ASD Project Phase 3

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

December 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 third 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]).

The second phase focused on optimizing 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].

This third and final phase aims to improve the system by adding a replication mechanism. The goal of the replication here is to have additional spare nodes: each node that actively belongs to the system runs two additional replicas of the Pub/Sub system, that replicate the state through Multi-Paxos. To accommodate this new mechanism, the Publish Subscribe Layer will evolve to become the state machine being replicated. Finally, some corrections will be made in the Scribe and Chord protocols regarding the implementation in the second phase.

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 to processes that are 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.

Each node stores and manages its own subscriptions and messages that have to pass through it in order to flow in the network. The Pub/Sub System can disseminate the messages in two different ways: using an Unstructured Overlay (via the Gossip Broadcast Protocol and the HyParView protocol) or a Structured Overlay Network (via the Scribe Protocol and the Chord Protocol).

The system has a replication mechanism, in which every active node has two replicas of the Pub/Sub system that replicate the state through Multi-Paxos.

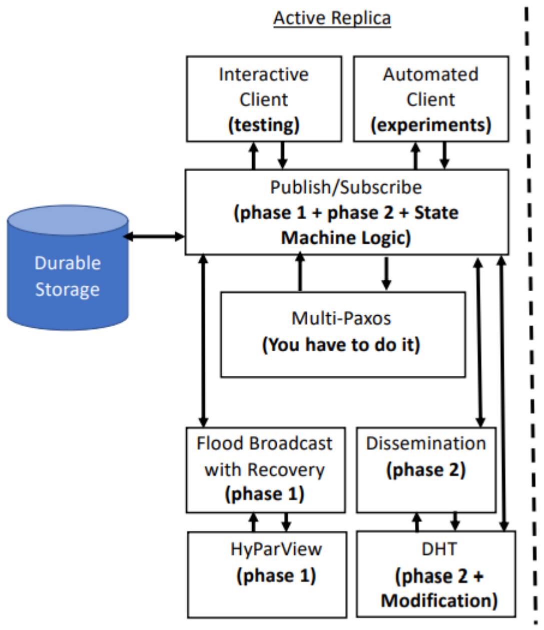
Active Replica
Backup Replicas
Interactive Client (testing)
Automated Client (experiments)
Publish/Subscribe (phase 1 + phase 2 + State Machine Logic)
Durable Storage
Multi-Paxos (You have to do it)
Flood Broadcast with Recovery (phase 1)
Dissemination (phase 2)
HyParView (phase 1)
DHT (phase 2 + Modification)
Interactive Client (testing)
Automated Client (experiments)
Publish/Subscribe (phase 1 + phase 2 + State Machine Logic)
Durable Storage
Multi-Paxos (You have to do it)

Figure 1: System architecture - layers and protocols

1 Architecture

In the system, every active node has two backup replicas. The active node is the node which is effectively connected to the network, while the backup replicas are the spare nodes used for replication. The architecture of the active node is composed by five distinct layers, as shown in Figure 1:

- **Client layer** - Interactive Client or Automated Client;
- **Publish/Subscribe layer** (with State-Machine logic);
- **Multi-Paxos layer**;
- **Dissemination layer** - Gossip Broadcast with Recovery or Scribe;
- **Overlay layer** - HyParView or Chord.

The backup replicas are composed of three layers: Client layer, Pub/Sub layer with Machine-State logic and Multi-Paxos layer.

For this third phase we continued to focus on improving the Scribe and Chord protocols. For the **subscribe** and **unsubscribe** operations, the flow of the execution remains the same as in the second phase, with some improvements and corrections. However, for the **publish** operation, with the addition of the replication mechanism, the execution is different. A client that wishes to publish a message to a topic, first sends a request through Chord to discover the topic owner. When the topic owner receives the notification from Chord that a node is trying to publish a message, the topic owner sends a direct message to original client containing a list with its address and the addresses of its backup replicas. When the original client receives this message, he sends a new message with the content he wants to publish to one of the nodes in the list. The message is ordered though **Paxos** and then a request is sent to Scribe to disseminate this message through the Multicast Tree.

This is a simple explanation of the execution flow. A complete description of the protocols involved in this phase is available in the next chapters, following a bottom-up approach.

2 Unstructured Overlay Network

The full description can be found on the first phase report.

2.1 HyParView

The full description can be found on the first phase report.

2.2 Broadcast

The full description can be found on the first phase report.

3 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 uses the first 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.

Phase 3 Improvements. In this final phase some improvements were made to these protocols, mainly fixes to the implementations of the phase 2 and to accommodate the new logic in the Pub Sub Protocol and the addition of Paxos. All changes will be addressed in each protocol below.

3.1 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, adding $m-1$ links where m is the number of bits used to represent the unique ID of the nodes, and the ID is obtained applying a HASH function ($SHA-256$) to the "$ip:port$" string.

In this final phase, new messages and requests were added to allow the execution of the new interaction in the publishing of a message to a topic: a publisher needs to first find the owner of some topic and then send it directly to this owner; the owner then decides what to do with the message.

3.2 Messages flow

The messages can be defined as **Intra** and **Inter Layer**.

