FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

1st Phase Report

Algoritmos e Sistemas Distribuídos

Daniel Flamino nº45465, Diogo Silvério nº45679 e Rita Macedo nº46033

1. Introduction

In the first phase of the project we aimed to implement a decentralized publish-subscribe protocol on top of a peer-to-peer distributed hash table. In the chapter overview we explain how the system is organized, then we explain each layer of the architecture and what is their main objectives. In the 4th chapter we show the pseudo code of our implementation and finally in the last chapter we show the results of our application testing and explain them.

2. Overview

The Project was implemented in the Akka/Scala ecosystem. Three layers were implemented each one representing an actor on the system:

- The Main/Test where we start the protocols and verify that they are in fact communicating with each other and working correctly;
- the Publish-Subscribe protocol that manages the subscriptions and publications that the nodes make to some topic;
- and the Distributed Hash table that is responsible for the communication between nodes, building and maintaining the structured overlay network.

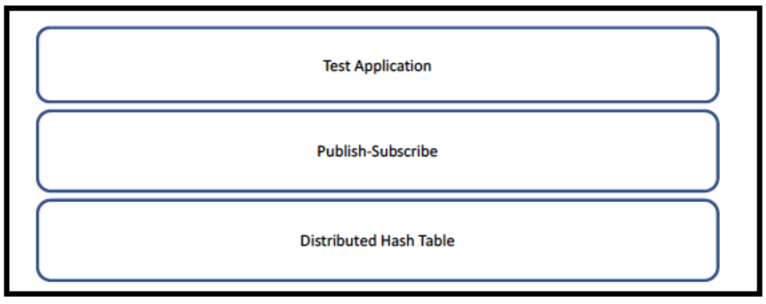

Figure 1. Layers of the System

Each node of the system contains one instance of each of the three layers. Each layer communicates with the layer above and under it, and with the same DHT layer of the other nodes. The Structured Overlay Network built by the Chord protocol provides efficient topic routing meaning that finding a topic in the layer will be much easier and efficient than in an unstructured overlay.

3. Detailed Overview

3.1. DHT

For the implementation of the Distributed Hash table we used the Chord protocol described in [1]. The objective of this layer is to manage all the nodes in the system in a decentralized way, whilst trying to have a good load balance through the usage of Consistent Hashing. The main idea is that with a given key assigned by a consistent hash function, the key can be mapped to a specific node in the system, that has the topic corresponding to that key. Each node has an identifier and identifiers are ordered in an identifier circle forming a ring. A key is assigned to the first node whose identifier is equal or follows its value clockwise in the circle. This allows to do efficient lookups for a given resource without knowing who has it, since the hashed value of the resource will be the same as the value of the key of the resource owner. Each node keeps a table with a small amount of routing information allowing it to locate keys (resources/nodes) requiring only $O(\log N)$ messages. This is what is called the Finger table, which has m entries (m being a positive integer number known to every node in the network) and contains the node's successor and fingers (the successor itself is the first entry in the finger table). The fingers are calculated using the following mathematical expression $(n + 2^{k-1}) mod\ 2^m, 1 \ \leq k \leq m,$ where n is the node's id, k is the table entry number and m is a constant which defines the maximum size of a key (2^m bits).

The network can have 2^m nodes but that might not always be the case so it's possible that a node's finger doesn't exist in the network. Because of this the node's finger will be the first node whose identifier is equal or follows clockwise the result of the expression.

All the nodes have a balanced work load on them thanks to consistent hashing.

This layer is the one that deals with nodes who join or leave the system, having methods to update the finger table of each node as the network evolves over time.

To finalize, this protocol also deals with node failure. Each node receives the heartbeat of its predecessor and its successor, having a limited time to assume the node is still alive in the system or not. If some node's successor dies the node makes its successor's successor its own successor in order to repair the broken ring. Nodes also check if their predecessor is alive periodically the same way. In case of a predecessor failure this is set to null. The stabilize method will eventually correct this.

This network repair strategy allows the simultaneously failure of $N/2$ interleaved nodes in the network. It should be noted that [1] does mention the usage of successor list which could handle the case of two or more consecutive nodes failing, as long as the number of consecutive fails doesn't exceed the size of that list.

