ASD Project Phase 2

Emanuele Vivoli*[1], Joao Miguel Costa†[2] and Pedro Guilherme Silva‡[2]

[1]Department of Informatics Engineering, University of Florence, IT
[2]Department of Informatics Engineering, NOVA FCT, PT

November 2019

Contents

*57284 - emanuele.vivoli@stud.unifi.it
†50171 - jmp.costa@campus.fct.unl.pt
‡50390 - pg.silva@campus.fct.unl.pt

i Introduction

This is the second phase report for the project "publish/subscribe system based on Topics" assigned in the class "Algorithm and Distributed Systems" 2019/ 2020 [Leitao, 2019].

A distributed system is based on the idea of decoupling (of space and time), where lots of different processes cooperate without sharing memory (everyone has its own memory not accessible from others) or timers (they are not synchronized with each other and they don't need to be online at the same time). In this pub/sub system the decoupling of both space and time is obtained between the message generators and the message consumers.

The first phase of the project focused on the decoupling of space, where each process has its own data structure to memorize the messages that flow in the network via broadcast. The work included the design and implementation of a Gossip Broadcast protocol with Recovery mechanisms and the implementation of an Unstructured Overlay Network [Leitao, 2019] (HyParView protocol [Leitao et al., 2007]).

This second phase aims to optimize the number of messages sent in the network by adding an alternative to the broadcast method. The alternative is composed from a dissemination layer implementing the "Scribe" protocol [Rowstron et al., 2002] and a lower layer that has the role to build and maintaining a Structured Overlay Network using the protocol defined by "Chord" [Stoica et al., 2001].

The project uses the Babel framework described in [Leitao et al., 2019].

ii Publish/Subscribe System

In topic-based publish/subscribe systems, each node can do three actions: **subscribe** and **unsubscribe** to a topic, and **publish** a message with one or more topics. Messages that flow in the network are delivered only from processes that have been subscribed to that or those topics. A process can unsubscribe a topic to inform the system that it is no more interested in receiving messages about that topic.

For this phase each node will store and manage its own subscriptions and the subscription that have to pass throw the node, in order to flow in the network. Each message can flows in the network in two different ways, using a Structured or an Unstructured overlay network.

The first was build the phase 1, the second one is the aim of this phase and it's composed of the Chord and Scribe protocols implementation. The idea to have a structured and an unstructured overlay network is to let the Public-Subcribe layer decides how to flows messages through the network, in according to the popularity of the messages in the network. The popularity information is stored in the node responsible for a specific topic, and the responsible is chosen by it's nearest ID to the Key of the topic, as the Chord protocol defines. This is the ideal mechanism, in this project we have built the Chord - Scribe - Pub/Sub interaction, letting for the third phase the real implementation of the popularity mechanism.

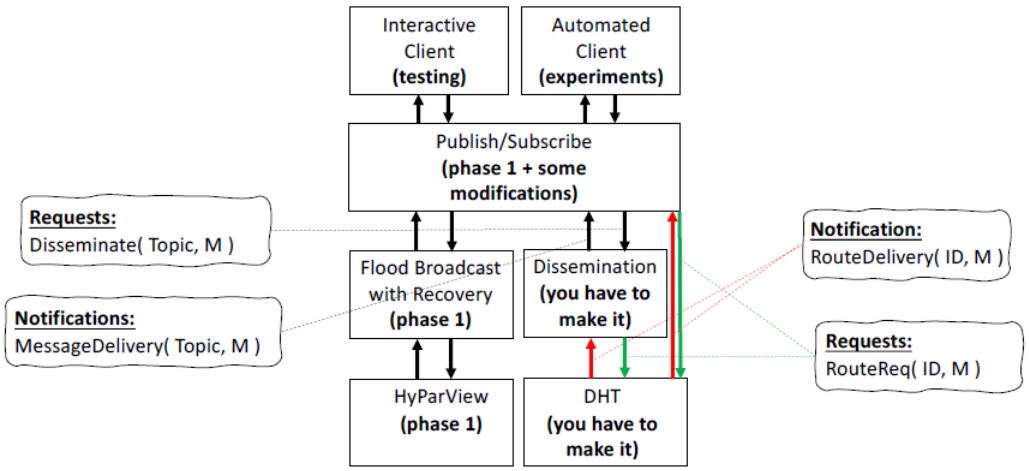

Figure 1: System architecture and description of inter layer messages

1 Unstructured Overlay Network

We described it in the first report.