3.2.1 Intra-Layer messages

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

- **JOIN Request**. 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**. Reply message containing the successor node for the new node.
- **PREDECESSOR Request**. Periodic message generated by a node in order to obtain the predecessor of its successor.
- **PREDECESSOR Reply**. Reply message generated by a node containing its predecessor.
- **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**. Message used to propagate information along the chord network, making one jump to the node with the closest identifier. This message is mainly used for routing purposes, when there is a request from Scribe (via Sub Operation). When this message is received, Chord delivers it to Scribe to manage the continuation of the routing.
- **ROOT message**. Message used to find the owner of some key (topic). This message is different from the Route Message because its being routed only through Chord to the key owner (no interaction with Scribe). When this message arrives in the owner of the key, a notification will be issued to PubSub. This message was added in this final phase to accommodate the new logic in the PubSub Protocol (Pusblish operation), where a message is sent directly to the topic owner, and the owner disseminates the message.

3.2.2 Inter-Layer messages

The interface of the Chord layer is composed by two Requests and two Indications: "Route Request", "Route Delivery Notification", "Find Root Request" and "Find Root Notification".

The requests are used in two different situations:

- **ROUTE Request**. Received from the upper layer when the upper layer wants to route the message through Chord level. In practice this request is only issued by Scribe in the Sub operation, when Scribe doesn't have information about the topic being Subscribed.
- **FIND ROOT Request**. Request received from the upper layer when the upper layer wants to know the owner of some key (topic). In practice this request is only issued by PubSub, in the Publish operation, when it doesn't know the owner of the topic (the message has to be sent to the owner).

Chord issues two different notifications:

- **ROUTE Delivery Notification**. Used to inform the upper layer protocol of two different things:
  - When the upper layer asks the Chord to route 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. Its the responsibility of the upper layer to decide how to handle the rest of the routing process.

  In practice, the upper layer is always Scribe.
- **FIND ROOT Notification**. This notification is issued to inform the upper-layer that its the owner of some key (topic). This also informs the upper layer that some other node is trying to find the owner of said topic. This notification is useful for the Publish operation in the PubSub protocol.

3.2.3 Pseudocode

```
Interface :
  Requests :
    RouteRequest(m)
    FindRootRequest(m)
  Indications :
    RouteDeliverNotification(m)
    FindRootNotification(m)


upon init (contact) do
  $predecessor \leftarrow null$
  $successor \leftarrow myself$
  $contactNode \leftarrow contact$
```

```
    // referred to SHA-256
    $NUMBER\_OF\_BITS \leftarrow 256$
    $next \leftarrow 1$
    $id \leftarrow HASH(ip : port)$
    $myself \leftarrow (id, myHost)$
    $fingers \leftarrow \{\}$
    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 $id \in\ (myself, successor]$ then
        $Send(m.getNewNode(), JOIN\_REPLY, JOIN\_CODE, successor)$;
    else
        $destination \leftarrow$ call $closest\_preeciding\_node(id)$;
        $Send(destination, JOIN\_REQUEST, JOIN\_CODE, m.getNewNode())$;
    end if


procedure $closest\_preeciding\_node(id)$ do
    for $i \leftarrow NUMBER\_OF\_BITS$ downto 1 do
        if $fingers[i] \in (myself, id)$ then
            return $fingers[i]$;
        end if
    end for
    return $myself$;


upon event $JoinProtocolMessage(JOIN\_REPLY, JOIN\_CODE, sender)$ do
    $successor \leftarrow sender$;
    // 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
```

```
    $x \leftarrow successor\_predecessor$;
    **if** $x \in (myself, successor)$ **then**
        $successor \leftarrow x$;
    **end if**
    $Send(successor, NOTIFY)$
```

```
**upon event** $notifyMessage(NOTIFY, sender)$ **do**
    **if** $predecessor$ **is null or** $sender \in (predecessor, myself)$ **then**
        $predecessor \leftarrow sender$;
    **end if**
```

```
**upon timer** $fix\_fingers$ **do**
    $next \leftarrow next + 1$;
    **if** $next > NUMBER\_OF\_BITS$ **then**
        $next \leftarrow 1$;
    **end if**
    **if** $next \notin (myself, successor]$ **then**
        **trigger** $find\_successor(next, m, NEXT\_CODE)$;
    **end if**
```

```
**procedure** $find\_successor(id, m, NEXT\_CODE)$ **do**
    **if** $id \in (myself, successor]$ **then**
        $Send(m.getNewNode(), id, FIX\_FINGER\_REPLY, NEXT\_CODE, successor)$;
    **else**
        $destination \leftarrow$ **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**
    $fix\_fingers[next] \leftarrow newNode$;
```

```
**upon event** $routeRequest(id, m)$ **do**
    **if** $id \in (predecessor, myself]$ **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
```

```
procedure $find\_root(keyId, originalSender)$ do
    if $keyId \in (predecessor, myself]$ then
        trigger $FindRootNotification(keyId, originalSender)$;
    else
        $receiver = closes\_preceding\_node(keyId)$;
        trigger $FindRootMessage(receiver, keyId, originalSender)$;
    end if
```

```
upon event $findRootRequest(keyId, originalSender)$ do
    trigger $find\_root(keyId, originalSender)$;
```

```
upon event $findRootMessage(keyId, originalSender)$ do
    trigger $find\_root(keyId, originalSender)$;
```

```
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
```

3.3 Scribe

This layer is responsible to build a multicast 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.

