Distributed System 2 - 1st Assignment report (A Broadcast-only Communication Model Based on Replicated Append-only Logs)

The L0ne Pr0grammer
Ivan Martini, 207597
ivan.martini@studenti.unitn.it

November 2020

1 Introduction

This report is the documentation of the work done for the second assignment of the *Distributed System 2* course. The group, as stated above, is "The L0ne pr0grammer", which is composed just by me. The given task is the implementation (and evaluation) of a paper published in December 2020 [1], which is related of the previous project paper [2]. The repo, although already embedded into the professor repository, can be always found at the URL:

`https://github.com/Iron512/GossipBroadcastDS2`

Right there you will find the last updated versions (in case those ever exist in the future), feel free to contact me at any time to discuss about it. This report will be a little more elaborated than the first one, as the workflow I adopted has been different. Indeed, even if I delivered the first assignment on time, I implemented only most of the features I planned to do, excluding some others i worked on, since those were working correctly only some times (which means that they are not working at all).

For this reason, just a few days after the delivery I started refactoring that same project, in order to have a better version of the infrastructure to develop the second project on. All the changes are in the previous repository, actually under the branch `updated`. In this report I decided to embed also part of the refactoring process, as I feel this has been really useful to expand my knowledge on the *Agent Based Modelling*, even without evaluation. The general structure of the report, however stays more or less the same: After a recap on the first project and some discussions on the refactoring process (Sect. 1.1), there will be the sections dedicated to the definition (Sect 2) and implementation of the gossip system (Sect 3) as well with some personal consideration and explanations on the choices taken. Then there will be a brief section on running and configuring the system, as the latter underwent some changes (Sect. 4). Finally the sections dedicated to the analysis (Sect 5), which has been developed more in depth compared to the first one and the final considerations (Sect 6).

1.1 Project Refactoring

As said, I wanted to spend some words on the refactoring process, as it has been essential in the development of this second assignment. Analyzing the precedent development, I tried to isolate the weak spots in order to resolve and fix them. The first one was pretty easy, as i already knew it: even if i managed somehow to keep everything ordered, there were some messy code, which i simply fixed adding more hierarchy. The new code presents some abstract classes and many extension to them, which provide an higher level abstraction and help keeping the consistency required for the modular design [3]. The second part was a little tricky. For every project I take, even if usually there is no continuation, i try to develop in order to have other people take on my code and go on. Using this mindset just helps me to develop better and what I felt missing from the implementation i chose was the lack of a real customisation. The parameters offered by repast are pretty handy, but they are limited and offer only a narrow degree of freedom. I solved this issue by developing a simple JSON file parser, in order to store a wider set of configuration, which will be explained in further sections.

2 Protocol Description

This paper, differently from the precedent offered a more high level solution, based on unicast messages, to keep consistency among different append only logs, using a gossiping protocol. The concepts of the elements interacting in the system are described and here briefly reported. During the implementation i followed the straightforward definition, pruning the unnecessary add-ons.

1. **Event** - is the basic unit of information, the building block of the whole "append-only log". It provides the content and the index of the message (unique for each creator) and the hash of previous message (which is more powerful than just checking the index). It also provides the public key of the creator and the signature of the content. I decided to avoid the usage of the PKI infrastructure, as it doesn't affect the algorithm workflow even if in the real world they are a must (*Signatures* provide integrity and authentication, which are not probed in this simulator). I opted to use the *Relay* entity that generated the message as identifier, and this is the only change that i brought to the paper.

2. **Log** - is the direct implementation of the "append-only log" (as the name suggest) which titles the paper: An orderd list of *events*, which is enforced to provide some properties (restricted possibility to append, connection, monotonicity).

3. **Store** - is a collection of logs, held by the actors (*Relays*). Each one has its own store and at the goal of the algorithm is to keep all the stores eventually consistent.

4. **frontier -** is an abstraction obtained by the store. It is a set of pairs $\langle relay, event \rangle$, where the *event* is the last event generated by *relay* inserted in the store.

Although these are the only specification, i just assumed the existence of some generators (called *Relays*, to keep consistency with the old protocol) and the network overlay is "bad": A **dynamic network** (relays can join and leave) with some **jitter** and **packet losses**.
In the paper, two different flavours are specified, a naive approach (*open gossip*), where nodes replicate all the log without any distinction and a *transient gossip*, where, though a mechanism of "follow/block" each relays decides which logs to synchronize an which ones to not consider. This latter implementation is quite common on wide network, where the interactions are "limited" to just some components.

3 Architecture Development

The architecture is similar to the precedent that i proposed, at least on the communication part. Let's however browse all the classes implemented.

3.1 Event

As mentioned, is the founding block of all the project. It took the place of the *Perturbation*, as it fulfills the same purpose. Other than the *standard methods* (Constructor, equals, hashcode, toString) and the *getters*, there are two methods to retrieve the next reference and the hash of the previous message.

3.2 Log

The log class contains the identifier of the log and all the events generated. It contains only the method explicited in the paper: Constructor, append (only to self), last, getId and update.

