Distributed Systems 2 - Project I
An implementation of a Broadcast-Only Communication Model Based on Replicated Append-Only Logs

Massimo Mengarda
214290

Matteo Contrini
215037

November 2020

1 Introduction

In this report we introduce an implementation of a broadcast-only communication model based on the concept of perturbations, as described in the paper *A Broadcast-Only Communication Model Based on Replicated Append-Only Logs* [1]. The work also includes an evaluation part which consists in the analysis of the performance of the model.

The implementation was done by using Repast Simphony, a simulator for agent-based communication models, which has allowed us to represent the model graphically and to easily gather statistics during runtime.

In the following sections, we summarise how the broadcast-only model works at a high level, we then explain the main architectural choices of the implementation and finally we evaluate the performance of the model.

2 The broadcast-only model

The broadcast-only communication model described by Christian F. Tschudin [1] is based on the idea of propagation of perturbations, typical of how information is transmitted in the analog world but not very frequently found in digital applications.

This is what makes this model fundamentally different: all communications happen as a broadcast, from the application perspective. This means that there is no concept of link and no destination of a message. Unless stopped, perturbations that disseminate information through waves eventually reach all the entities in the system.

The presented model also makes a distinction between local and global waves, where local waves are sensed by relays that then "forward" the perturbations to other nodes.

Different variations of the model can be conceived, but the one we've taken into consideration is the one for dynamic networks with recovery from losses. Each relay keeps a log of perturbations that have been received, allowing to resend perturbations that other relays might have lost for different reasons.

The Automatic Repeat reQuest (ARQ) mechanism to recover from losses is straightforward to implement: a relay periodically asks for the next expected perturbation for each source, and waits for "replies".

On top of this basic broadcast-based model we can build abstractions to achieve more sophisticated communication models. For example:

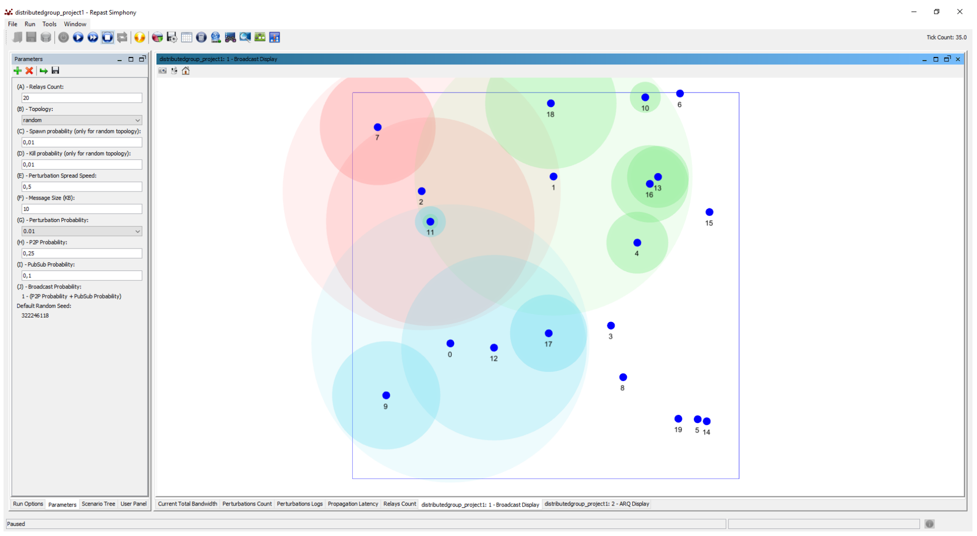
- point-to-point communication can be achieved by sending the destination relay ID together with the message;
- publish/subscribe or in general group communication can be achieved in a similar way by including a "topic" ID;
- privacy can be achieved with public key cryptography, so that only the intended destination can decrypt the message with a private key.

3 Architecture and implementation

In this section we introduce the main architectural choices that we made and present the most important aspects of our implementation.

3.1 General idea

In our implementation, we tried to represent as closely as possible the perturbation concept presented above. Perturbations are in fact graphically represented as circles that grow over time, until they reach a certain radius and vanish. This is a simplification with respect to the paper, and we made this choice primarily because it doesn't affect the broadcast behaviour of the model while allowing us to better understand what's going on with the simulation while it is running.



