Option B: Publish-Subscribe using an Unstructured Overlay Network

ASD TP1 2018/2019

Manuel Henriques 42546, Sebastião Pamplona 42735

1. Introduction

The project consisted in implementing a publish-subscribe protocol, on top of an unstructured overlay network. To implement the unstructured overlay, we implemented the HyParView protocol, a membership protocol for reliable gossip-based broadcast. As for the publish-subscribe, we resorted to the Plumtree, a protocol that makes use of trees for epidemic broadcast.

2. Overview

In this section, we cover the essentials of each component of the project, as well as the protocols chosen to implement them.

2.1 Unstructured Overlay Network

An unstructured overlay network is built by establishing random neighboring relationships between nodes, thus having a random topology. Some interesting aspects of these types of overlays is that they have a low cost on building and maintaining, usually evenly distributed by all nodes, and they normally present a natural level of redundancy, making them more robust, not only to node failure as well as to message losses.

2.1.1 HyParView

The HyParView protocol is based on an approach that relies in the use of two distinct partial views, which are maintained with different goals by different strategies.

There are two different main strategies that can be used to maintain partial views, namely reactive, which changes the partial view as a reaction to some event that affects the overlay, or cyclic, updating the partial view as a result of some periodic process.

Active view - used for message dissemination. This view is relatively small, with size $fanout + 1$, and symmetric links, meaning that if a node q is in the active view of the node p, then p is also in the active view of q.

A reactive strategy is used to maintain this view, reacting to two events: a new node joining the system and a node failure detection. It's important to notice that each node tests its entire active view every time it forwards a message, thus implicitly testing the entire broadcast overlay at every broadcast, allowing a very fast failure detection.

Passive view - used for reconstructing the active view, when failures occur, by maintaining a list of nodes that can be promoted to the active view. It should be noted that the overhead of this view is minimal, since no connections are kept open. This view is usually larger than the active view, with a minimum size of $log(n)$.

A cyclic strategy is used to maintain this view, each node (periodically) performing a shuffle operation with onde random node, possibly updating its passive view. Essentially, a shuffle operation consists of exchanging a set of node identifiers, composed by the node itself and some members of the passive and active view.

2.1.2 Fault tolerance

There are two ways a fault can be detected: when trying to contact a node, the connection cannot be established and after a node crashes, if it is done with the command "quit" (which

terminates the system, with the scala method *system.terminate( )*), that node sends a crash notification to all the members in its active view (done by overriding the method *poststop( )*).

2.2 Publish-Subscribe

The publish subscribe architecture doesn't require special programming for senders to send messages directly to specific receivers. This property guarantees that no special knowledge is necessary to disseminate messages and to know which processes are subscribed to certain topics.

2.2.1 Plumtree

The Plumtree is the component that materializes our publish-subscribe message dissemination. Although the Plumtree itself doesn't store the topics that a process is subscribed to, it guarantees that the process only delivers the same message once.

Spanning Tree Construction - When it is initialized, the Plumtree has the ability to request information on neighbors from the HyParView protocol through the Peer Samping Service. Then, it selects a subset of those neighbors (bigger than 1 but not exceeding the fanout of the protocol).
Each broadcast creates an unique message identifier using the current system time of the process, the message content and the information of the sender, guaranteeing that no message is delivered more than once but also ensuring that every broadcast by different processes are disseminated correctly.
We opted to not use a policy to select a subset of messages from the lazy queue, sending *IHave* messages immediately after they are created.

Tree Repair and Optimization - Because *IHave* messages are sent after being created, processes do not receive a set of *announcements* on a single message.
Our approach made some changes on the control flow of lazy push messages, which will be explained later in this report.

2.2.2 Peer Sampling Service

In our approach, the Peer Sampling Service is responsible for keeping track of subscribed topics, creating the other protocols and forwarding messages between the HyParView and PlumTree protocols when they need to communicate with each other (eg. PlumTree requesting neighbors when created).
It also has the ability to *Subscribe* to topics, *Unsubscribe* to topics, *Publish* messages by topic and to *Deliver* received messages.
Finally, each process has an application that subscribes to 5 random topics (from 50), publishes *x* amount of messages and stores all the information in a text file for testing purposes.

3. Specification and Arguments

3.1 Peer Sampling Service

Our application can be tested locally with predefined commands. Nodes can be created by using the '*sbt* "run <ip-address> <port>"' command on a terminal in the root of our project. The first process to be initialized must be the contact node with *'sbt "run 127.0.0.1 2550"'*. After initialization, every node subscribes to 5 random topics.
Alternatively, one can run the script *test_script_17.sh*, which launches 17 nodes, starting with the contact node at 127.0.0.1:2550.
After that, the commands have to inputted manually (i.e., run the command *start* on every node, and after they are done, run the command *received*).
Each process allows the use of the following commands in the terminal:

- **"help":** lists commands.
- **"quit":** quits the system.
- **"neighbors":** prints a set with the current neighbors.
- **"subscribe":** subscribe to a topic.
- **"unsubscribe":** unsubscribe a topic.

- **“publish”:** publish a message with topic T and message M.
- **“start”:** starts a loop of 40 publishes to random topics.
- **“received”:** creates text files for the 3 protocols to be used with the python script.

The contact node has a special command to shutdown *n* random nodes, according to the failure percentage.

- **“kill_random”:** shutdown *n* random nodes, according to the failure percentage.

3.2 Overview

Only processes that were created by the system can broadcast messages, ensuring no creation.