3.2. Publish-Subscribe

The Publish-Subscribe layer implementation is based on the Scribe protocol explained in [2], and for that reason some of the pseudo code was taken from it. This protocol has the main objective of registering nodes interested in topics and delivering messages published under said topics to nodes subscribed to it, as is typical of a publish-subscribe protocol.

Each topic has an id, one node parent (the node that just passed a published message) and a list of children (nodes that subscribed to the topic). This allows the protocol to build a multicast tree by joining the routes from each subscriber to a rendez-vous point associated with a topic. Nodes that receive a subscribe message become forwarders which means they will send messages published under certain topics to their subscribers even if they're not interested in it themselves.

The node which receives the command from the upper layer to create, subscribe, publish or unsubscribe from a topic sends it downwards to the DHT layer via the route operation in order to reach the owner node of the topic. Once again, that's the one whose ID matches the hashed value of the message. The message will be forwarded through Chord which will surface at every node it passes by sending the Forward message to pub-sub layer. In this case, Scribe only cares about subscription messages in order to turn a passed node into a forwarder to decrease the number of sent messages, in the case of two nodes which must pass by the forwarder to get to the resource owner. In any case, the message will be sent down again into the Chord layer to continue its route towards the owner node.

Publish messages are special because once they reach the owner, it will then be disseminated via the multicast tree to all subscribers. This also allows all forwarders to know who their parent (the node between them and the owner) is, which is useful when the time comes to unsubscribe from a topic. If a forwarder isn't itself subscribed to a topic and if all of its children eventually unsubscribe, then there's no point in continuing to receive messages to broadcast, so it sends a message to its parent informing them that they should no longer propagate messages to us about this topic.

3.3. Test Application

The Main/Test layer is almost split vertically into two sections, the command line interface and the actual Test actor. The command line interface is available in any node and allows the user to input commands for the node to execute. The list of available commands follows:

- `create <TOPIC>`: creates a new topic
- `subscribe <TOPIC>`: subscribes to a topic
- `publish <TOPIC> <MESSAGE>`: publishes a message to topic
- `unsubscribe <TOPIC>`: unsubscribes from a topic
- `fingertable`: prints the finger table for the current node
- `subs`: prints the subscriptions of the current node
- `topics`: prints the topics table of the current node
- `test <create/subscribe/publish/print>`: runs the tester
- `help`: shows the available commands
- `exit`: exits the program

The Tester is the actual Actor which should only be used when launching the network through the provided test.sh file (only tested on git-bash for Windows 10) and only on the Main Tester terminal which should be the first one to open. The test.sh should launch 20 terminals for each of the 20 nodes. There are four commands already mentioned above:

- create, creates 50 topics in the network
- subscribe, tells the 20 nodes to subscribe to 5 topics each
- publish, tells the 20 nodes to publish a message to 5 random topics (don't need to be subscribed to them)

- print, prints out the number of logged creates, subscribes, sent and delivered messages, as well as the number of expected delivered and the percentage of delivered