The events are stored using a classic *ArrayList* of *Event*, and the class itself is basically a wrapper to enforce some operation on the list and avoid an incorrect usage (i.e. the possibility of removing an event). The most important method of this class is the *update* one, which inserts each newer event in the *log*, starting from the events stored in another log.

3.3 Store

The store class is also implemented as specified on the paper. It handles a *Map* which associates *Relays* with *Logs*. As in the *Log* class, the goal is to restrict the normal usage of the map, in order to obtain the consistency required. The most important method is the *update* one, which fills the store with the new keys and updates each log (calling the *update* function written above)

3.4 Manager and Wavefront

These two classes are basically the same from the old project, but they got renewed where needed.

The *Wavefront* are the way to exchange messages through the channel, but the time taken from the generation and the delivery is now determined using a normal distribution, configured for each node using **latency** and **jitter** of the packets. This value is then multiplied from the distance (in order to have again closer nodes to receive the messages faster). The second change is the abstraction of the class, which is extended by the `PushPullWavefront` and the `ReplyWavefront`, which are identical, but distinguished at runtime (which is necessary to understand when to reply to a wavefront and also to develop the network graph).

The *Manager* as always act as a *ContextBuilder*, handles the distribution of the wavefronts (at the end of each tick, in order to avoid mismatching) and it cares the run termination (after the protocol correct termination). Again, after a predetermined tick, the Manager tells the nodes to stop termination, in order to have the algorithm to reach the wanted eventual consistency.

3.5 Relay

This class, which was the most complex of the previous project has been refined a lot. Now the Relay class is abstract as well and it is extended by 2 classes: *OpenRelay* and *TransientRelay*. The abstract class handles the common *standard methods* and the *getters/setters* of the class, as well as the **step** method, which is executed at each tick.

Here the relay decides if generating a new event and join/leave the network for a while. It then checks if there are new messages and acts accordingly (i.e. if it received a *pushpull* it will answer with a *reply*).

The abstract methods are called: *onSense*, *compareStatus* and *printStatus*. The first determines what happens when a message is received (and how the Stores are updated), the second returns a boolean telling if two stores are identical and the last just prints out the frontier at the end, just to confirm that the situation is consistent.

The *OpenRelay* keeps all the logs updated, while the *TransientRelay* holds two lists: one for the blocked logs and one for the followed logs. Whenever it receives a store containing a new log, it decides (randomly and arbitrary) if it is going to follow that log or just block it.

3.6 Parser

The most important change done to the project it has been the higher degree of freedom and customization. At the load of the scene, a JSON file, is loaded, and its parsed thought this class. The file stores all the configuration needed to the project and it looks like:

```
{
        "Dimensions" : {
                "X":50,
                "Y":50
        },
        "Duration": 30000,
        "Network": "OpenRelay",
        "FixedRelays": [
                {
                        "X": 10.0,
                        "Y": 25.0,
                        "PerturbationProb" : 0.01,
                        "Latency": 15.0,
                        "Jitter": 3.0,
                        "SpawnProb": 0.006,
                        "LeaveProb" : 0.005,
                        "LossProb" : 0.1
                }
        ],
        "RandomRelays": [
                {
                        "Count" : 2,
                        "PerturbationProb" : 0.01,
                        "Latency": 15.0,
                        "Jitter": 3.0,
                        "SpawnProb": 0.006,
                        "LeaveProb" : 0.005,
                        "LossProb" : 0.1
                }
        ]
}
```

The explanation is quite intuitive (at least I hope): *Dimensions* passes X and Y as the values of the size of the field, *Duration* the number of total ticks of execution, after which the termination protocol will take place and the *Network* type (Which can be composed by istances of **OpenRelay** or **TransientRelay**).

The arrays of *FixedRelays* and *RandomRelays* are instead used to specify more precisely the relays in play. Each relay contains the *Perturbation Probability*, *Latency*, *Jitter*, *Spawn Probability*, *Leave Probability* and *Loss Probability*, but fixed relays also have the position established (X and Y) and random relays have the *count* (i.e. how many randomly positioned relay are wanted.)

Even if this improvement might seem trivial (actually, the implementation is it) it opens a wide set of opportunity for the simulation part. Having a fixed topology, differentiating Relays able to produce events from the ones which only replicates them, creating low latency nodes or particularly slow ones are only few of the possibilities that this approach offers. The configuration files are actually 3, and stored under `/src/config` inside the project folder.

4 Building/running instructions

To run the simulator, it must be imported in the *Eclipse IDE*, along with the last version of Repast Simphony. Once imported the simulator can be launched clicking the *run* icon and selecting `TheL0nePr0grammer_2ndAssignment model`. By default the `config_1.json` file is set in the parameter tab, but there are also `config_2.json` and `config_3.json` or any file you can possibily create, following the rules described above. If the file is malformed, the simulator will throw the correspondent exception and stop the execution.

On the side the *DataLoader* and the *Displays* can be found. Two displays are implementend, one showing the Relays and the messages exchanged to synchronize their stores and the other shows the relationship between the transient relays. Using an *Open Relay* configuration will show this last display blank, but using a *Transient Relay* (i.e. `config_2.json`) one will provide green arrows (`follow` relationship) and red arrows (`block` relationship)

In the same window, the *Data Set* and the *Chart* on the **New Event distribution** can be found, but they will be discussed in the next section.