In this final phase we fixed some errors related with the implementation of the sub/unsub operations in phase 2. We also updated the publish operation on this protocol to accommodate the updates on the overall publish interaction.

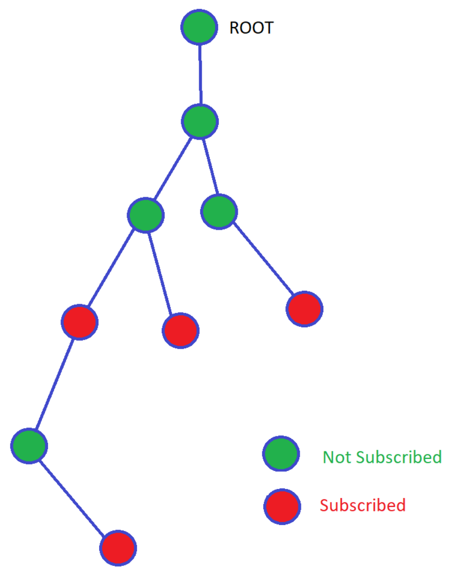

Figure 2: Scribe example of a topic tree with subscribed and unsubscribed nodes.

3.4 Messages flow

The messages can be divided in **Intra** and **Inter Layer**.

3.4.1 Intra-Layer messages

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

- **SCRIBE_MESSAGE**. Message used to disseminate information along the topic tree at the Scribe Protocol level.

3.4.2 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. The Subscribe and Unsubscribe operations have a direct impact on the tree's structure (of that topic). The "MessageDelivery" is used to notify the PubSub protocol that a message belonging to an interested topic has been received. The Scribe protocol has the responsibility of notifying 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 Scribe protocol interacts with the PubSub protocol using the following messages:

- **DISSEMINATE Request**. Received from the upper layer when it wants to execute an operation. This operation can be a publish, subscribe or unsubscribe message to a certain topic.

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

The Scribe protocol also interacts with Chord, on the Subscribe Operation, in **ROUTE Request** and **ROUTE Deliver**. When Scribe receives a Subscribe request from PubSub, there are two situations: if Scribe knows the topic, just subscribes to it; if not, Scribe makes a request to Chord (ROUTE Request) to route the request through the closest preceding node to the topic owner. This effectively builds the Multicast Tree. The interaction is briefly described in 3.1.

3.4.3 Pseudocode

```
Interface :
    Requests :
        Disseminate(OPERATION, 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
        $manager \leftarrow topicManagers.get(topic);$
        $manager.subcribe(topic, true);$
    else
        $newManager \leftarrow Manager(topic)$
        $topicManagers.put(topic, newManager)$
        $id \leftarrow Hash(topic)$
        $m \leftarrow Serialize(id, topic, msg)$
        $Send(ROUTE\_REQUEST, SUB, m)$
    end if

upon event $DisseminateMessageRequest(UNSUB, topic, msg)$ do
    if $topic \in\ topicManagers$ then
        $manager \leftarrow topicManager.get(topic);$
        $manager.setSubscription(false);$
        if $manager.getChildrenSize() = 0$ then
            $topicManagers \leftarrow topicManagers/\{topic\}$
            if $!manager.isRoot(myself)$ then
                $SendMessage(SCRIBE\_UNSUB, msg)$
            end if
        end if
    end if

upon event $DisseminateMessageRequest(PUB, topic, msg)$ do
    if $topic \in\ topicManagers$ then
        $sendSet \leftarrow \{\}$
        $sendSet \leftarrow sendSet \cup manager.getRoot();$
        $manager \leftarrow topicManager.get(topic);$
        if $!manager.isRoot(myself)$ then
            $sendSet \leftarrow sendSet \cup manager.getRoot()$
        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
    end if
```

```
upon event $RouteDelivery(SUB, isOwner, nextNode, sender, m)$ do
    $(id, topic, msg) \leftarrow Deserialize(m)$
    if $isOwner$ then
        // it creates the tree if it doesn't exist
        $topicManagers.addRoot(topic, nextNode);$
    else
        if $topic \in\ topicManagers$ then
            $manager \leftarrow topicManager.get(topic);$
            $manager.addChildren(sender);$
        else
            $id \leftarrow Hash(topic)$
            $topicManagers.createTree(topic);$
            $manager \leftarrow topicManager.get(topic);$
            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 $topic \in\ topicManagers$ then
        $manager \leftarrow topicManager.get(topic);$
        $manager.removeChildren(sender);$
        if $manager.getChildrenSize() = 0$ and not($manager.amISubscribed$) then
            $topicManagers \leftarrow topicManagers / \{topic\}$
            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 $topic \in\ topicManagers$ then
        $sendSet \leftarrow \{\}$
        $sendSet \leftarrow sendSet \cup manager.getRoot();$
        $manager \leftarrow topicManager.get(topic)$;
        if $!manager.isRoot(myself)$ then
            $sendSet \leftarrow sendSet \cup manager.getRoot()$
        end if
        $sendSet \leftarrow sendSet / \{sender\}$
        for $Host\ h : sendSet$ do
            $sendMessage(\text{SCRIBE\_PUB}, m)$
        end for
        if $manager.amISubscribed()$ then
            trigger $MessageDelivery(topic, msg)$
        end if
    end if
```

4 Multi-Paxos

In this section we describe the Multi-Paxos layer that was to be implemented. This layer description corresponds to the Ideal Multi-Paxos implementation, that is quite different to our real implementation. We will present these differences in section 4.2, where we will justify our actual implementation.

