Project 3 - Report

Mathias Brekkan and Ruiyang Li

The code snippet below shows how we have master instructing nodes to join or create a network. After this two methods sendAllRequestsStart and masterWaitForFinish will start executing the simulation. The first method gives the nodes the all clear to start sending requests. The second method waits for all the nodes to finish executing the requests, after which it takes the statistics sent from the node to calculate the average hop count.

```
master(NumberOfNodes, NumberOfRequests, M, Nodes, NumberOfNodesToAdd) ->
    if
        NumberOfNodesToAdd > 0 ->
            receive
                {create, Node} -> % Register node
                    io:format("Master initiate node:\n"),
                    io:format("~w~n", [Node]),
                    io:format("Current Nodes:~n"),
                    UpdatedNodes = [Node | Nodes],
                    printList(UpdatedNodes),
                    Node#node.pid ! {create, NumberOfRequests},
                    master(NumberOfNodes, NumberOfRequests, M, UpdatedNodes, NumberOfNodesToAdd - 1);
                {join, Node} ->
                    io:format("Master joining node: \n"),
                    io:format("~w~n", [Node]),
                    io:format("Current Nodes:~n"),
                    UpdatedNodes = [Node | Nodes],
                    printList(UpdatedNodes),
                    %%% find a random existing node in the network to initiate join
                    io:format(">>>>>>>>>>>>>>>>>>>>>>>>>>>>>~w~n", [getRandomNumber(1, length(Nodes))]),
                    Node#node.pid ! {join, lists:nth(getRandomNumber(1, length(Nodes)), Nodes), NumberOfRequests},
                    master(NumberOfNodes, NumberOfRequests, M, UpdatedNodes, NumberOfNodesToAdd - 1)
            end;
        true ->
            sendAllRequestsStart(NumberOfNodes, 1, Nodes),
            masterWaitForFinish(NumberOfNodes, NumberOfRequests, 0, NumberOfNodes)
    end.

sendAllRequestsStart(_, _, []) -> ok;
sendAllRequestsStart(NumberOfNodes, CurrentIndex, [Node | Tail]) ->
    Node #node.pid ! {startSendingRequests},
    sendAllRequestsStart(NumberOfNodes, CurrentIndex + 1, Tail).
```

```
masterWaitForFinish(TotalNumberOfNodes, NumberOfRequests, NumberOfHopsSoFar, NumberOfNodesLeft) ->
    if
        NumberOfNodesLeft == 0 ->
            io:format("Average hops: ~w~n", [NumberOfHopsSoFar/(TotalNumberOfNodes * NumberOfRequests)]),
            io:format("Finished running program...");
        true ->
            receive
                {finito, NewNumberOfHops} -> % Node completed all required requests
                    io:format("||||||||||||||||||||||||||||||||||||||||~w~n", [NumberOfNodesLeft]),
                    masterWaitForFinish(TotalNumberOfNodes, NumberOfRequests, NumberOfHopsSoFar + NewNumberOfHops, NumberOfNodesLeft - 1)
            end
    end.
```

The following code snippet shows the actions a node takes when being created. It creates a random name and then waits for an order whether to create or join a network.

```
nodeInit(MasterNode, IsFirstNode) ->
    RandomName = getRandomString(8),
    Node = #node{id = getHash(RandomName), pid = self(), key = #key{key = RandomName, id = getHash(RandomName)}},
    case IsFirstNode of
        true ->
            master ! {create, Node};
        false ->
            master ! {join, Node}
    end,

    receive

        {create, NumberOfRequests} ->
            Predecessor = Node,
            Successor = Node,
            FingerList = lists:duplicate(getM(), Node),
            io:format("Node is online:~n"),
            io:format("~w~n", [Node]),
            operate(MasterNode, NumberOfRequests, Node, Predecessor, Successor, FingerList, false, 0);
        {join, KnownNode, NumberOfRequests} ->
            Predecessor = nil,
            KnownNode#node.pid ! {findSuccessor, Node#node.key, Node, 0},
            io:format("11111111111111111111111"),

            receive
                {found, Key, FoundWhere, NumHops} -> % NumHops must be ignored in this case
                    io:format("Node is online:~n"),
                    io:format("~w~n", [Node]),
                    %%% node_p needs to notify node_s that it needs to change predecessor
                    FoundWhere#node.pid ! {changePredecessor, Node},
                    FingerList = lists:duplicate(getM(), FoundWhere),
                    operate(MasterNode, NumberOfRequests, Node, Predecessor, FoundWhere, FingerList, false, 0)
            end
    end.
```