4. Pseudo Code

4.1. DHT (Chord)

```
State:
    m; //size of id = 2^m
    thisNode; //the information about this node, its id, ip and port
    fingerTable; //Table of nodes with which this node communicates
    start; //variable to calculate the start of the interval of the node on the finger table
    end; //variable to calculate the end of the interval of the node on the finger table
    next; //Next finger table index to fix
    predecessor; //node that precedes this node
    successorSuccessor; //node that succeeds this node’s successor
    heartBeatCounterPredecessor;  //time counter to check if the predecessor is alive;
    heartBeatLimitPredecessor;  //limit time to receive the predecessor heartbeat;
    heartBeatCounterSuccessor; // time counter to check if the successor is alive;
    heartBeatLimit Sucessor; // limit time to receive the successor heartbeat;
    join complete; //boolean verifying if the join process has been completed

Upon Init do:
    m <- 8;
    node <- node(createID(), myHostName, myPort);
    fingerTable <- {};
    foreach k ∈ m do:
        start <- calculateStart (thisNode.id, k);
        end <- calculateStart (thisNode.id, k+1);
        node <- thisNode;
        fingerTable(k) <- Finger (start, end, node);
    next <- 0;
    predecessor <- null;
    successorSuccessor <- fingerTable(1).node;
    heartBeatCounterPredecessor <- 0;
    heartBeatLimitPredecessor <- 3;
    heartBeatCounterSuccessor <- 0;
    heartBeatLimit Sucessor <- 3;
    joinComplete <- false;
    Setup Periodic Timer Stabilize(T);
    Setup Periodic Timer Fix_Fingers(T);
    Setup Periodic Timer Trigger_Heart_Beat_Successor(T);
    Setup Periodic Timer Trigger_Heart_Beat_Predecessor (T);

Procedure calculateStart(n, k):
    Return 2^k-1 mod 2^m;

Upon Receive(Create) do:
    joinCompleted <- true;
    predecessor <- null;
    start <- calculateStart (thisNode.id, 1);
    end <- calculateStart (thisNode.id, 2);
    fingerTable[1] <- finger (start, end, thisNode);

Upon Receive(Join, contact) do:
    predecessor <- null;
    **Trigger** Find_Successor (contact, thisNode);

**Upon** Receive(Find_successor, n) **do**:
    **If** n.id ∈ (thisNode.id, fingerTable[1].node.id] **then**
        **Trigger** Find_Successor_Response(n, fingerTable[1].node);
    **else**
        otherN <- Closest_Preceding_Node(n);
    **If** otherN.id == thisNode.id **Then**
        **Trigger** Find_Successor_Response(n, thisNode);
    **else**
        **Trigger** Find_Successor(otherN, n);

**Upon** Receive(Find_Successor_Response, successor) **do**:
    **If** successor.id != thisNode.id **Then**
        joinCompleted <- true;
        fingerTable[1].node <- successor
        successorSuccessor <- successor
        heartBeatCounterSuccessor <- 0
     **else**
        thisNode <- new Node(createID(), thisNode.ip, thisNode.port)
     **foreach** k ∈ m + 1
          start <- calculateStart(thisNode.id, k);
          end <-  calculateStart(thisNode.id, k+1);
          node <- thisNode;
          fingerTable[i] <- new Finger(start, end, node);
     next <- 0
     predecessor <- null
     successorSuccessor <- fingerTable(1).node;
     heartBeatCounterPredecessor <- 0
     heartBeatCounterSuccessor <- 0

**Upon** Receive(Fix_Fingers) **do**:
     **If** joinCompleted == true **Then**:
        next <- next + 1
        **If** next > m **Then**:
          next <- 1
        **Trigger** FF_Find_Successor(thisNode, fingerTable[next].intervalStart)

**Upon** Receive(FF_Find_Successor, n, start) **Do:**
     **If** start ∈ (thisNode.id, fingerTable[1].node.id] **Then**
        **Trigger** FF_Find_Successor_Response(n, fingerTable[1].node);
     **else**
        otherN <- FF_Closest_Preceding_Node(start)
        **If** otherN.id == thisNode.id **do**
            **Trigger** FF_Find_Successor_Response(n, thisNode);
        **else**
            **Trigger** FF_Find_Successor(otherN, n, start);

**Upon** Receive(FF_Find_Successor_Response, successor) **do**
    fingerTable[next].node <- successor

**Upon Periodic Timer** Receive(Stabilize) **do**:
    **If** joinCompleted == true **Then**
        **Trigger** Ask_For_Predecessor(fingerTable[1], thisNode);

**Upon** Receive(Ask_For_Predecessor, n) **Then**
    **If** predecessor == null **Then**
```

```
            predecessor <- n
            heartBeatCounterPredecessor <- 0
    else
            Trigger Predecessor_Response(n, predecessor)

Upon Receive(Predecessor_Response, predecessor) do:
     If predecessor.id ∈ (thisNode.id, fingerTable[1].node.id Then
            fingerTable[1].node <- predecessor
      Trigger Notify(fingerTable[1], thisNode);

Upon Receive (Notify, pretender) do:
    If predecessor == null || pretender.id ∈ (predecessor.id, thisNode.id do:
          predecessor <- pretender;
          heartBeatCounterPredecessor <- 0

Upon Periodic Timer Receive(Trigger_Heart_Beat_Predecessor)
    If joinCompleted == true
            If heartBeatCounterPredecessor > heartBeatLimitPredecessor Then
                   predecessor <- null;
                   heartBeatCounterPredecessor <- 0;
            else if predecessor != null Then
                   Trigger Heart_Beat_Predecessor(predecessor, thisNode);
                   heartBeatCounterPredecessor  <- heartBeatCounterPredecessor + 1;

Upon Receive(Heart_Beat_Predecessor, node) do:
        Trigger Receive(Heart_Beat_ACK_Predecessor, node);

Upon Receive(Heart_Beat_ACK_Predecessor) do:
        heartBeatCounterPredecessor <- 0;

Upon Periodic Timer Receive(Trigger_Heart_Beat_Successor) do:
       If joinCompleted == true Then
            If heartBeatCounterSuccessor > heartBeatLimitSuccessor Then
                  fingerTable[1].node <- successorSuccessor
                  heartBeatCounterSuccessor <- 0;
            else
                 Trigger Heart_Beat_Successor(fingerTable[1].node, thisNode)
                 heartBeatCounterSuccessor <- heartBeatCounterSuccessor + 1;

Upon Receive(Heart_Beat_Successor, node) do:
       Trigger Receive (Heart_Beat_ACK_Successor, fingerTable[1].node);

Upon Receive(Heart_Beat_ACK_Successor, successorOfSuccessor) do:
       successorSuccessor <- successorOfSuccessor;
       heartBeatCounterSuccessor <- 0;

Upon Receive(Send, msg, node) do:
       Trigger Receive(Ask_To_Deliver, thisNode, msg);

Upon Receive(Ask_To_Deliver, msg) do:
       Trigger Deliver(thisNode, msg);

Upon Receive(Route, msg, topicId) do:
        If topicId ∈ (thisNode.id, fingerTable[1].node.id) then
            Trigger Receive(Ask_To_Deliver, fingerTable[1].node, msg);
        else
            otherN <- FF_Closest_Preceding_Node(topicId);
        if otherN.id == thisNode.id then
            Trigger Receive(Ask_To_Deliver, thisNode, msg)
        else
            Trigger Receive(Ask_To_Forward, otherN, msg),

Upon Receive(Ask_To_Forward, msg) do:
        Trigger Forward(thisNode, msg);

Procedure Closest_Preceding_Node(node)
     Foreach i <- m to 1 do:
        If fingerTable[i].node.id ∈ (thisNode.id, node.id) Then
             return fingerTable[i].node
        return thisNode;

Procedure FF_Closest_Preceding_Node(start)
     Foreach i <- m to 1 do:
        If fingerTable[i].node.id ∈ (thisNode.id, start) Then
             return fingerTable[i].node
        return thisNode;

Procedure createID():
        return new random($2^m$) //nextInt(0,exclusive)
```