4.1 Messages flow

The messages can be divided in **Intra** and **Inter Layer**.

4.1.1 Intra-Layer messages

The messages in the same layer of the Multi-Paxos can be of three types, and are aplicable to the messages and theirs Ok replies. The first two message are called **PREPARE** and are used in the phase that described in Multi-Paxos papers as the Leader Election phase.

- **PREPARE message**, it is a message from a Multi-Paxos that is trying to be a new Leader. It is sent in more than one case, but generally it has the Sequence Number of the Propose and the Paxos Instance of the proposer Replica.
- **PREPARE OK message**, it is the message response to the message above, directed only to the sender. It means that the sender node has changed his Leader to the PREPARE message sender. It can contain the Paxos Instance of the sender, and the sequence number of the new Leader. It is also possible to append to this message a list of *byte*[ ]*array* with the last operations choosen. It can help in some situation where the Replica that is trying to be the new Leader, has the Paxos Instance that is lower than Paxos Instance of receiving Replicas (we will discuss later on when it can happen).

Once the leader is chosen, it can emit an **ACCEPT** message of an Operation that comes from the upper layer Publish/Subscribe. This Operation is intuitively a serialization of a AddNode, RemoveNode or TopicMessage. The important thing here is that the Multi-Paxos doesn't have to know any information about the type of the Operation, it just accepts an Operation in one Paxos Instance.

- **ACCEPT message**, this is a message that transports information about an Operation. It is generated after the Request from the Public/Subscribe layer. It contains the Paxos Instance, the Sequence Number and the *byte*[ ] *array* of the Operation that it transports.
- **ACCEPT OK message**, this is the answer message of the message above. It means that the destination have received the previous ACCEPT message. This message is directed to all the replicas within the ReplicaSet from every replica, it means that every one has to check if the majority is reached, and in case, deliver the chosen Operation to the upper layer.

This message describe bellow is generated periodically by using a timer. The timer happens each 1000ms for the Leader and each 2000ms for the Replicas. When the timer is invoked it makes different things if the node is the Leader or if it a Replica. If the node is a Leader than it generates a message.

- **NOOP message**, is generated from the Leader when it see that a large amount of time has passed without doing a propose operation. So in order to present the Replica from assuming it crash, the Leader send a No Operation message with a timer that is refresh by every Accept demanded from the upper layer.

It can be useful to add also a **NOOPOK message** to notify the Leader that a replica is still alive. In a system where the fault model is not crash-fault model the *OK messages* Replica senders can be saved in a set (like receivedOkSet), and in every *NOOP* timer activation we can check which replica (that is in the replicaSet) is not in the receivedOkSet. What we can also do, is to use the **NodeDown** primitive function that use this mechanism intrinsically because of the TCP connection between Replicas and Leader. With this hook we can detect a Replica failure.

4.1.2 Inter-Layer messages

The subsection describes the messages (Request and Notification) that can be sent from the Multi-Paxos layer to the Publish/Subscribe upper layer, and vice-versa. The main request are:

- **START request**, it is expressed in the task document that it has to be called to initiate the Multi-Paxos and let it knows if it is a leader or not, and provide him the replica set. (See how the method PubSubStart works to understand where the Pub/Sub takes this element).
- **PROPOSE request**, it is the base request that incorporate itself a message. This request can incorporate messages of AddReplica, RemoveReplica or TopicMessagePublication. This messages, depending on the implementation, can be *byte* [ ] *array* or just messages. In every case the Multi-Paxos doesn't have to look at it.

These two requests from the Public/Subscribe layer to the Multi-Paxos are done only when an Operation (serialized from the upper layer) has been approved from the paxos layer.

- **ADDREPLICA request**, is a request that comes from the upper layer, with the objective of adding replica to the ReplicaSet. This operation is done in loco, and this request is done only after having approved by the Multi-Paxos an Operation of adding a replica (and every existing replica has delivered this operation to the Pub/Sub).
- **REMOVEREPLICA request**, is a request from the upper layer that is asking to remove a replica. As before, at least the majority of all nodes have already approved the Remove operation to the Pub/Sub, without knowing it was a remove replica operation.

When one of the above request arrives, the Multi-Paxos do the necessary steps, and inform with a notification the upper layer that the operation was concluded. The notification mensioned can be seen below:

- **ADDED-REPLICA notification**, is a reply generated after having added a new replica to the replica set. It is generated as answer to the *ADDREPLICA request*.
- **REMOVED-REPLICA notification**, is a reply generated after having removed a replica from the replica set. It is generated as answer to the *REMOVEREPLICA request*.