1.1 HyParView

We described it in the first report.

1.2 Broadcast

We described it in the first report.

2 Structured Overlay Network

In this section we present our implementation of the Structured overlay network based on Chord [Stoica et al., 2001] and Scribe [Rowstron et al., 2002] protocols. A general scheme is presented in the Figure 1.

The first protocol is used to build and maintain a structured network, the second protocol use the first one in order to have information about the connected nodes, and to build a tree representation of the nodes interested on some topic, in a Publish-Subscribe environment.

2.1 Chord

This layer is responsible to build a network where nodes have a limited number of different neighbours in a manner that initially follows the circle pattern. The circle pattern is then updated, in the Chord protocol, adding $m - 1$ links where m is the number of bit used to represent the unique ID of the nodes, and the ID is obtain applying a HASH function ($SHA - 256$) to the "$ip : port$" string.

2.2 Messages flow

The messages, also in this case, can be defined as **Intra** and **Inter Layer**.

2.2.1 Intra-Layer messages

The messages that flows in Chord layer of different processes are:

- JOIN request, it's a message generated by a new node that will join the chord network; this message allows the contact node to spread the message along the chord network in order to find the successor of the new node.
- JOIN reply, it's an answer message containing the successor node for the new node.
- PREDECESSOR request, it's a periodic message generated by a node in order to obtain the predecessor of its successor.
- PREDECESSOR reply, it's an answer message generated by a node cointaining its predecessor (in the next phase will be add in this message the fingers table of itself).
- NOTIFY message, message generated by a node in order to inform the successor of its election of successor. This guarantees a symmetric relationship among successors and predecessors.
- ROUTE message, it's a message used to propagate information along the chord network, making one jump to the node with the closest identifier. Every time this message is sent, the chord deliver it to the scribe protocol that have the responsibility to decide how to route the message; either by Chord or Scribe level.

2.2.2 Inter-Layer messages

As shown in Figure 1 the Chord layer provide to the Publish/Subscribe and Scribe layer two functionalities that allow to "RouteRequest" and "RouteDelivery" a message. The interaction are used to route a message along the Chord level and notify the upper layers about the reception of a message. The notification can be also an announce that a message was forwording in the Chord layer. These two functionalities work with a simple pair of messages:

- ROUTE request, received from the upper layer when the upper layer wants to disseminate a message to the Chord level.
- ROUTE reply, is used to inform the upper layer protocol of two things:
  - When the upper layer ask the Chord to send a message, it notifies the layer above the destination of the message. This message is important for the Scribe layer to build a bidirectional tree.
  - When the node receives a Chord Level message it notifies the upper layer about the message. Is a role of the upper layer to decide how to handle the rest of the dissemination process.

2.2.3 Pseudocode

Interface :
  Requests :
    RouteRequest(m)
  Indications :
    RouteDeliverNotification(m)