4.2. Publish-subscribe (scribe)

```
State:
   topics; //map with all the topics the known topics
   thisNode; // this node
   m; // constant, size of network = $2^m$
   subscriptionExpirationLimit;

Upon Init() do:
   topics <- {};
   thisNode <- new Node(-1, myHostname, myPort)
   m <- 8
   subscriptionExpirationLimit <- 5;

 Upon Receive(Forward, msg) do:
      if msg.msgType == "SUBSCRIBE" then
        if (msg.topic ∉ topics) then
          topics <- topics ∪ msg.topic;
          newMsg <- new Message(msg.msgType, thisNode, msg.topic, msg.content);
          Trigger Route(thisNode, newMsg, hash(msg.topic);
        topics(msg.topic).children <- topics(msg.topic).children ∪ msg.source;
      else
        Trigger Route(thisNode, msg, hash(msg.topic));


Upon Receive(Deliver, msg) do:
      msg.msgType match {
        case "CREATE":
          topics <- topics ∪ (msg.topic -> new Topic(msg.topic);
        case "SUBSCRIBE":
          if msg.topic ∈ topics then
```

```
                if msg.source.ip != thisNode.ip && msg.source.port != thisNode.port then
                    topics[msg.topic].children <- topics[msg.topic].children ∪ msg.source
            case "PUBLISH":
                if msg.topic ∈ topics then
                    topics[msg.topic].parent <- msg.source
                    newMsg <- new Message(msg.msgType, thisNode, msg.topic, msg.content)
                    foreach p ∈ topics[msg.topic].children
                        Trigger Send(newMsg, p);
                    if topics[msg.topic].subscribed == true) then
                        // Trigger the upper layer event
                    else
                        if topics[msg.topic].children.size == 0 then
                            if topics[msg.topic].parent != null then
                                newNewMsg <- new Message("UNSUBSCRIBE", thisNode, msg.topic, "UNSUBSCRIBE
FROM TOPIC: "+msg.topic)
                                Trigger Send(thisNode, newNewMsg, topics[msg.topic].parent)
                                topics <- topics \ msg.topic
            case "UNSUBSCRIBE":
                if msg.topic ∈ topics then
                    topics[msg.topic].children <- topics[msg.topic].children \ msg.source
                    if topics[msg.topic].children.size == 0 && topics[msg.topic].subscribed != true &&
topics[msg.topic].parent != null then
                        newMsg <- new Message(msg.msgType, thisNode, msg.topic, msg.content);
                        Trigger Send(thisNode, newMsg, topics[msg.topic].parent);
                        topics <- topics \ msg.topic;

Upon Receive(CreateTopic, topic) do:
        msg <- new Message("CREATE", thisNode, topic, "CREATE TOPIC: "+topic);
        Trigger Route(thisNode, msg, hash(topic);

Upon Receive(Subscribe, topic) do:
        if (msg.topic ∉ topics)
            topics <- topics ∪ new Topic(topic);
        topics[topic].subscribed <- true;
        topics[topic].renewalCounter <- 0;
        msg <- new Message("SUBSCRIBE", thisNode, topic, "SUBSCRIBE TO TOPIC: "+topic)
        Trigger Route(thisNode, msg, hash(topic));

Upon Receive(Publish, topic, content) do:
        msg <- new Message("PUBLISH", null, topic, content);
        Trigger Route(thisNode, msg, hash(topic));

Upon Receive(Unsubscribe, topic) do:
        if msg.topic ∈ topics
            topics[topic].subscribed <- false;
            if topics[topic].children.size == 0 && topics[topic].parent != null then
                newMsg <- new Message("UNSUBSCRIBE", thisNode, topic, "UNSUBSCRIBE FROM TOPIC:
"+topic)
                Trigger Send(thisNode, newMsg, topics(topic).parent);
                topics <- topics \ topic;

Upon Receive(Trigger_Subscriptions_Renewal) do:
        Foreach p in topics do:
            if p.subscribed == true then
                if p.renewalCounter > subscriptionExpirationLimit
                    msg = new Message("SUBSCRIBE", thisNode, topic, "SUBSCRIBE TO TOPIC: "+topic);
                    Trigger Route(thisNode, msg, hash(topic))
                            p.renewalCounter = 0 // Reset time
                else
                    p.renewalCounter = topic.renewalCounter + 1;
```

