Distributed Systems 2

First Project report

Luca Fregolon, 211511
luca.fregolon@studenti.unitn.it

Alessandro Sartori, 215062
alessandro.sartori-1@studenti.unitn.it

I. Introduction

The following work aims at implementing and simulating the protocol proposed in [1], in order to evaluate its performance and reliability through the extraction of several metrics.
The chosen platform for this project is Repast Simphony, an agent-based simulation framework for Java.

II. Protocol

The protocol that we implemented, which is described in [1], is based on a broadcast-only communication model (we will show that it is possible to implement unicast and multicast delivery based on broadcast) and on a specific data structure, which are the append-only logs. The cited paper shows, in an incremental way, various protocols, which solves specific problems. The protocol introduces solitary waves as an abstraction to the real message delivery (which can be cabled or wireless without distinctions), which are based on the concept of solitons, which are particle waves which travel through space without leaving any disturbance behind them.

III. Installation and Execution

The simulation can both be compiled and run in Eclipse, with Repast Symphony or executed by installing the jar file, see the *README.md* in the main folder for more details

IV. Architecture of the Simulation

A. Simulation Parameters

Our model, upon initialization of the environment, takes account of the following tunable parameters:

- `sym_type` is an integer and sets the "increment" of the project to use:
  1) Nodes implement what is referred to as "Relay 1" in [1], i.e. a static network communicating through "frontiers".
  2) Nodes instantiated as "Relay 2" make use of a bag variable in order to handle out-of order perturbations and correctly reconstructing the append-only log. This implementation solves the problem of dynamic network, that is nodes entering and leaving at any time
  3) In simulation type 3, a retransmission mechanism is introducted and it allows to recover missing data (both because of failing or because of late entering)
  4) Type 4 is a super-set of Type 3 where nodes communicate through unicast transmissions instead of broadcast ones.
  5) Again a super-set of Type 4 but with asymmetrically encrypted point-to-point links.
  6) A super-set of Type 3 with Multicast, topic-based, message exchanges.
- `nNodes` indicates the (starting) number of nodes to generate upon initialization of the space
- `BroadcastDistance` limits the maximum distance at which a link can be established between two nodes
- `propTime` controls the delay to be multiplied with the distance to create a concept of "propagation delay"
- μ Sending time interval, which corresponds to the average interval that a node waits before generating another perturbation, regulated by a Gaussian distribution
- σ Sending time interval, which corresponds to the variance of the Gaussian distribution aforementioned

Moreover, for simulation types above or equal to 2:

- `insertInterval` is the interval at which a new node insertion event is generated
- `insertProb` controls the probability with which, at the aforementioned events, the node is actually created and added to the environment
- `failInterval` works similarly as `insertInterval` but to schedule node failures
- `failProb`, as `insertProb`, controls the probability of node failure upon a failure event

B. Simulation Builder

The construction of the simulation environment is performed in `SimBuilder.java` and consists of an initialization phase (where user parameters are read from the Repast Environment and this is set up to use a graph of nodes) and a successive construction phase where nodes are added to the graph and several types of events are scheduled to happen.
The insertion of nodes is handled by a method which takes account of the `sym_type` parameter and chooses the type of Relay to instantiate accordingly. Lastly, a scheduling section generates at regular intervals method calls to insert new nodes and set others to a failed state.

C. Relays Hierarchy

Given that relays share many properties and functionalities, their different types are implemented as child classes of `Relay1`, therefore taking advantage of Java's polymorphism to keep entities organized and decoupled:

- `Relay1`: this entity provides the basic structure of a relay to all the sub-classes: essential instance variables like the node ID, an `onSense` method, and a `forward` procedure to propagate a perturbation to all neighbors. In this type of relay, the logic of global solitary waves is enforced by keeping track of the next expected reference for each known source. Upon sensing of a perturbation, its reference is checked against the expected one and it is either re-propagated or discarded, in order not to have it processed more than once per node.
- `Relay2`: given the introduction of a dynamic network topology, a sequence of perturbations may encounter different paths and reach their destination out of order. To handle this possibility, this node introduces a `bag` variable where perturbations are temporarily kept until the frontier is ready to accommodate them.
- `Relay3`: these relays, in addition to arbitrary joins can handle temporary link or node failure by means of Automatic Retransmission Requests (ARQs). Essentially, every node periodically broadcasts its next expected perturbation ID for each known source, and if a neighbor has knowledge of more recent messages, it will retransmit this data in an attempt to extend the requester's log.
- `Relay3Unicast`: this child class of `Relay3` implements unicast point-to-point communications by assigning to each node a unique identifier, used in turn by themselves to tag messages with the intended destination. These messages propagate to every member of the network (as per protocol description), but are filtered out by each node according to its ID.
- `Relay3UnicastEncryption`: in order to improve the provided privacy of the previous relays, this class implements asymmetric RSA encryption on each unicast transmission. Receiving nodes can check if the message was intended for them by trying to decrypt it with their private key. If it was directed to them, it will have been encrypted with their public key and the decryption will succeed, otherwise the procedure will throw an exception and the message will be ignored.
- `Relay3Multicast`: the multicast version of relays type 3 differs from the unicast one on the semantics of the IDs. Instead of only using destination IDs, a node can direct a message to entire multicast groups by using a "group ID" o a "topic". As for the unicast case, nodes will read this destination information and decide whether to process or discard the message by comparing the intended target with their ID or subscribed topics/groups.

D. Edges and Bandwidth Control

At first, our implementation of message exchanges consisted in invoking a receiver method on the destination node from the source one, passing the needed data as a parameter. This approach, however, turned out to be too little effective in terms of metadata collection and performance control. Consequently, after some tests, we decided to instead take advantage of an existing "link" entity: the graph's edges. The `RepastEdge` class was extended to `CustomEdge` in order to better represent the type of link to be simulated and accommodate the needed data. Such structure consists in a queue of transiting perturbations, limited by a maximum instantaneous quantity representing a link's maximum bandwidth. Moreover, this queue allows both the collection of statistics on the network load by means of its size, and the real-time visualization of a link's load through its color.

E. Perturbations and ARQs

Perturbations are represented through a class that carries 3 pieces of information: its source, its reference, and its value. Additionally, but only for analysis purposes, perturbations carry the tick count at which they were created. This allows the measurement of average latency to be carried out fairly easily, by just comparing the reception time with the creation time. ARQs are a special

instance of Perturbations used by newly-joined nodes to recover missing information. ARQs are periodically broadcast by everyone (but only to the neighbors) to announce their next expected reference for each known source. If a neighbor has knowledge of a more recent reference than that announced for a given source, it will retransmit the perturbation to allow the requester to update its log to match the latest data. `ARQReply` is again an implementation choice to only ease the manipulation of statistics: specifically, if a node joins after some time, the perturbations it will receive in response to its ARQs will carry a generation time that will result in extremely high registered latencies, even though this is not correct. Therefore, perturbations resulting from ARQ requests are sent as ARQReplies so that their generation time is not taken in account when calculating the link latency.

F. Unicast and Multicast Transmissions

Relays can operate in unicast or multicast mode by tagging the perturbation with a destination ID. This ID can either be a target node's unique identifier, or a multicast topic ID. Upon reception, nodes will retransmit the perturbation as usual in order for it to reach the whole network, but will only process the message if they are (one of) the intended targets.

G. Privacy-Preserving Communications

Since unicast communications provide no privacy guarantees (given that messages eventually reach every active node), encryption can be employed to keep content obfuscated to all who do not possess the key. In our case, encryption is performed with the asymmetric algorithm RSA, where each node generates at initialization time a pair of keys, a public and a private one. The public one is made available to all other peers, which will use this key to encrypt anything intended to be only read by the owner. Receiving nodes, in turn, will not know if they are the intended targets, and will therefore try to decrypt the data with their private key. If this succeeds, it means that their public key was used during encryption and that they are the correct destination. If, on the other hand, the decryption fails, the message will be repropagated but not processed.

V. Results

A. Statistics Collection

In order to collect the data, we created a data set in Repast which, at defined intervals, collects latency and dropped perturbations measures from the nodes. The latency showed in the plots is an average measure and is not precise, but it is used to show the behavior on the real time plots. In order to have more precise measures we created a *Logger* class, which has static methods which can be called to save data on some csv files. This is useful because it allows us to collect data based on events rather than on time. The collected statistics are:

- Received Perturbations (only the ones accepted by the Relay)
- Total Received Perturbations
- Repast Data Set as File Sink

The first one is used to compute the statistics about the latency, as we measure latency based on the difference between the creation of the Perturbation and the acquisition by the Node. The replies to the ARQ are not computed in this statistic. The second one is used to compute how many times a node receives the same perturbation. The third one is used mainly to show the drop rate of the network.

