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 first 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 April 2019 on the *SIGCOMM Computer Communication Review (CCR)* journal [1]. The repo, although already embedded into the professor repository, can be always found at the URL:

`https://github.com/Iron512/BroadcastLogsMartiniDS2`

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. The report will briefly describe the content of the referenced paper (Sect. 2) and the architecture I designed to implement the simulator (Sect. 3). My goal on this section is to provide not only the crude architecture, but also all the questions that guided me during the development, in order to ease the understanding of most of my choices. Finally, the running instruction will be presented (Sect. 4) along with the evaluation of the simulator (Sect. 5) and the conclusion on the whole experience (Sect. 6)

2 Protocol Description

This is just a brief recap of the paper presented above ([1]). For a detailed description consult the original source

The protocol presented by Christian Tschudin, aims at disrupting the base communication abstraction in most of network (which is naturally unicast based) with the proposition of a true broadcast system, on the model of some existing algorithms, summarizing the desirable properties that such architecture should provide. Typically broadcast domains lack of consistency, since the source is not

aware of the presence of targets and vice versa, therefore they are often set aside.

The key of the solution presented is a clever translation: The packet propagation problem becomes a data structure problem, so keeping that structure consistent means keeping the whole network consistent. The data structure proposed in this paper, is the append-only log (*frontier*), that works perfectly even with its simplicity. The basic concept is the one of **Solitons**, which are *information packets* propagated as a solitary wave. With solitary wave we mean a wave that travels through some kind of media, *without leaving any disturbance behind.* Obviously the communication of information involves at least two entities: neither the generator nor the receiver of the solitons are aware of the presence of the other (obviously, upon receiving a wave, the latter will empirically discover the existence of the former) or of any other entity in the system. This is clearly the basic concept of a pure broadcast system, which must enforce this 3 properties:

- Each perturbation has a source identifier, that marks each perturbation generated (addressability)
- A perturbation *eventually* reaches all anonymous observers
- All observers sense subsequent perturbations coming from a specific source in the same order

The algorithm analyzed is build in an incremental way. Some easy use cases are presented, which are then developed in order to address more general scenarios. Different increments are proposed, while i developed 4 of them (the ones in **bold**). On my Github repo the 5^{th} increment can be found, but due to some code issues I preferred to not include it into the delivery, in order to present only a full operative, tidy and well tested project.

1. **Relay I** - The first implementation of the algorithm, works only on a static network without any jitter.
2. **Relay II** - In this increment, the network is dynamic, and the messages arrive with some delays.
3. Relay III - Here, at the cost of memory, also packet losses are tolerated.
4. **Point to point communication (Unicast)** - Using the already defined architecture, the sources can also generate a perturbation directed to one specific destination.
5. **Multicast and Pub/Sub Architecture** - In this case a message can be sent to different destinations using multicast (which is **not** implemented as set of unicast messages) or using the classic publisher/subscriber architecture. (In the project there are both implementations).
6. Privacy-Preserving Communication - As in both the points above, the same architecture is exploited to produce high level features, like the encrypted communication of some members. Note that of course only the recipient can decrypt the message, but the others can not even tell who the message is for (they can only determine if the message is for theirs or not).

7. *Some other increment concern Synchronization, Offline communication and Distribution (Replication without conflicts), but they are not part of the evaluation*

Even if just brief, this description allows the understanding of the next section, where the architecture adopted will be explained. Where the paper already propose a solution, i followed the straightforward implementation, where instead the paper doesn't I opted for the one (I thought) most coherent

3 Architecture Development

The architecture i developed was originally conceived for having 5 different classes to interact in the scenario. Working on it, i found different redundancies between two of them, so i decided to collapse one into the other. The scene is composed by the *Perturbation*, the *Wavefront*, the *Relay* and the *Wavefront Manager*. The latter implements also the "ContextBuilder" for the Repast philosophy.

3.1 Perturbation

The perturbation is the basic unit of a message. It contains the *source* that generated it, the *reference* (implemented as a simple logical clock) and the *dimension* of the packet. The original paper places the value instead, but i opted for the dimension, in order to provide the possibility of analyzing some throughput. Having implemented also points 4 and 5 of the assignment, the perturbations **might** also contain the destination(s) or the topic. Those are actually arbitrarily decided by the Relays, which, as we'll see below, generate the messages. The Perturbation can be initialized with 3 different constructors (one for the classic broadcast message, one for the uni/multicast message and one for the pub/sub architecture) and contains some getters methods

3.2 Wavefront

The Wavefront is the simulation of the soliton, described abstractly in Sect 2. It carries a perturbation over the designed medium. Different wavefronts can carry the same perturbation, as each Relay typically forwards any perturbation received. The main attributes are obviously the source and the perturbation (which as said might not coincide).

Inside the Wavefront, the delivery mechanism is implemented. The simulation is initialized with some parameters, two of them being `minDistancePerTick` and `maxDistancePerTick`. Those are the boundaries of the distance that a message can travel at each tick. Obviously this has no real mapping on the reality (as the wireless waves typically travel at the speed of light) except for some cases (like satellite communications) but i decided to implement this sort of abstraction to keep the model consistent. Indeed the simulation executes sequentially in ticks and now as a continuous flow.