upon init (*contact*) **do**
  $predecessor \leftarrow null$
  $successor \leftarrow myself$
  $contactNode \leftarrow contact$
  // referred to SHA-256
  $NUMBER\_OF\_BITS \leftarrow 256$

```
    next ← 1
    id ← HASH(ip : port)
    myself ← (id, myHost)
    fingers ← {}
    trigger JoinProtocolMessage(contactNode, JOIN_CODE)

upon event JoinProtocolMessage(JOIN_REQUEST, JOIN_CODE, sender) do
    trigger find_successor(id, m, JOIN_CODE);

procedure find_successor(id, m, JOIN_CODE) do
    if id ∈ (myself, successor] then
        Send(m.getNewNode(), JOIN_REPLY, JOIN_CODE, successor);
    else
        destination ← call closest_preeciding_node(id);
        Send(destination, JOIN_REQUEST, JOIN_CODE, m.getNewNode());
    end if

procedure closest_preeciding_node(id) do
    for i ← NUMBER_OF_BITS downto 1 do
        if fingers[i] ∈ (myself, id) then
            return fingers[i];
        end if
    end for
    return myself;

upon event JoinProtocolMessage(JOIN_REPLY, JOIN_CODE, sender) do
    successor ← sender;
    // this procedure change the fingers table until id < sender
    call change_fingers_table(sender);

upon timer stabilize do
    Send(successor, STABILIZE_REQUEST);

upon event stabilizeRequestMessage(STABILIZE_REQUEST, sender) do
    Send(sender, STABILIZE_REPLY, predecessor);

upon event stabilizeReplyMessage(STABILIZE_REPLY, sender, successor_predecessor) do
    x ← successor_predecessor;
    if x ∈ (myself, successor) then
```

```
        successor ← x;
    end if
    Send(successor, NOTIFY)

upon event notifyMessage(NOTIFY, sender) do
    if predecessor is null or sender ∈ (predecessor, myself) then
        predecessor ← sender;
    end if

upon timer fix_fingers do
    next ← next + 1;
    if next > NUMBER_OF_BITS then
        next ← 1;
    end if
    if next ∉ (myself, successor] then
        trigger find_successor(next, m, NEXT_CODE);
    end if

procedure find_successor(id, m, NEXT_CODE) do
    if id ∈ (myself, successor] then
        Send(m.getNewNode(), id, FIX_FINGER_REPLY, NEXT_CODE, successor);
    else
        destination ← call closest_preeciding_node(id);
        Send(destination, id, FIX_FINGER_REQUEST, NEXT_CODE, m.getNewNode());
    end if

upon event fixFingersMessage(FIX_FINGER_REQUEST, next, NEXT_CODE, sender) do
    trigger find_successor(next, m, NEXT_CODE);

upon event fixFingersMessage(FIX_FINGER_REPLY, next, NEXT_CODE, newNode) do
    fix_fingers[next] ← newNode;

upon event routeRequest(id, m) do
    if id ∈ (predecessor, myself] then
        trigger RouteDelivery(master : true, sendedTo : null, sendBy : null, message : m);
    else
        nextNode = closes_preceding_node(id);
        trigger SendRouteMessage(nextNode, id, m);
        trigger RouteDelivery(master : false, sendedTo : nextNode, sendBy : null, message : m);
    end if
```

upon event $routeMessage(sender, id, m)$ **do**
  if $id \in (predecessor, myself]$ **then**
    trigger $RouteDelivery(master : true, sendedTo : null, sendBy : sender, message : m)$;
  else
    trigger $RouteDelivery(master : false, sendedTo : null, sendBy : sender, message : m)$;
  end if

upon crash $(host)$ **do**
  if $predecessor = host$ **then**
    $predecessor \leftarrow null$;
  else
    if $successor = host$ **then**
      // if in the first table doesn't exists, failures_replace checks in the second table
      $new\_successor \leftarrow$ **call** $failures\_replace(host)$;
      $changeSuccessor(new\_successor)$
      $changeTable(new\_successor)$
    end if
  end if

2.3 Scribe

This layer is responsible to build a network tree, for each topic, where every nodes is connected (directly or indirectly) to all the nodes that are subscribed to that topic. This allows a faster message dissemination process along the subscribed nodes and reduces the number of messages flowing in the network.

2.4 Messages flow

The messages, also in this case, can be defined as **Intra** and **Inter Layer**.

2.4.1 Intra-Layer messages

The messages that flows in Scribe layer of different processes are:

- SCRIBE_MESSAGE, it's a message used to disseminate information along the topic tree at the Scribe Protocol level.

2.4.2 Inter-Layer messages

As shown in Figure 1 the Scribe layer provides to the Publish/Subscribe layer two functionalities called "DisseminateRequest" and "MessageDelivery".

The "DisseminateRequest" is used for Publish, Subscribe and Unsubscribe to a certain topic and this has a direct impact on the tree's structure (of that topic). The "MessageDelivery" are used to notify the PubSub protocol that a message belonging to an interested topic has been received, so the scribe has the responsibility to notify the upper layer about it. Note that the message is only forwarded to the upper protocol (Pub/Sub) if the node in question is subscribed to that topic; otherwise the notification wont be triggered.

The Pub/Sub protocol interacts with the Scribe protocol using the following messages:

- DISSEMINATE request, received from the upper layer when it wants to send a message. This message can be a publish, subscribe or unsubscribe message to a certain topic.