(A) - Relays Count: 20
(B) - Topology: random
(C) - Spawn probability (only for random topology): 0,01
(D) - Kill probability (only for random topology): 0,01
(E) - Perturbation Spread Speed: 0,5
(F) - Message Size (KB): 10
(G) - Perturbation Probability: 0.01
(H) - P2P Probability: 0,25
(I) - PubSub Probability: 0,1
(J) - Broadcast Probability:
1 - (P2P Probability + PubSub Probability)
Default Random Seed:
322246118

Figure 1: An example of the simulation with 20 relays.

We implemented pure broadcast communication, together with the Automatic Repeat reQuest (ARQ) mechanism, but also added the capability to exchange Point-to-Point (P2P) messages and the concept of topics to allow Publish-Subscribe communications. P2P messages are encrypted with *RSA*, an asymmetric encryption algorithm, thus making sure that only the actual sender and the receiver can read the contents of the message.

We implemented multiple topologies that the user can choose from, in order to analyze the behaviour of the model with different arrangements of relays. When the topology is "random", we also made the set of relays dynamic, meaning that relays are spawned or killed with the given probabilities.

3.2 `Relays` and `Perturbations`

At the core of our implementation lie the two classes that represent the two main types of agents in the model: the `Relay` and the `Perturbation`.

Relays are placed in the model when the simulation is started, according to the number of relays and the topology that were chosen with Repast parameters. The spawning/killing of relays is implemented through the `RelaysManager` agent, which makes the simulation more dynamic by adding and removing relays randomly according to a pair of probabilities that can be set through parameters.

Relays periodically send new perturbations with a given probability by using the `@ScheduledMethod` Repast feature. At each invocation the code evaluates whether it should create a new perturbation by taking into consideration the *Perturbation probability* parameter (`createAndSendPerturbation` method). The creation of a perturbation is also largely driven by probabilities: three Repast parameters are used to determine whether the new perturbation should be a classic broadcast, a P2P message, or a PubSub message, while the *Message Size* parameter determines the length of the content of the message that will be sent.

In the case of Point-To-Point communication, the destination relay ID is also randomly chosen, and the content of the message is encrypted with *RSA* by using the destination's public key. In the case of Publish-Subscribe communication, it's the destination topic that is chosen randomly, but messages are not encrypted.

For the ARQ mechanism to work properly, perturbations are inserted in the log before being broadcasted. The log consists of a dictionary (Java `Map`) that stores for each source the list of perturbations that were sent (by the current relay) or received from other relays.

In our implementation, broadcasting a perturbation simply means placing the perturbation in the space, in the same position as the relay that produced it. The perturbation will then expand by itself and be sensed by other relays, as we will see in the following section.

3.3 Graphical representation of perturbations

To represent perturbations graphically, we created a custom Repast 2D style with the `PerturbationView` class. This class is set as the "Style Class" in the settings of the display.

The class `PerturbationView` implements the `repast.simphony.visualizationOGL2D.StyleOGL2D` interface, which allows us to create a custom graphical representation for an agent.

In our case, we want to represent perturbations as circles that grow over time, so in the style implementation we build a circle by using the `ShapeFactory2D.createCircle()` method (line 26 in `PerturbationView`).

This method is automatically called at every tick to render the agent, so we can simply increment the radius by a small amount each time to produce an effect of wave propagation. This is in fact what can be seen in the code of the `Perturbation` class, where the `propagate()` method is called by Repast at every tick and increments the radius. When the radius exceeds a maximum radius, which we set to 25 units, it is removed from the view.

We chose 25 units as the maximum perturbation radius because the size of the display is 50 x 50, therefore a perturbation should cover most of the area. However, if a low number of relays is chosen, there's the possibility that some broadcasts will not reach all the relays, especially with the random topology. This can be easily fixed by increasing the maximum perturbation radius. In our tests we made sure to use an appropriate number of relays in order to avoid the issue, while making the simulation a bit clearer to understand and more pleasant to watch.

The custom style also allows to specify a color for the perturbation, by providing a `java.awt.Color` instance. We use this feature to give a different color to different types of perturbations (broadcast, P2P, etc.) (Figure 2).