5 Simulator Evaluation and Analysis

I wanted to keep some form of bounding with the previous simulator, so i carried an analysis on the number of message exchanged. In particular I wanted to track the number of new event that any node discovered during each tick. I run the experiment on both a fixed topology and a random topology; Here I decided to present only a subset of the graphs to avoid to lengthening the report too much. I chose the most characterized graph, as with similar scenarios i obtained quite similar results (as expected).

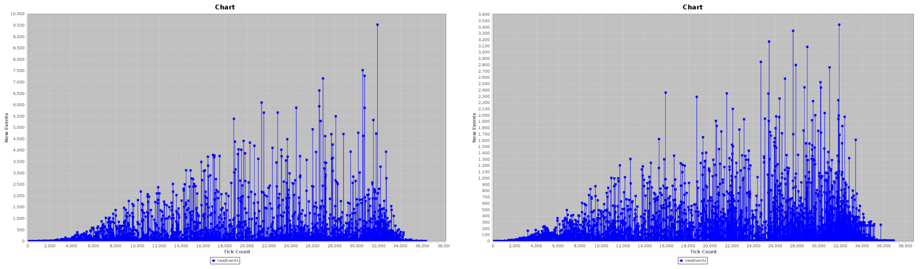

Figure 1: A test run with 30000 total ticks and 140 relays. On the left side, the model adopted is the *Open Relay*, while on the right side there is the *Transient Relay* one.

In Figure 1, having the same number of relays (140, 35 generating events and 105 just listening) we can see how the graphs are extremely similar. The keeps rising (in an irregular and not monotonic way of course) until some ticks (∼ 2000) after the terminating protocol. Then the curve rapidly slows until eventually reaching 0. The big difference is in the scale however: The *open relays* scale is almost 3 times bigger than the *transient relays* one.

This result might seem trivial, as the number of event registered in the second case is lower (not all logs are present in each stores → less events to update) but is however unexpected: The *transient relays* are hardcoded to follow 80% of the logs, which should reduce (assuming a linear correlation, which evidently doesn't hold) the events of about $\frac{1}{5}$. This result is quite impressing (at least to me, I honestly didn't expect it), and justifies (as we could expect) the large usage of *transient nodes* in wide networks.

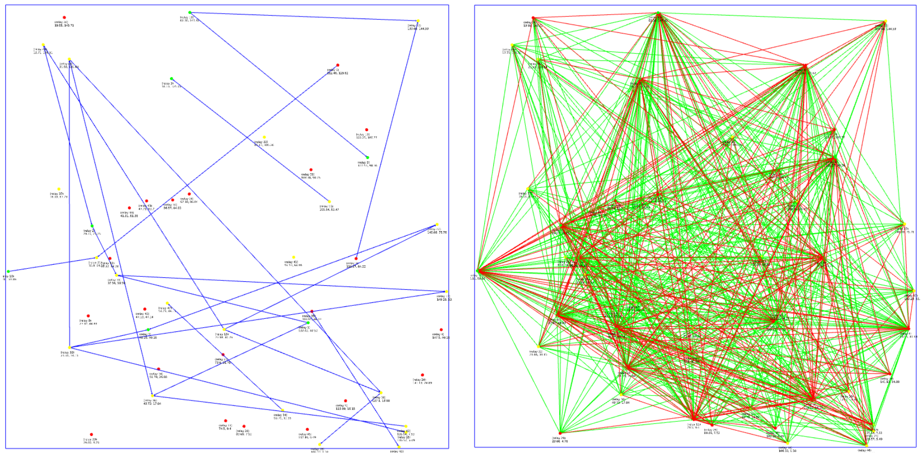

Figure 2: Two screenshots of the two different views available: the *field* (on the left) and the *transient relationship* graph (on the right)

In Figure 2 instead, the spatial dimension is presented. This is obviously not a quality measure, but I wanted to show the two views presented to the user, which are described above.

6 Conclusions

As in the first project, it has been an interesting challenge to implement, also due to the novelty of the papers to be implemented, which have been published really recently. In this case, I implemented all the features that i planned to, providing a reliable and highly customizable product. Algorithm like this offer lot of tweaking parameters that can be explored (the frequency of the synchronization for example, which I just chose arbitrary) and thus can be implemented in very different environments always with a good trade-off.

References

[1] Lavoie Erick Kermarrec Anne-Marie and Tschudin Christian. Gossiping with append-only logs in secure-scuttlebutt. `http://ericklavoie.com/pubs/dicg2020.pdf`., 2020.

[2] Tschudin Christian. A broadcast-only communication model based on replicated append-only logs, *ACM SIGCOMM Computer Communication Review.* `https://ccronline.sigcomm.org/wp-content/uploads/2019/05/acmdl19-295.pdf.`, 2019.

[3] David L. Parnas. *Software Fundamentals: Collected Papers by David L. Parnas.* Pearson Education (US).