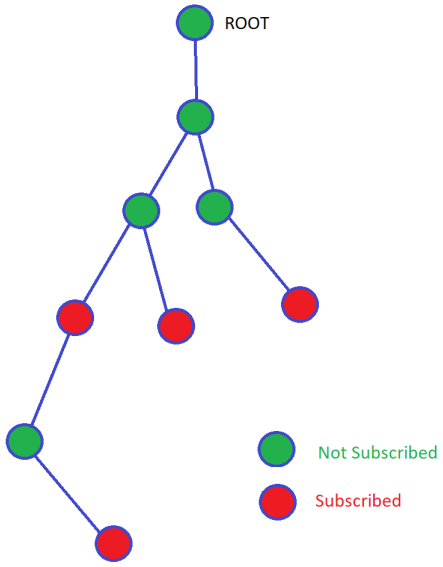

Figure 2: Scribe example of a topic tree with subscribed and unsubscribed nodes.

- DELIVER notification, is used to inform the upper layer protocol that a message (which it is interested on) has been received.

The Scribe protocol is strongly connect to Chord. Scribe and Chord work in a complementary way where the Scribe represents the brains of the dissemination process by making decision of how a message should be disseminates; and the Chord just send it to the closest preceding node (as specified in the protocol) reporting to the Scribe a notification of the action.

To interact with Chord the Scribe uses the interface described in 2.1.

2.4.3 Pseudocode

Interface :
Requests :
Disseminate(topic, m)
Indications :
MessageDelivery(topic, m)

upon init () **do**
$topicManagers \leftarrow \{\}$
$myself \leftarrow (id, myHost)$

upon event $DisseminateMessageRequest(SUB, topic, msg)$ **do**
if $topic \in\ topicManagers$ **then**
$manager \leftarrow topicManager.get(topic)$;
$manager.subcribe(topic, true)$;
else

```
        id ← Hash(topic)
        m ← Serialize(id, topic, msg)
        Send(ROUTE_REQUEST, SUB, m)
    end if

upon event DisseminateMessageRequest(UNSUB, topic, msg) do
    if topic ∈ topicManagers then
        manager ← topicManager.get(topic);
        manager.setSubscription(false);
        if manager.getChildrenSize() = 0 then
            topicManagers ← topicManagers/{topic}
            if !manager.isRoot(myself) then
                SendMessage(SCRIBE_UNSUB, msg)
            end if
        end if
    end if

upon event DisseminateMessageRequest(PUB, topic, msg) do
    if topic ∈ topicManagers then
        sendSet ← {}
        sendSet ← sendSet ∪ manager.getRoot();
        manager ← topicManager.get(topic);
        if !manager.isRoot(myself) then
            sendSet ← sendSet ∪ manager.getRoot()
        end if
        for Host h : sendSet do
            // serialize topic and msg in m before sending
            sendMessage(SCRIBE_PUB, topic, msg)
        end for
        if manager.amISubscribed() then
            trigger MessageDelivery(topic, message)
        end if
    else
        id ← Hash(topic)
        m ← Serialize(id, topic, msg)
        Send(ROUTE_REQUEST, SUB, m)
    end if

upon event RouteDelivery(PUB, isOwner, nextNode, sender, m) do
    (id, topic, msg) ← Deserialize(m)
    manager ← topicManager.get(topic);
    if isOwner then
        manager.setRoot(myself);
    else
        if topic ∈ topicManagers then
            sendSet ← {}
            sendSet ← sendSet ∪ manager.getRoot();
            manager ← topicManager.get(topic);
```

```
            if !manager.isRoot(myself) then
                sendSet ← sendSet ∪ manager.getRoot()
            end if
            sendSet ← sendSet/{sender}
            for Host h : sendSet do
                sendMessage(SCRIBE_PUB, m)
            end for
            if manager.amISubscribed() then
                trigger MessageDelivery(topic, message)
            end if
        else
            Send(ROUTE_REQUEST, SUB, m)
        end if
    end if

upon event RouteDelivery(SUB, isOwner, nextNode, sender, m) do
    (id, topic, msg) ← Deserialize(m)
    if isOwner then
        // it creates the tree if it doesn't exist
        topicManagers.addRoot(topic, nextNode);
    else
        if topic ∈ topicManagers then
            manager ← topicManager.get(topic);
            manager.addChildren(sender);
        else
            id ← Hash(topic)
            topicManagers.createTree(topic);
            manager ← topicManager.get(topic);
            if sender = null then
                manager.addRoot(nextNode);
            else
                manager.addChildren(sender);
            end if
            Send(ROUTE_REQUEST, m)
        end if
    end if

upon event ScribeMessage(UNSUB, sender, id, topic, msg) do
    if topic ∈ topicManagers then
        manager ← topicManager.get(topic);
        manager.removeChildren(sender);
        if manager.getChildrenSize() = 0 and not(manager.amISubscribed) then
            topicManagers ← topicManagers/{topic}
            if !manager.isRoot(myself) then
                Send(SCRIBE_MESSAGE, UNSUB, id, topic, msg)
            end if
        end if
    end if
```

```
upon event ScribeMessage(PUB, sender, id, topic, msg) do
    if topic ∈ topicManagers then
        sendSet ← {}
        sendSet ← sendSet ∪ manager.getRoot();
        manager ← topicManager.get(topic);
        if !manager.isRoot(myself) then
            sendSet ← sendSet ∪ manager.getRoot()
        end if
        sendSet ← sendSet/{sender}
        for Host h : sendSet do
            sendMessage(SCRIBE_PUB, m)
        end for
        if manager.amISubscribed() then
            trigger MessageDelivery(topic, msg)
        end if
    end if
```