This implementation allows the simulation consider different cases, having different times of reception depending on the distance. I thought a lot on a suitable

way to implement this feature (using for example just timers), but i finally chose this which offers both an high level of abstraction and a direct connection with the real measurements of average time and jitter (they can both easily mapped into the other).

Inside the class there are the standard constructor and some methods (equals, hashcode, ..), plus getters as in the class *Perturbation*. Two custom methods handle the messages, the *live* method, used by the Relays to determine if a Perturbation has been sensed or not and the *track* method, used by the Manager to determine whether or not a wavefront can still be sensed by any node or not (to keep track of the number of active wavefronts).

3.3 Relay

The Relay is the main actor of the system. It represents the node of the network and it can both generate and receive messages. On the paper Relays are descripted as "observers", which is just a general case of my implementation: A normal Relay can indeed become a simple Observer placing its `pertGen` parameter (which represent the probability of generating a message at each tick) to 0. The relays are not aware of the network composition nor the topology, therefore they communicate through a medium, implemented in the class *WavefrontManager*.

The class is built starting with the standard class methods (Constructor, equals, hashcode, .., getters) and is defined by an ID as required (for identification). The Relay also contains the position where it is placed and the manager that handles all the Wavefronts (sent and received). For most of the attributes i decided to use high level data structures, in order to keep the code easy to read/understand even tho this might result in a cost on the efficiency of the simulator (but not of the simulation). The frontier and the bag are an `HashMap<Relay,Integer>` and an `ArrayList<Perturbation>` which suited the properties required in the paper, while an `ArrayList<Wavefront>` called *incomingWavefronts* handles the wavefronts that are going to be sensed.

Obviously, in the real world, no *observer* would able to tell if and when it is going to receive a perturbation, since the perturbation is carried itself over some medium: The medium here is simulated by both this ad hoc data structure and the *WavefrontManager* class. The class also presents the list of topics subscribed and the probability of generating any kind of message (`double pertGen`).

The whole class is based on two methods: The method *step* handles the execution of the process checking if any message has been delivered and generating new messages, while the *onSenseMessageRelayII* is the straightforward implementation of the pseudocode present in the paper. The latter also handles the definition of the network edges, shown in blue on the GUI while running the simulator. The auxiliary method implemented are however necessary for the execution: the simulator will stop generating messages after a precise number of steps, but will run until all the frontiers are synchronized. This is possibile only using the auxiliary method *checkFrontiers*, which returns a true or false whether two frontiers are the same. Even if it might look trivial, defining when

two frontier are consistent might lead to some troubles. Indeed a Relay r might contain the reference of a Relay s that logged out from the network before Relay t joined the network. Obviously t has no clue on the existence of s, therefore the frontier of r and t doesn't coincide on the value of s (t will have a null frontier for Relay s), so I decided to consider two frontier identical if each element present in one is either null or identical in the other.

3.4 WavefrontManager

The last class implemented is another bulky one. Originally this class was conceived for simulating just the medium where the perturbation travel (indeed the only istance of the class is called "aether") but, since during development more and more functionalities were added, i decided to collapse the ContextBuilder in this class, changing the structure a bit, but maintaining a better logical consistency. The Manager, after the initialization adopts two scheduled methods to check the execution of the simulation.

The first method (*propagatePerturbation*) acts as a "time aligner", thus is called with the `ScheduleParameters.LAST_PRIORITY` priority parameter. The execution of each Relay is obviously sequential, even tho it should be parallel. Using this method, combined with the method named *generatePerturbation* the Manager can collect all the Perturbation generated during each tick from all the Relays, and dispatch them (using the method from the Relay called *addPerturbation*) after every execution. The dispatch is still sequential, but it is perceived as parallel.

The second method (*run*) has the task of handling the population of Relays and the execution boundaries, therefore it's priority parameter has been set as `ScheduleParameters.FIRST_PRIORITY`. When the total tick number (the parameter that determines the execution time) is reached, the Manager commands every relay to stop sending messages, in order to let all the *bags* to empty and the frontier to synchronize. Then, when the frontiers are synchronized (as defined above in the Relay class) the execution terminates, printing the frontiers of each still active Relay. I chose to keep a generic case for the spawning/leaving of the Relays. No Relays can rejoin after they left, since they would have to reset all their frontiers, which is the same as creating a new Relay. On the same principle, when a Relay leaves the network, no procedure is issued. In this way, a Relay leaving the network can also simulate a crash, demonstrating that the algorithm is resilient.

4 Building/running instructions

To run the simulator 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 `BroadcastAppendOnlyLogs model`. By default some parameters are set in the parameters tab, as well as the *DataLoader* and the *Display*. In the same window, the *Data Set* and the *Chart* on the **redundancy evaluation** can be found, but they will be discussed in the next section.