In addition to this couple of Request-Notification set, we have some other notifications that not demand any following Reply-Notification linked with the upper layer. Those notification are:

- **LEADER-CHANGED notification**, that notify the upper layer that the Leader changed, it is not an operation that have to be decided, so it doesn't need to contain the Paxos Instance in it.
- **POSSIBLE-REMOVE-REPLICA notification**, it is a notification that comes from the Leader when he see that a node fails. It can detect it in various way, the simplest one is to use $NodeDown$.
- **DECIDED notification**, this is the notification that informs that last operation proposed from the Public/Subscribe has been decided. This operation can be *ADDNODE*, *REMOVENODE* or just a *TOPICMESSAGE*.

4.1.3 Pseudocode

Interface :

Requests :

StartProtocolRequest(paxosInstance, leader, replicaSet)
ProposeRequest(msg)
AddReplicaRequest(newHost)
RemoveReplicaRequest(toRemoveHost)

Indications :

DecidedNotification(N, msg)
LeaderChangedNotification([N], newLeader)
AddedReplicaNotification(N, newReplica)
RemovedReplicaNotification(N, removedReplica)
RemoveReplicaProposeNotification(toRemoveReplica)

Timers :

NoOpTimer()
AcceptTimer()
PrepareTimer()

```
upon init () do
    $myself \leftarrow (id, myHost)$
    $replicaSet \leftarrow \{\}$
    $proposerSeqNumberCounter \leftarrow 0$
    $proposerSeqNumber \leftarrow Hash(myself)$
    $paxosInstance \leftarrow 0$
    $higherSequenceNumberByPaxosRound \leftarrow newHashMap < int, int >$
```

```
upon event $StartProtocolRequest()$ do
    $leader \leftarrow myself$
    $replicaSet \leftarrow \{myself\}$
    $paxosInstance \leftarrow 0$
    $noOpTimer \leftarrow start(NO\_OP\_DELAY\_LEADER)$
```

```
upon event $StartProtocolRequest(paxosInstance, leader, replicaSet)$ do
    $leader \leftarrow leader$
    $replicaSet \leftarrow \{replicaSet\}$
    $paxosInstance \leftarrow paxosInstance$
    $noOpTimer \leftarrow start(NO\_OP\_DELAY\_REPLICA)$
    $higherSequenceNumberByPaxosRound \leftarrow < 0, null >$
```

```
upon event $ProposeRequest(msg)$ do
    $message \leftarrow serialize(msg)$
    $higherSequenceNumberByPaxosRound \leftarrow < paxosInstance, proposerSeqNumber >$
    $AcceptProtocolMessage\ paxosAccept \leftarrow (paxosInstance, proposerSeqNumber, message)$
    $lastPaxosAccept \leftarrow paxosAccept$
    $Accept()$
    $cancel(noOpTimer)$
    $AcceptTimer \leftarrow start(ACCEPT\_DELAY)$
```

```
procedure $Accept()$ do
    $cancel(AcceptTimer)$
    for $Host\ replica : replicaSet$ do
        // I should already have the LastPaxosAccept
        $sendMessage(LastPaxosAccept, replica)$
    end for
    $proposerHigherSeqNumberCounter = 0$
    $AcceptTimer \leftarrow start(ACCEPT\_DELAY)$
    $checkMajorityAcceptOk()$
```

```
procedure $checkMajorityAcceptOk()$ do
    if $proposerHigherSeqNumberCounter = majorityNeeded$ then
        if $leader = myself$ then
            $cancel(AcceptTimer)$
            $timeToWait \leftarrow NO\_OP\_DELAY\_LEADER$
        else
            $timeToWait \leftarrow NO\_OP\_DELAY\_REPLICA$
        end if
        $DecidedNotificationdecideNotification \leftarrow (paxosInstance, lastPaxosAccept)$
        $proposerHigherSeqNumberCounter \leftarrow 0$
        $this.paxosInstanceNumber + +$
        $higherSequenceNumberByPaxosRound \leftarrow < paxosInstanceNumber, null >$
        $noOpTimerId \leftarrow start(timeToWait)$
    end if
```

```
upon event $AddReplicaRequest(newHost)$ do
    $replicaSet \leftarrow replicaSet \cup \{newHost\}$
    // it just check the replicaset and update majority number
    majority $\leftarrow newMajority()$
    $AddedReplicaNotification$ addedReplica $\leftarrow (paxosInstance, newHost)$
    $triggerNotification($addedReplica$)$
```

```
**upon event** $RemovedReplicaRequest(toRemoveHost)$ **do**
    $replicaSet \leftarrow replicaSet \setminus \{newHost\}$
    // it just check the replicaset and update majority number
    majority $\leftarrow newMajority()$
    **if** $toRemoveReplica = myself$ **then**
        $clearAllTimers()$
    **end if**
    $RemovedReplicaNotification$ removedReplica $\leftarrow (paxosInstance, toRemoveHost)$
    $triggerNotification($removedReplica$)$

**upon event** $PrepareMessage(INpaxosInstance, INsequenceNumber, almostLeader)$ **do**
    $sequenceNumber \leftarrow higherSequenceNumberByPaxosRound.get(paxosInstanceNumber)$
    **if** $INsequenceNumber > sequenceNumber$ **then**
        $PrepareOkMessage\ prepareOK \leftarrow (INpaxosInstance, INsequenceNumber, almostLeader)$
        **if** $INpaxosInstance < paxosInstance$ **then**
            $lostMessages \leftarrow recoveryPhase(INpaxosInstance)$
            $prepareOK \leftarrow lostMessages$
        **end if**
        $sendMessage(prepareOK, almostLeader)$
        $leader \leftarrow almostLeader$
        $LeaderChangedNotification\ leaderExchange \leftarrow (almostLeader)$
        $triggerNotification(leaderExchange)$
    **end if**

**upon event** $PrepareOKMessage(INpaxosInstance, INsequenceNumber, almostLeader, set)$ **do**
    **if** $INpaxosInstance = paxosInstance$ **then**
        $sequenceNumber \leftarrow higherSequenceNumberByPaxosRound.get(paxosInstanceNumber)$
        **if** $INsequenceNumber = sequenceNumber$ **then**
            $proposerHigherSeqNumberCounter++$
            $checkMajorityPrepareOk()$
        **end if**
        $LeaderChangedNotificationleaderExchange \leftarrow (leader, paxosInstanceNumber)$
        $triggerNotification(leaderExchange)$
    **end if**

**procedure** $checkMajorityPrepareOk()$ **do**
    **if** $proposerHigherSeqNumberCounter = majorityNeeded$ **then**
        $cancelTimer(PrepareTimer)$
        $leader \leftarrow myself$
        $timeToWait \leftarrow NO\_OP\_DELAY\_LEADER$
    **end if**

**upon event** $AcceptMessage(INpaxosInstance, INsequenceNumber, message)$ **do**
    **if** $INpaxosInstance = PaxosInstance$ **then**
        $sequenceNumber \leftarrow higherSequenceNumberByPaxosRound.get(paxosInstanceNumber)$
        **if** $INsequenceNumber = sequenceNumber$ **then**
        end if
        $AcceptOkMessage paxosAcceptOk \leftarrow (INpaxosInstance, INsequenceNumber, message)$
        for $Host\ replica : replicaSet$ do
            $sendMessage(paxosAcceptOk, replica)$
        end for
        $cancelTimer(noOpTimer)$
        // check if I'm leader or not and use the right delay
        $noOpTimer \leftarrow start(NO\_OP\_DELAY)$
    end if
```

```
upon event $AcceptOKMessage(INpaxosInstance, INsequenceNumber, almostLeader, set)$ do
    if $INpaxosInstance = paxosInstance$ then
        $sequenceNumber \leftarrow higherSequenceNumberByPaxosRound.get(paxosInstanceNumber)$
        if $INsequenceNumber = sequenceNumber$ then
            $proposerHigherSeqNumberCounter++$
            $checkMajorityAcceptOk()$
        end if
    end if
```