The following code snippet shows the inner workings of the main method for the nodes. Here they wait for messages from other nodes. Aprox every second it will try to stabilize, fix it's fingers and send a random request.

```
operate(MasterNode, NumberOfRequestsLeft, Node, Predecessor, Successor, FingerList, CanSendRequests, TotalNumHops) ->
    io:format("~p~n", [CanSendRequests]),
    receive
        {showMeYourFingers} ->
            printList(FingerList),
            operate(MasterNode, NumberOfRequestsLeft, Node, Predecessor, Successor, FingerList, CanSendRequests, TotalNumHops);
        {showMeYourSuccessor} ->
            io:format("My Successor is:~n ~w~n", [Successor]),
            operate(MasterNode, NumberOfRequestsLeft, Node, Predecessor, Successor, FingerList, CanSendRequests, TotalNumHops);
        {showMeYourPredecessor} ->
            io:format("My Predecessor is:~n ~w~n", [Predecessor]),
            operate(MasterNode, NumberOfRequestsLeft, Node, Predecessor, Successor, FingerList, CanSendRequests, TotalNumHops);
        {startSendingRequests} ->
            operate(MasterNode, NumberOfRequestsLeft, Node, Predecessor, Successor, FingerList, true, TotalNumHops);
        {whatsYourPredecessor, WhoAsked} ->
            WhoAsked#node.pid ! {predecessor, Predecessor},
            operate(MasterNode, NumberOfRequestsLeft, Node, Predecessor, Successor, FingerList, CanSendRequests, TotalNumHops);
        {whatsYourSuccessor, WhoAsked} ->
            WhoAsked#node.pid ! {successor, Successor},
            operate(MasterNode, NumberOfRequestsLeft, Node, Predecessor, Successor, FingerList, CanSendRequests, TotalNumHops);
        {findSuccessor, Key, WhoAsked, NumHops} ->
            %io:fwrite("Node:\n"),
            %io:fwrite("~w~n", [self()]),
            %io:fwrite("received findSuccessor request from:\n"),
            %io:fwrite("~w~n", [WhoAsked]),
            findSuccessor(Key, Node, FingerList, Successor, WhoAsked, NumHops + 1),
            operate(MasterNode, NumberOfRequestsLeft, Node, Predecessor, Successor, FingerList, CanSendRequests, TotalNumHops);

        {found, Key, FoundWhere, NumHops} ->
            io:format("Node: ~p~n", [self()]),
            io:format("Key: ~p~n", [Key#key.key]),
            io:format("Key identifier: ~p~n", [Key#key.id]),
            io:format("Found at node: ~p~n", [FoundWhere#node.pid]),
            io:format("Which as identifier: ~p~n", [FoundWhere#node.id]),
            io:format("Hops: ~p~n", [NumHops]),
            operate(MasterNode, NumberOfRequestsLeft - 1, Node, Predecessor, Successor, FingerList, CanSendRequests, TotalNumHops + NumHops);

        {notify, NewPredecessor} ->
            io:format("Node:~n"),
            io:format("~w~n", [self()]),
            io:format("Is notified of:~n"),
            io:format("~w~n", [NewPredecessor]),
            if
                (Predecessor == nil) or ((NewPredecessor#node.id > Node#node.id) and (NewPredecessor#node.id < Predecessor#node.id)) ->
                    operate(MasterNode, NumberOfRequestsLeft - 1, Node, NewPredecessor, Successor, FingerList, CanSendRequests, TotalNumHops);
                true ->
                    operate(MasterNode, NumberOfRequestsLeft - 1, Node, Predecessor, Successor, FingerList, CanSendRequests, TotalNumHops)
            end;

        {changePredecessor, NewPredecessor} ->
            io:format("Node notified changePredecessor:~n"),
            io:format("~w~n", [Node]),
            io:format("to ~w~n~n", [NewPredecessor]),
            operate(MasterNode, NumberOfRequestsLeft, Node, NewPredecessor, Successor, FingerList, CanSendRequests, TotalNumHops)

        after 1000 ->
            io:format("Node run stablize:~n"),
            io:format("~w~n", [Node]),
            NewSuccessor = stabilize(Node, Successor, Predecessor),
            NewSuccessor#node.pid ! {notify, Node},

            io:format("Node run Fix Finger~n"),
            if
                Node == Successor ->
                    NewFingerList = FingerList;
                true ->
                    NewFingerList = fixFinger(FingerList, Node, Successor, getM(), 0, [])
            end,

            if
                CanSendRequests and NumberOfRequestsLeft > 0 ->
                    RandomKeyValue = getRandomString(8),
                    HashedKey = getHash(RandomKeyValue),
                    NewId = HashedKey rem round(math:pow(2, getM())),
                    NewKey = #key{id = NewId, key = RandomKeyValue},
                    findSuccessor(NewKey, Node, NewFingerList, Successor, Node, 0),
                    operate(MasterNode, NumberOfRequestsLeft, Node, Predecessor, NewSuccessor, NewFingerList, CanSendRequests, TotalNumHops);
                CanSendRequests == false ->
                    operate(MasterNode, NumberOfRequestsLeft, Node, Predecessor, NewSuccessor, NewFingerList, CanSendRequests, TotalNumHops);
                true ->
                    master ! {finito, TotalNumHops}
            end
    end.
```