B. Default Parameters

Parameter Name	Value
Insertion Interval of Nodes	5000
max # of Perturbation that can be queued	1000
Probability of a node to fail	0.05
Failure Interval	5000
σ Sending Time Interval	200
μ Sending Time Interval	2000
Probability to drop a Perturbation	0
Propagation Time	16
(Starting) Number of Nodes	30
Broadcast Distance	15
Insertion Probability of a Node	0.1
max # of Perturbation per Link	10000

These are the parameters used to compute the baseline latency for all the types of simulations, for the sake of brevity we report only the most relevant ones. The bandwidth of the edges (the max number of perturbation per link) is excessively large, in order for the perturbation not to be queued. In Figure 1 we can see the latency for Relay III with the default parameters (i.e. over provisioning of bandwidth). The plot shows a quite constant behavior of the latency. The drop rate is 0, since both the capacity of the link and of the queue are large. Figure 2 shows the average number of times a node receives the same Perturbation. This metric can be used to show that uninformed broadcast leads to a large redundancy in information dissemination. Even if it may seem bad that a nodes receives multiple times the same information, this somehow shows that the protocol is quite robust, since if we imagine that some Perturbations are lost, the nodes have an high chance to receive the

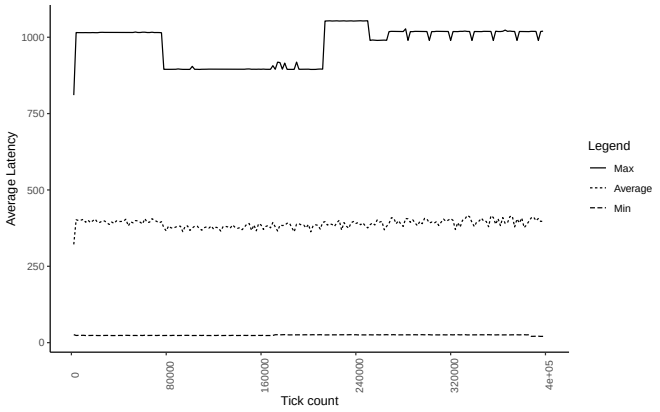

Fig. 1. Latency of Relay III with over provisioning

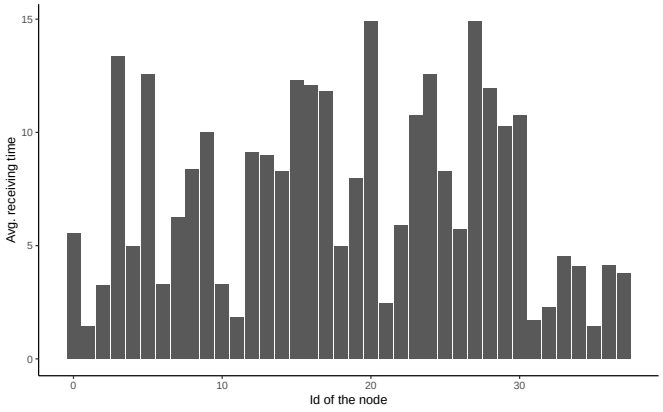

Fig. 2. Relay III: average number of times a node receives the same Perturbation

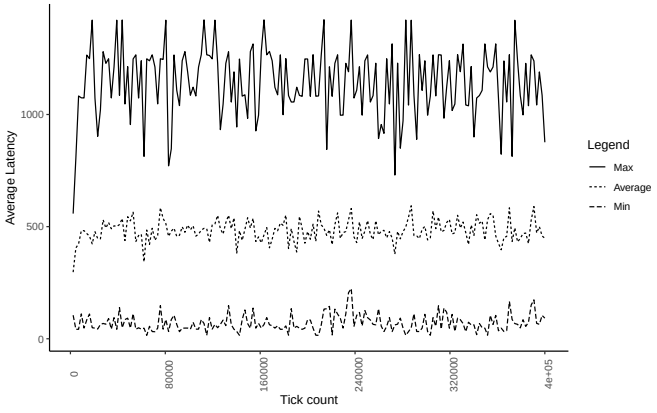

Fig. 3. Unicast model, over provisioning of bandwidth

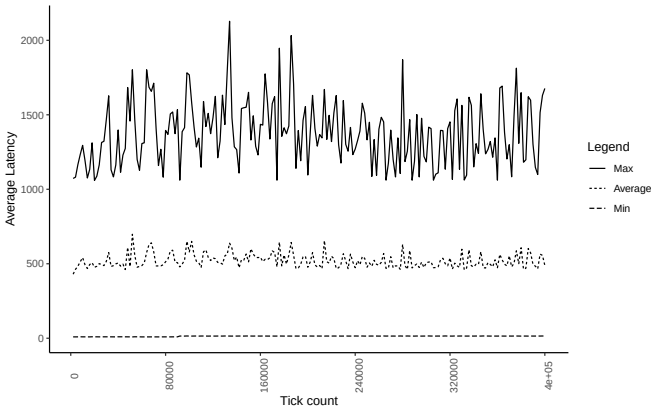

Fig. 4. Relay III with reduced bandwidth (12 Perturbations per edge)