3 Publish/Subscribe

The Publish/Subscribe module was used as a simple filter layer between the client (that want to receive only the messages that belongs to topic it is interested to) and the Broadcast layer that receive and forward to the upper layer every message it delivered (it receive every message 'broadcasted', but deliver each message just once). In the second phase it stopped to apply the filter and to memorize the subscription topics, since it is the work of the Scribe layer. In the next phase will be added to this layer the managing of topic popularity information.

3.1 Messages flow

We described it in the first report.

3.1.1 Inter-Layer messages

In the inter-layer messages nothing changed with respect to the previous phase, but now we have built two new handler called "RouteRequest" and "RouteDelivery" provided by the Chord protocol. Those will be used for asking the chord to propagate a message until it reaches the topic owner that has to decide the message dissemination method (between 'broadcast' and 'scribe'). The message dissemination method will be send back from the topic owner to the sender node in the Publish/Subscribe layer, building the first Intra-Layer message.

4 Client

We described it in the first report.

4.0.1 Manual Client Commands

We described it in the first report.

5 Experimental Results

The following experiences were aimed at testing Chord and Scribe Protocols and then comparing these with the ones developed in the first phase. In order to evaluate the correctness of the protocols we conducted some tests using the developed Automated Client, which runs random commands and can be configured to

publish a specific number of messages. The experiments were mainly aimed at measuring the following stats: reliability without failures; total number of messages received in the Scribe Layer by all nodes; convergence time (time it takes all the messages to reach the expected nodes); total number of Chord messages sent by all the nodes; load balancing in Chord layer.

5.1 Experimental Setting

The experiments were conducted using the same protocol configurations. The only specific parameters were on the Chord layer, regarding the Stabilize and Fix Fingers rate.

5.2 Reliability without failures

The first set of experiments were aimed at testing the protocols on a environment free from failures to check the correctness of the protocols. In all the following three tests the measured Reliability was almost 100% every time- all the nodes in the network received all the desired messages.

5.2.1 Total messages received on Scribe layer

This test was conducted in a network of 20 nodes with each node generating 10,25,50 or 75 publish messages. Figure 3 shows the relation between the total number of messages received in the Scribe layer and the number of publish messages generated by each node.

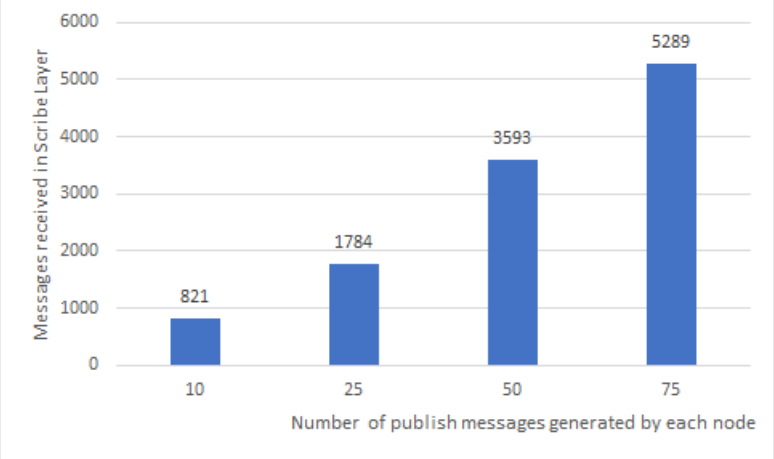

Figure 3: Total number of messages received in the Scribe Layer by all nodes

5.2.2 Convergence time

The goal of this test was to measure how much time it takes for all the messages to reach every node. It was conducted in a network of 20 nodes with each node generating 10, 25, 50 or 75 publish messages.Figure 4 shows the relation between the convergence time and the number of messages generated by each node.