The code compiles, but it has some problems that make it not run to completion. The most challenging part of the project we found was to coordinate time out of different nodes. Here is the code for fix finger table and stabilize. The two functionalities need to coordinate across different nodes so that no process is stuck when the other node is busy.

```
fixFinger(_, _, _, M, M, NewList) ->
    io:format("++++++++++++++++++++++++++++++++++++++++"),
    lists:reverse(NewList);
fixFinger(FingerList, Self, KnownNode, M, I, NewList) ->

    Key = #key{id = round(Self#node.id + math:pow(2, I)) rem round(math:pow(2, getM())), key = nil},

    Self#node.pid ! {findSuccessor, Key, Self, 0},
    io:format("----------------------------------------~w~n", [I]),
    receive
        {found, Key, Successor, NumHops} ->
            io:format("========================================~w~n", [I]),
            fixFinger(FingerList, Self, KnownNode, M, I + 1, [Successor | NewList])
        after 100 ->
            io:format("Fix Finger time out~n"),
            FingerList
    end.


stabilize(Self, Successor, Predecessor) ->
    if
        Self == Successor ->
            Predecessor;
        true ->
            Successor#node.pid ! {whatsYourPredecessor, Self},
            CircleSize = getCircleSize(),
            receive
                {predecessor, SuccessorPecessor} ->
                    if
                        (Successor#node.id =< Self#node.id) and (SuccessorPecessor#node.id > Self#node.id) and (SuccessorPecessor#node.id < CircleSize) ->
                            io:format("New Node detected:~n"),
                            io:format("by ~w~n", [Self]),
                            io:format("of ~w~n", [SuccessorPecessor]),
                            SuccessorPecessor;

                        (SuccessorPecessor#node.id > Self#node.id) and (SuccessorPecessor#node.id < Successor#node.id) ->
                            io:format("New Node detected:~n"),
                            io:format("by ~w~n", [Self]),
                            io:format("of ~w~n", [SuccessorPecessor]),
                            SuccessorPecessor;

                        true ->
                            Successor
                    end
                after 50 ->
                    io:format("Time out~n"),
                    Successor
            end
    end.
```

This portion of the code is responsible for n(log(n)) lookup of the successor for an identifier. The most critical part of this portion of the code is to check the edge cases. One such edge case is when the node's successor is the first node. In this case, the successor's identifier is smaller than the node. This condition is guarded with.

```
        (Node#node.id > Successor#node.id) and ((Key#key.id > Node#node.id) and (Key#key.id =< CircleSize)) ->
```

```
findSuccessor(Key, Node, FingerList, Successor, WhoAsked, NumHops) ->
    io:format("~w~n", [Key]),
    io:format("~w~n", [Node]),
    io:format("~w~n", [Successor]),
    %%% io:fwrite("Test\n"),
    %%% io:write(Key#key.id),
    %%% io:fwrite(" - "),
    %%% io:write(Node#node.id),
    %%% io:fwrite("\n"),
    CircleSize = getCircleSize(),
    if
        (Node#node.id == Successor#node.id) ->
            io:fwrite("Goal case1\n"),
            WhoAsked#node.pid ! {found, Key, Successor, NumHops};



        (Node#node.id > Successor#node.id) and ((Key#key.id > Node#node.id) and (Key#key.id =< CircleSize)) ->
            io:fwrite("Goal case2\n"),
            WhoAsked#node.pid ! {found, Key, Successor, NumHops};

        (Key#key.id > Node#node.id) and (Key#key.id =< Successor#node.id) ->
            io:fwrite("Goal case3\n"),
            %%% io:fwrite("Goal\n"),
            WhoAsked#node.pid ! {found, Key, Successor, NumHops};

        true ->
            io:format("case 3 keep looking~n"),
            ClosestPrecedingNode = closestPrecedingNode(Key, Node, FingerList, getM(), WhoAsked),
            ClosestPrecedingNode#node.pid ! {findSuccessor, Key, WhoAsked, NumHops}
    end.

closestPrecedingNode(_, Node, _, 0, WhoAsked) ->
    Node;
closestPrecedingNode(Key, Node, FingerList, I, WhoAsked) ->
    FingerListElement = lists:nth(I, FingerList),
    if
        (FingerListElement#node.id > Node#node.id) and (FingerListElement#node.id < Key#key.id) ->
            FingerListElement;
        true ->
            closestPrecedingNode(Key, Node, FingerList, I - 1, WhoAsked)
    end.
```