5. Experimental Evaluation

Our tester was based on launching 20 nodes, creating 50 random topics. Afterwards, each node subscribes to 5 random topics and publishes to other 5 random topics. In the end we count the number of messages sent and the number of messages received by the different nodes, as well as the percentage of delivered messages when compared with what we expected to receive. The tester was only built to work on a single machine, usually achieving 100% delivery rate. There was also some manual testing through the command line interpreter done, launching multiples nodes, some of them on different machines, and testing with the various pubsub operations. During these manual tests there the delivery rate also appears to be great, but we can't say for sure how well it scales when on different machines since there's a limit to manual testing and we couldn't get the tester in time to work with multiple machines.

6. Conclusions

With this project we were able to understand the logic behind the publish-subscribe protocol, Scribe, and the Distributed Hash Table, Chord, more in depth and know their true complexity. It was extremely hard to do this because of the lack of experience with the Akka framework and the Scala language, facing major setbacks when it came to remote communication. Regardless, we believe we were able to successfully complete the assignment.

7. References

[1] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. 2001. Chord: A scalable peer-to-peer lookup service for internet applications. In Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM '01). ACM, New York, NY, USA, 2001.
[2] Antony Rowstron1, Anne-Marie Kermarrec1, Miguel Castro, and Peter Druschel. Scribe: The Design of a Large-Scale Event Notification Infrastructure. Microsoft Research, Cambridge.