->
    erlang:demonitor(MonitorRef, [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 successor or predecessor** we need to demonitor the old node and monitor the new.

You must modify the `connect`, `stabilize`, and `notify` procedures to create a new monitor reference and the `stabilize` and `notify` procedures to demonitor a node.

3.3 Failure handling

Now let's go 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 heart beats) 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 handle all decisions in a procedure. The extra message handler could then look like follows:

```
%% TO BE ADDED TO FUNCTION node/5
{'DOWN', Ref, process, _, _} ->
    {Pred, Succ, Nxt}  = down(Ref, Predecessor, Successor, Next),
    node(MyKey, 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) ->
    {nil, Successor, Next};
down(Ref, Predecessor, {_, Ref, _}, {Nkey, Npid}) ->
    %% TODO: ADD SOME CODE
    %% TODO: ADD SOME CODE
    {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 them self as a 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 (by sending ourselves a `stabilize` message).

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.

Open Questions. Another thing to ponder is what will happen if a node is falsely detected of 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?

4 Replication

The way to maintain the store in 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.

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?

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 the predecessor's replica. Modify the `node` and `handover` procedures accordingly.

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`. The predecessor of the dead node must also update the `Replica` in the successor of the dead node with its `Store`.

Take a copy of `node3`, call it `node4` and try to add a replication strategy.

5 Carrying on

If you are willing to enhance your Chord implementation (even though this is not part of this assignment), you can try to add a finger table for each node, which will allow us to find any given key in $log(n)$ hops. This is of course important if we have a large ring.

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 seams easy to implement and replication could probably be solved.

Appendix A: test program

```
-module(chordy).
-export([connect/1, test/2]).

% 'Peer' stands for the node used to join the system
connect(Peer) ->
    spawn(fun() -> wait(Peer) end).

wait(Peer) ->
    receive
        {add, Key, Value} ->
            Ref = make_ref(),
            Peer ! {add, Key, Value, Ref, self()},
            receive
                {Ref, ok} ->
                    io:format("[Add] Key added correctly~n")
            end,
            wait(Peer);
        {lookup, Key} ->
            Ref = make_ref(),
            Peer ! {lookup, Key, Ref, self()},
            receive
                {Ref, Value} ->
                    io:format("[Lookup] Key: ~w Value: ~w~n",[Key, Value])
            end,
            wait(Peer);
        {test, NumRequests} ->
            test(Peer, NumRequests),
            wait(Peer)
    end.

test(Peer, NumRequests) ->
    Begin = erlang:now(),
    {KeysAdd, RefsAdd} = add(Peer, NumRequests, [], []),
    io:format("Requests done. Waiting for answers...~n"),
    waitAdds(RefsAdd),
    io:format("Answers received. Making lookups...~n"),
    RefsLook = lookup(Peer, KeysAdd, []),
    io:format("Lookups received. Waiting for values...~n"),
    Values = waitLookups(RefsLook, []),
    End = erlang:now(),
    Elapsed = timer:now_diff(End, Begin)/1000.0,
    io:format("Values received. Checking correctness...~n"),
```

```
    checkLookups(Values, KeysAdd),
    io:format("Elapsed Time: ~w ms~n", [Elapsed]).

add(_, 0, KeysDone, RefsDone) ->
    {KeysDone, RefsDone};
add(Peer, NumRequests, KeysDone, RefsDone) ->
    Key = key:generate(),
    Qref = make_ref(),
    Peer ! {add, Key, Key, Qref, self()},
    add(Peer, NumRequests-1, [{Key}|KeysDone], [Qref|RefsDone]).

waitAdds([]) ->
    ok;
waitAdds([Qref|Refs]) ->
    receive
        {Qref, ok} ->
            waitAdds(Refs)
    end.

lookup(_, [], RefsLook) ->
    RefsLook;
lookup(Peer, [{Key}|KeysAdd], RefsLook) ->
    Qref = make_ref(),
    Peer ! {lookup, Key, Qref, self()},
    lookup(Peer, KeysAdd, [Qref|RefsLook]).

waitLookups([], Values) ->
    Values;
waitLookups([Qref|RefsLook], Values) ->
    receive
        {Qref, Value} ->
            waitLookups(RefsLook, [{Value}|Values])
    end.

checkLookups([], _) ->
    io:format("Everything correct!~n"),
    ok;
checkLookups([{Value}|Values], KeysAdd) ->
    case lists:keymember(Value, 1, KeysAdd) of
        true ->
            KeysDeleted = lists:keydelete(Value, 1, KeysAdd),
            checkLookups(Values, KeysDeleted);
        false ->
            io:format("Something's wrong!~nValues: ~w~nKeys: ~w~n",
```

```
        [[{Value}|Values],KeysAdd]),
        false
end.
```