For the NOOP method, called from the NOOPTimer, if the Host is the leader it sends a NOOP message to every replica, and reset the NOOP timer. If the Host is a replica it means that it didn't receive an operation from the leader for a wile, so it try to do Prepare, setting the PrepareTimer, deleting the NOOP timer. O metodo Prepare is called from this method.

4.2 Ideal vs. Reality

As say above the real practical implementation of Multi-Paxos combined with the Pub/Sub protocol has a little different from the version described in the pseudo-code.

The main difference that we should point out is that in the practical implementation the Multi-Paxos algorithm is not totally impartial to the request issued by the Pub/Sub layer, in short words the Multi-Paxos does not operate the requests as being a black-box that represents any type of possible request (Add Replica, Remove Replica and Propose operation). In the practical implementation there is a specific request for every type of operation and a individual.

The group later agreed that this was not the most optimal implementation, but opted for this decision because it was simpler to understand, describe and even debugging the solution developed.

Other option made by the group that we later regret about was related to the addReplica operation. This operation in ours implementation is directly issued to Paxos without passing to the Pub/Sub layer. The notification however are delivered as described as above as thus guarantees the global uniform vision on the replica set by the Pub/Sub Layer.

5 Publish/Subscribe

In the first phase, the Pub/Sub layer was used as a simple filter between the client and the Broadcast layer. In the second phase, this layer stopped to apply the filter and to memorize the subscriptions since it was the work of the Scribe layer. In this final phase, with the addition of the replication mechanism, the Pub/Sub layer is the State-Machine being replicated and has to interact with Multi-Paxos. Unlike the previous phase, Pub/Sub has to interact with the Scribe and Chord protocols in a different way, regarding the publishing of messages. The subscribe and unsubscribe operations have the same execution as in the previous phase.

The interactions, regarding messages, requests and notifications, will be revisited in the sections below.

5.1 Messages flow

The messages can be divided in **Intra** and **Inter Layer**.

5.1.1 Intra-Layer messages

The messages that flow in the PubSub Layer, between different nodes, are:

- **Leader Announcement Message**. Message sent by a node whenever it receives a FindRoot Notification from Chord. It is used by the node to announce itself as the leader of the topic that the other node is trying to publish a message. In the message content is also present the list of backup replicas of the node.
- **Publish Request Message**. Message sent by a node that wants to publish a message, to the node that is the topic owner. This message can be sent in two different places. When PubSub receives a publish request from the client, if pubsub already knows the owner of the topic, a PubRequest Message is sent directly to this node. If PubSub doesn't know the owner of the topic, a request to Chord is made (FindRoot Request) and sometime after a Leader Announcement will be received; then the original node that wants to publish a message sends a PubRequest Message to the topic owner.