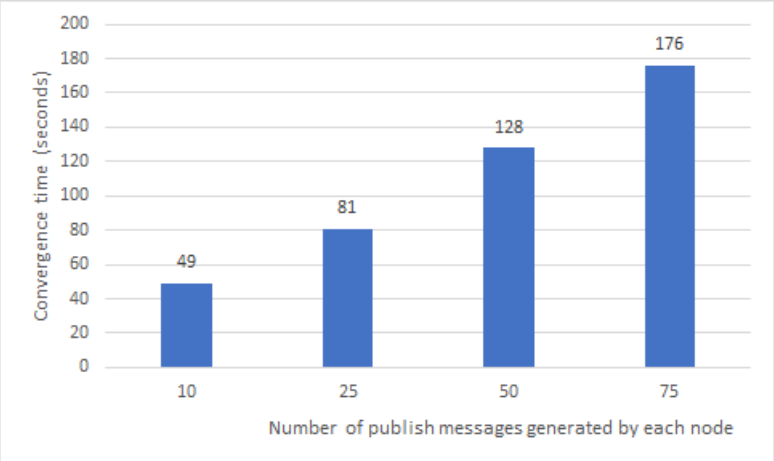

Figure 4: Convergence time

5.2.3 Total Chord messages sent

The goal of this test was to show the relation between the number of nodes in the network and the total number of Chord messages sent by all nodes. It was conducted in networks of 10/20/30 nodes with each node generating 20 publish messages. Figure 5 shows the result graph of this test.

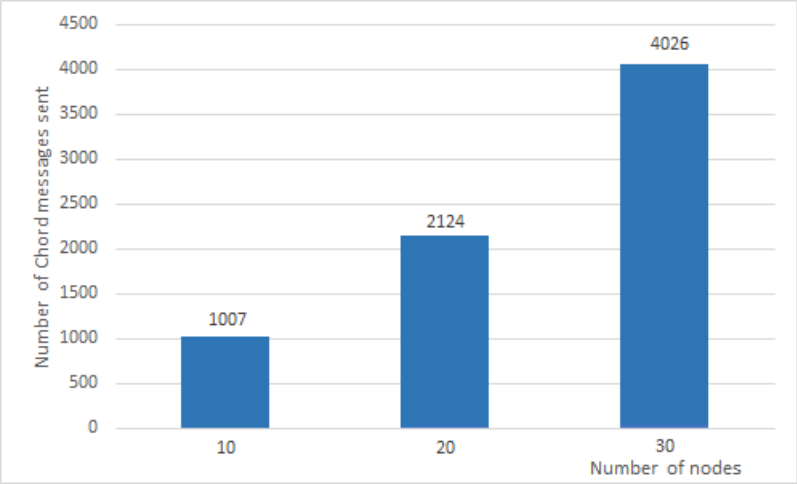

Figure 5: Total number of Chord messages sent by all nodes

5.2.4 Load Balancing on Chord Layer

The goal of this test was to check if the topics were being evenly distributed through all the nodes in the system: every node should own the same number of topics. It was conducted in networks of 10/20/30 nodes with 30 topics. Figure 6 shows the relation between the number of nodes in the network and the distribution of topics. For every network tested (10,20 or 30 nodes) it is possible to check the expected number of topics per node (totalTopics/totalNodes); the real value of the minimum topics per node (there are nodes that do not own a topic); and the real value of the maximum number of topics per node (there are nodes that own mores topics then expected). The main conclusion here is that by increasing the number of nodes in the network, the load of the system is more balanced for a fixed number of topics.