3.3 HyParView

The HyParView protocol guarantees a strong resilience to node faults, maintaining the overlay connected even if a high percentage of nodes fail. This assures that, even when there is a high percentage of node failure, there is a very high (100% most of the time, even for node failures percentages such as 85%) probability that a message that was broadcasted by a correct is delivered by another correct node.
The active view cannot have duplicate nodes in it, ensuring that a certain message broadcasted by a correct process is not broadcasted to the same node twice.

3.4 Plumtree

When a message is published, the node delivers the message before starting the broadcast. This way, we ensure that every correct process that broadcasts a message, delivers the message.
The Plumtree guarantees that no message is delivered more than once by each process. This is assured by using a set of message IDs that control which messages were already delivered.
Nodes can only deliver messages when they broadcast or when they receive a message. This way, we guarantee that if a process delivers a message, that message was broadcast by some correct process.
Every broadcast is disseminated by a series of eager push and lazy push messages. Because the overlay has full connectivity, every message is eventually delivered by every correct process.

We had a problem with the Plum Tree protocol that was related to the creation and deletion of timers. Because we were creating timers inside timers while removing tuples from the *missing* set, our code suffered from infinite loops with null values.

1. We create the timer for the first time like the protocol specifies.
2. When the timer executes for the first time, we start by removing the first occurence of a missing entry with id equal to message id. If that entry is null, we do not create another timer and do not continue with a trigger on the event Send(GRAFT) - that would contain null values.
3. When we receive a message that we did not deliver before, we always delete the timer related to that message, even if there is no occurrence of that message in the *missing* set. We do this because we always remove a entry of message id when creating a timer. So, if we used the approach from the paper, we wouldn't be deleting timers that would have no effect in the future.
4. When we receive a message for the first time, we also delete every occurrence of that message from the *missing* set. If we already received the message we wanted, there is no need to keep a record of it.

4. Evaluation

For all of our tests we run 17 terminal tabs on our OS, using the commands described in “3.1 Peer Sampling Service”. The file “application.conf” stores the constants used by our protocols.
The test were ran with the following parameters for each protocol:

- HyParView: active view size of 4, passive view size of 6, ARWL of 3, PRWL of 2, passive update time of 20 seconds, shuffle RWL of 2, ka of 1 and kp of 2.
- Plumtree: timeout1 of 500ms, timeout2 of 100ms, threshold of 10 and fanout of 4.

4.1 Experiments

We tested our solution with 17 nodes, each subscribed to 5 topics and each publishing 40 messages to random topics (from 50 topics), every second, for a total of 680 messages sent in 80 seconds.
After initializing every process, we wait for the overlay to stabilize. Then, we start the publishing process (saturating the overlay) and wait for the message dissemination to stabilize as well. Finally, we create the text files for every process' protocols messages.
We tested our solution with and without the optimization procedure.

4.2 Experiments With Fails

We used the same strategy of topic 4.1, but with 10%, 50% and 85% of failed nodes.
The only difference was: after creating the nodes, we stopped some of them and waited for the overlay to stabilize.

4.3 Results

-	Sent	Received	Reliability
S/ Opt	1127	1127	100%
C/ Opt	1185	1185	100%
Fail 20%	810	896	90.40%
Fail 50%	249	249	100%
Fail 85%	36	36	100%

Table 1. Published messages sent and received

-	Eager	Lazy	(Eager/Lazy)
S/ Opt	15182	52694	28.81%
C/ Opt	21586	58068	37.17%
Fail 20%	8457	36897	22.92%
Fail 50%	2262	6994	32.34%
Fail 85%	244	238	102.52%

Table 2. Eager push and Lazy push messages sent

-	HyParView
S/ Opt	1307
C/ Opt	1567
Fail 20%	1487
Fail 50%	758
Fail 85%	364

Table 3. HyParView messages disseminated

Table 1 - The optimization was tested with values 4, 5 and 6 for the threshold. With values 5 and 6, we almost had the same results when not using optimization. With value 4, almost 45% of the Plumtree messages were from the eager push model, showing the worst results of all the approaches. We suspect that this is due to the fact that we do not introduce new nodes to the overlay (no new and better paths are created) and the repair process, influenced by the lazy push model, did not have a negative impact during our stress test.

We think our model handled fails with great success, showing reliability of 100% even after 85% of node fail and overlay reconstruction.

Table 2 - Considering that lazy push messages are smaller than eager push messages, having an average of 70% of the messages being lazy push is a positive achievement.

Table 3 - The HyParView results did not change much during each test. This is due to the fact that the publishing of messages do not influence the maintenance of the overlay but failing nodes do.

In summary, we think our approach showed positive results, with a high reliability and a good ratio of eager to lazy push messages.

5. Conclusion

We found it challenging to do the project in Scala, because we had never programmed in this language. Overall, we found the language to be a lot more satisfying to work with, for distributed systems, compared to Java.
Regarding the project itself, we appreciated the fact that we had many different options to implement our solution, when compared to the projects of prior courses. And the fact that we had to research some papers was also new and challenging for us.

REFERENCES

[1] J. Leitão, J. Pereira and L. Rodrigues. HyParView: a membership protocol for reliable gossip-based broadcast, 2007.
[2] J. Leitão, J. Pereira and L. Rodrigues. On the Structure of Unstructured Overlay Networks
[3] J. Leitão, J. Pereira and L. Rodrigues. Plumtree: Epidemic Broadcast Trees.