5.1.2 Inter-Layer messages

The PubSub protocol interacts with the Client using the following interactions:

- **SUBSCRIBE Request**. Request received from the Client when the client wants to subscribe to some topic.
- **UNSUBSCRIBE Request**. Request received from the Client when the client wants to unsubscribe to some topic.
- **PUBLISH Request**. Request received from the Client when the client wants to publish a message to some topic.

The PubSub protocol also interacts with:

- Scribe, in **Dissemination Request** and **Message Delivery**. These interactions are briefly described in 3.3.
- Chord, in **FindRoot Request** and **FindRoot Delivery**. These interactions are briefly described in 3.1.
- Multi-Paxos, in **Propose Request** and **Decided Delivery**.

5.1.3 Pseudocode

```
Interface :
  Requests :
    SubscribeRequest(topic)
    UnsubscribeRequest(topic)
    PublishRequest(topic, m)
  Indications :
    SubscribedMessageDeliver(topic, m)



upon init () do
  $replicasByTopic \leftarrow \{\}$
  $myself \leftarrow (id, myHost)$
```

```
    $myReplicas \leftarrow \{\}$
    $pendingMsgs \leftarrow \{\}$

upon event $SubscribeRequest(topic)$ do
    trigger $DisseminateMessage(SUB, topic, message = null)$

upon event $UnsubscribeRequest(topic)$ do
    trigger $DisseminateMessage(UNSUB, topic, message = null)$

upon event $PublishRequest(topic, m)$ do
    $replicaSet \leftarrow replicasByTopic.get(topic)$
    if $replicaSet \neq \{\}$ then
        $replica \leftarrow getOwner()$ //OR $selectRandomReplica(replicatSet)$
        trigger $SendPubRequestMessage(replica, topic, message)$
    else
        trigger procedure $addMessageToPending(topic, message)$
        trigger $SendFindRootRequest(topic, originalSender = myself)$
    end if

upon event $MessageDeliver(topic, m)$ do
    trigger $SubscribedMessageNotification(topic, m)$

upon event $FindRootNotification(topic, m)$ do
    $Send(LEADER\_ANNOUNCEMENT\_MESSAGE, topic, myself, myReplicas)$

upon event $LeaderAnnouncementMessage(topic, owner, replicas)$ do
    $msgsToSend \leftarrow pendingMsgs.get(topic)$
    for message in msgsToSend do
        trigger $Send(PUB\_REQUEST\_MESSAGE, owner, topic, message)$
    end for

upon event $PubRequestMessage(topic, message)$ do
    $byteBufer \leftarrow ENCODE((\text{"}PubMessage\text{"}, topic, message))$

upon event $DecidedNotification(byteBuffer)$ do
    $message \leftarrow DECODE(byteBuffer)$
    $msgType \leftarrow message.msgType$
```

```
if $msgType = "AddNode"$ then
    -
end if
if $msgType = "RemoveNode"$ then
    -
end if
if $msgType = "PubMessage"$ then
    -
    (...)
    trigger $DisseminateMessage(PUB, message.topic, message.m)$
end if
if $msgType = "LeaderExchange"$ then
    -
end if
```

6 Client

The full description can be found on the first phase report.

6.0.1 Manual Client Commands

The full description can be found on the first phase report.

7 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.

7.1 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.

7.2 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.

7.2.1 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.

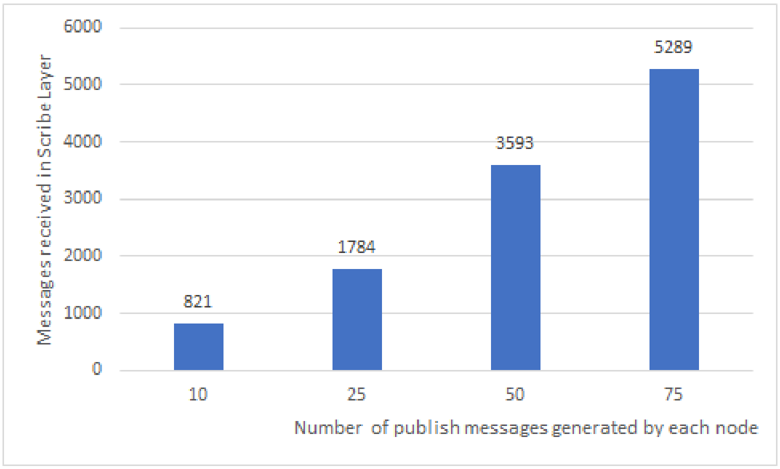

Figure 3: Total number of messages received in the Scribe Layer by all nodes

7.2.2 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.