Perturbation anyway from another path, without the need to request it. The behavior is quite the same for the first three simulations, while is a bit different for the Unicast simulation (Figure 3), since the latency is now computed only by the intended receiver. We can see that the behavior is much more variant, but it is quite constant anyway. Things start to change when we reduce the link capacity. Perturbations start to be queued and so the latency increases. In Figure 4 we can see that the latency start to increase (in particular the max latency) limiting the link capacity to 12 perturbations per time. The average latency, anyway, continue to show a constant behavior, even if the latency is a bit higher than before. In Figure 5 we can see a quite degraded execution, with the link capacity reduced to 5 Perturbations per edge. The latency increases a lot and a Perturbation can take almost 10 times the baseline latency to be delivered. The drop rate starts to increase too, as showed in Figure 6. Since the drop rate parameter value was set at 0, the drop rate

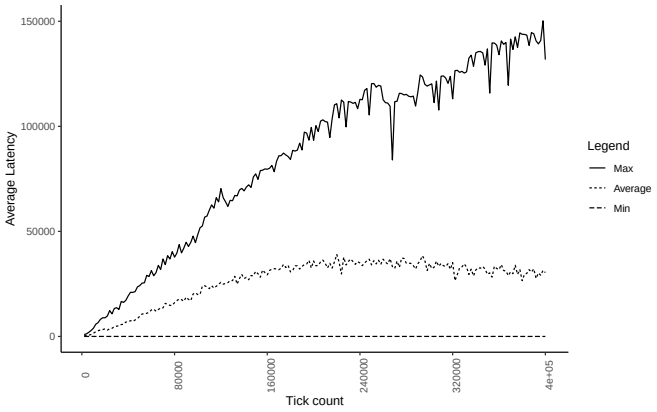

Fig. 5. Relay III with reduced bandwidth (5 Perturbations per edge)

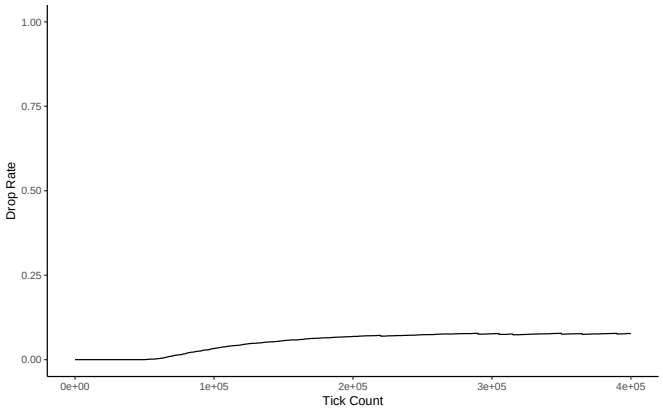

Fig. 6. Caption

is caused by the network not being able to deliver and to store in the wait queue all the Perturbations created.

VI. Conclusions

The model we simulated resulted quite robust. In the simulations we made all the nodes were able to receive all the Perturbations (even the ones that joined the network later) thanks to the redundancy of the protocol and to the re-transmission mechanism. The model was tested also with an higher number of nodes and with different load settings, showing good performances also with a number of nodes more than three times the default parameter. It has to be noticed that the plot in Fig 5 is a very extreme case, since it allows only 5 Perturbations per link (if we think about a perturbation as a packet of 500B it is as low as 3KiB).
The model is then well suited for applications with low transmission sizes/rates but which need a robust information dissemination.

References

[1] Christian F. Tschudin *A Broadcast-Only Communication Model Based on Replicated Append-Only Logs*. ACM SIGCOMM Computer Communication Review