5 Simulator Evaluation and Analysis

To evaluate the protocol I focused the analysis on the redundancy generated by the algorithm, in terms of bytes processed and byte that a Relay received (hypothetically busying the channel when not needed). I am not a great fan of the traditional communication protocols, as they are typically unicast based and many inefficiencies are well known, but most of the broadcast protocol are carried out through a great redundancy of data. Actually, bandwidth is still one of the bottleneck on the development of networks, so i spent most of my efforts on understanding if having this redundant architecture was it worth or not.

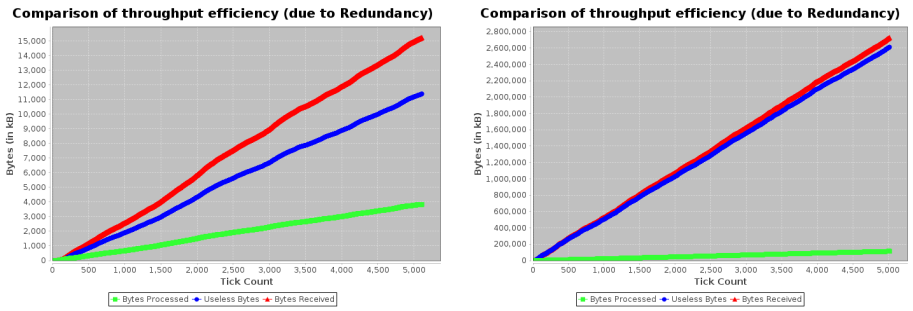

Figure 1: A test run with 5000 total ticks and a 0.05 probability of generating a perturbation and a fixed number of elements with a variable delay. The graph on the left is executed with 5 relays, while the one on the left with 25

In Fig. 1, where the number of Relays is static it is easy to see that the number of bytes processed is linear on the number of the Relays. When generating of forwarding a Perturbation obviously the number of receiver doesn't affect the system (Only one Perturbation is generated whether the is one Relay listening or there are thousands). but on the reception, this system leads to **too many bytes wasted** (received and dropped, since they were already processed). For this reason, having a more fragmented network can help reducing the jamming.

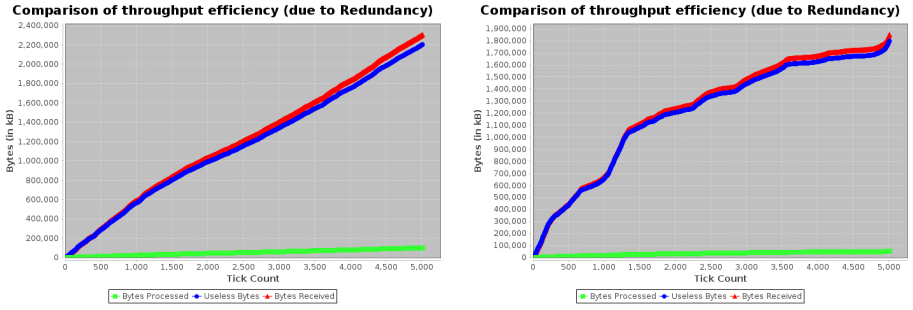

Figure 2: A test run with 5000 total ticks and a 0.05 probability of generating a perturbation and a variable number of elements (but at least 10) with a variable delay. The graph on the left is the one with a low mobility (0.001 spawn/leave probability), the graph on the right has instead an high mobility (0.1 spawn/leave probability)

In Fig. 2, the number of Relays is variable. We want to always keep at least 10 Relays informed coherently about the global situation. On a first sight, even if the *high mobility graph* is a little bit messier than the *low mobility one* (which instead sticks pretty fairly to the trend presented above), they both produce a similar quantity of bytes (∼2.2GiB low, ∼1.85GiB high). However, the high mobility one other than producing an unstable quantity of "data per tick" also delivers lot less useful bytes. the data processed indeed amounts to ∼107MiB for the low mobility and ∼51MiB for the high mobility one. This is due to the high generation rate of Relays (a newly generated Relay can't instantly produces a message) and to the perturbation generation rate being lower than the spawn/leave ones. In this situation, many relays could be destroyed without having generated any message, but potentially having received some.

The systems presented above, present and efficiency of the low mobility which is almost doubled compared to the high mobility one (∼same byte produced, ∼half bytes useful). For this reason, a stable network is preferable over a more mobile one.

6 Conclusions

Implementing this algorithm gave me a deeper insight on a true broadcast protocol, which i had never had the chance to explore. I have still room to improve the finesse of the simulator, implementing for example Relay III, which I am already doing. Unluckly, due to some issues, i decided to not include it on this delivery. The efficiency provided by this algorithm, especially in small networks is really high, provided a sufficient bandwith and memory. Those factors are indeed the biggest bottleneck in the solution presented by the writer of the paper.

The principle that i found more interesting is the disruption of the well consolidated unicast model in favour of a broadcast one, tackled and solved by the point of view of a data structure and not of the communication consistency.

References

[1] 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.