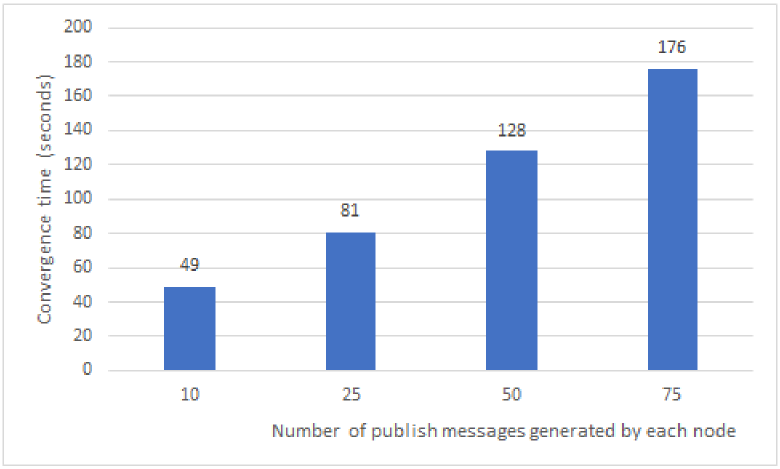

Figure 4: Convergence time

7.2.3 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.

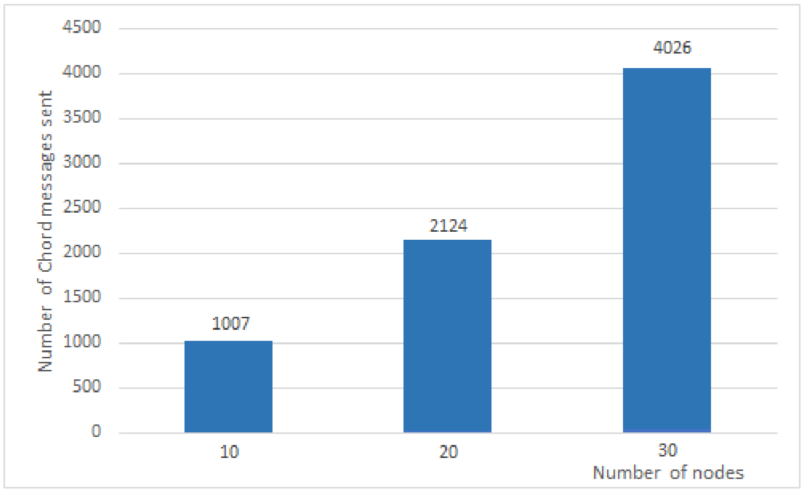

Figure 5: Total number of Chord messages sent by all nodes

7.2.4 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.

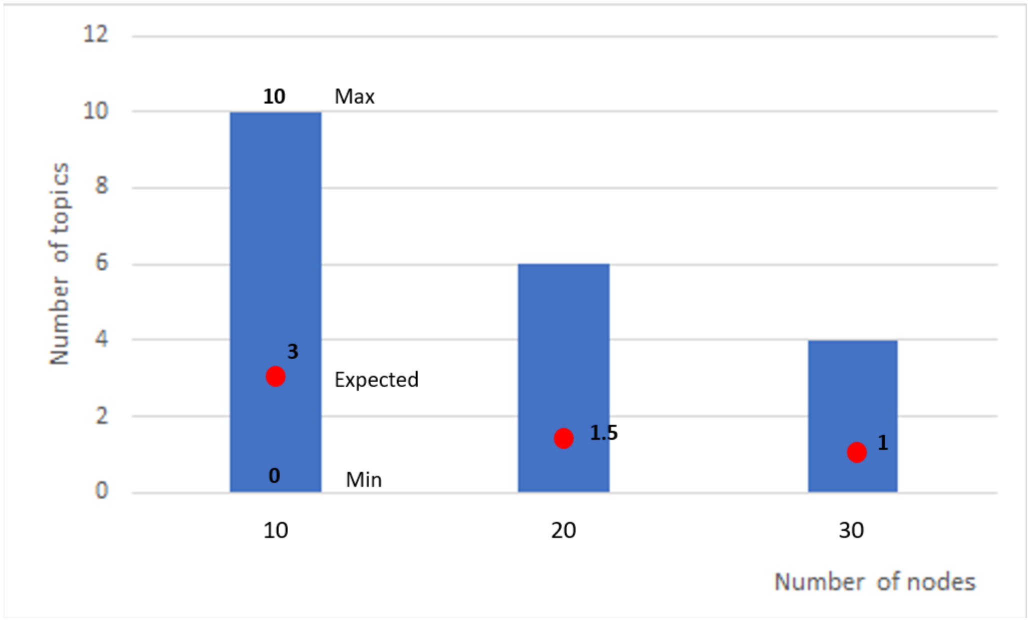

Figure 6: Load Balancing on the Chord Layer

7.3 Reliability with failures

These 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). In this final phase, with the corrections in the DHT and Tree management, we were able to test Reliability under small levels of failures, obtaining better results than in the second phase (with near 100% of reliability in all tests). The Chord layer was also tested apart, and the regeneration of the table fingers and of the ring structure were positively maintained.

7.4 Testing Multi-Paxos

Finally, the last set of test were aimed at testing Paxos. Although this set of tests were small, due to time constraints, we were able to obtain a good measure of correctness of the implemented solution. We tested if the replication was being well performed, mainly testing if all the replicas received the same information in the ordering of messages. Other operations like Leader election and adding and removing of replicas were also focused in the testing, also obtaining good correctness results.

7.5 Discussion

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.

8 Conclusion

The goals for this 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); implementation of Multi-Paxos. 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.*