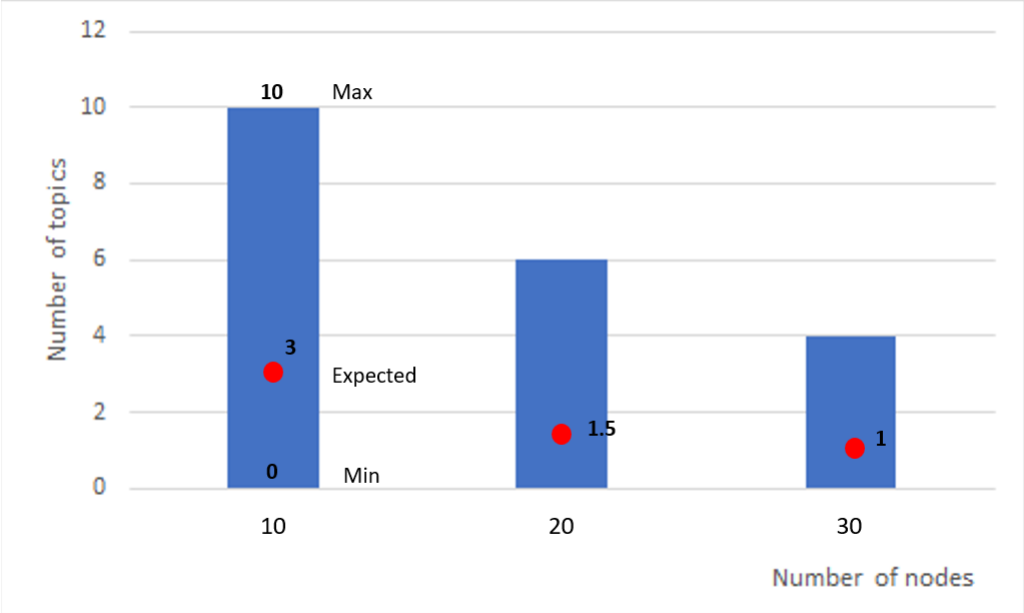

Figure 6: Load Balancing on the Chord Layer

5.3 Reliability with failures

The last set of experiments were aimed at testing the impact of failures in the system behaviour. The main goals were to measure the reliability of the protocols with failures and to check the correctness of the fault Recovery process (on Scribe and Chord). Given the characteristics of the system, the tests were only conducted by inducing small levels of failures but, as unexpected, even if the Chord had the capability to recovery a stable structure, the Scribe had problem with the Tree managing, making these tests impossible to evaluate. The Chord layer was also tested apart, and the regeneration of the table fingers and of the ring structure were positively mantained. In the other hand, we will do a real comparison between this phase and the previous phase under failure later, in a more realistic way.

5.4 Discussion and Comparison with the First Phase

The results of the experiments showed good overall measures of the developed solution. One of the main goals of these tests was to check the level of reliability of the solution and we obtained good results (every node received all the published messages by other nodes).

The tests were conducted in small networks (mainly constituted of 20 nodes) and with small published messages due to hardware restrictions. However, we think that the reliability of our solution in networks with more nodes and bigger messages would be equally good.

Regarding the results obtained in this phase compared with the previous phase, the main observation is that the number of messages circulating in the network is much less, although the convergence time is more or less equal; the load in the system is more balanced due to the DHT.

6 Conclusion

The goals for this second phase of the project were achieved: design and implementation of a set of protocols to improve the initial system; design and implementation of a Dissemination protocol (inspired on Scribe); implementation of a DHT protocol (inspired on Chord). The results obtained from the experimental tests showed a good measure of correctness of the implemented system.

References

[Leitao, 2019] Leitao, J. (2019). Algorithms and distributed systems. *MIEI - Integrated Master in Computer Science and Informatics, University Lectures*.

[Leitao et al., 2019] Leitao, J., Fouto, P., and Costa, P. A. (2019). Babel: Internal framework developed at nova lincs. *http://asc.di.fct.unl.pt/ jleitao/babel/docs/* , October - 2019.

[Leitao et al., 2007] Leitao, J., Pereira, J., and Rodriguess, L. (June 2007). Hyparview: A membership protocol for reliable gossip-based broadcast. *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, pages 419–429.

[Rowstron et al., 2002] Rowstron, A., Kermarrec, A.-M., Castro1, M., and Druschel, P. (2002). Scribe: The design of a large-scale event notification infrastructure. *IEEE*.

[Stoica et al., 2001] Stoica, I., Morris, R., Liben-Nowell, D., Karger, D. R., Kaashoek, M. F., Dabek, F., and Balakrishnan, H. (2001). Chord: A scalable peer-to-peer lookup protocol for internet applications. *Defense Advanced Research Projects Agency (DARPA) and the Space and Naval Warfare Systems Center, San Diego*.