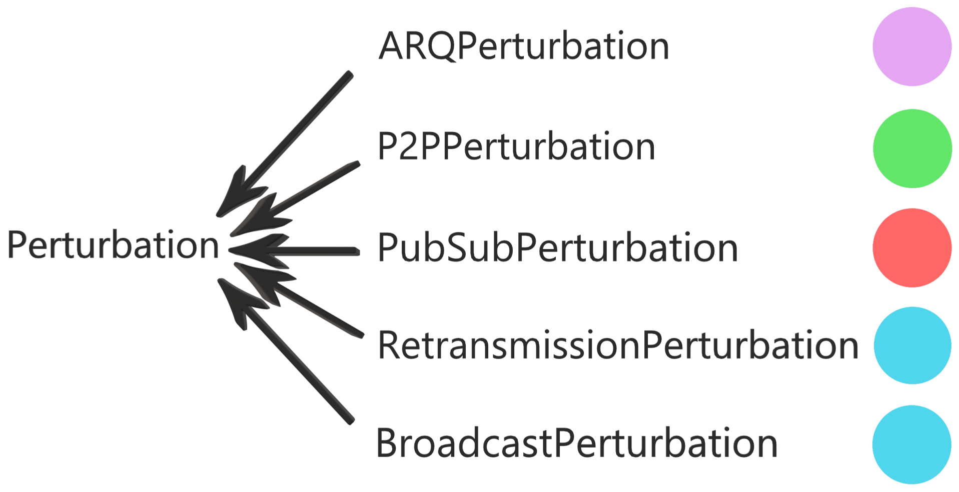


Figure 2: Types of perturbations and their representations.

We also specify an opacity value that we calculate depending on the radius of the perturbation. This has the effect of fading out the circle gradually while it is approaching its maximum radius.

In particular, the opacity when the perturbation has a given *Radius* can be calculated as:

$$\alpha = InitialOpacity - (\frac{Radius}{MaxRadius} \cdot InitialOpacity) \tag{1}$$

And by collecting *InitialOpacity* we get to the formula that can be found at lines 36-37 of the `PerturbationView`:

$$\alpha = InitialOpacity \cdot (1 - \frac{Radius}{MaxRadius}) \tag{2}$$

In the implementation we set $InitialOpacity$ to a constant value of 90 (percent), while the $MaxRadius$ for a perturbation is 25 units as mentioned above. Both the initial opacity and the maximum radius values can be changed in the corresponding fields in the source code.

3.4 Sensing of the perturbations

Whenever the wavefront of a perturbation reaches a relay, we say that the relay senses the perturbation. When it happens, the relay should process the message carried by the perturbation and act accordingly.

Since in our implementation the perturbation is represented by a circle with a growing radius, we need to compute periodically for each relay the distance between the "border" of the perturbations and the relay, to detect whether a wavefront has spread enough to touch the relay.

This is done with the `onPerturbation` method of the `Relay` agent class, which is called by Repast depending on the `@Watch` parameters that were defined. In particular, we made it so that the method gets called whenever a `Perturbation` instance is close enough to be able to "touch" the relay. Within the method we then perform the actual check on whether the perturbation has reached (or went beyond) the relay.

In practice, the `@Watch` parameters are set as follows:

- `watcheeClassName` is set to the full name of the `Perturbation` class;
- `watcheeFieldNames` is set to `radius` so that the method gets called whenever the perturbation radius changes;
- `query` is set to `within MAX_PERTURBATION_RADIUS` so that only perturbations whose wavefronts can reach the relay are taken into consideration;
- `whenToTrigger` is set to `WatcherTriggerSchedule.IMMEDIATE`.

Inside the `onPerturbation` method, the distance between the perturbation and the current relay is computed, and if the result is equal to or less than zero it means that the perturbation has been sensed by the relay.

Something to note is that this check will give a positive result many times even after the perturbation went beyond the relay, so we need a way to memorize which relays have already sensed and processed the perturbation. This is done through the `processedInRelays` set in the `Perturbation` class.

Finally, the perturbation can be handled by the relay. For ARQ requests, we will see in the next section, while for normal messages the handling depends on the type of communication. For example, in the case of P2P perturbations the message is decrypted to find out if the relay is actually the destination.

3.5 Recovering lost perturbations (ARQ)

Since the maximum radius of a perturbation is limited and relays are dynamic (i.e. they can disappear at any moment), perturbations could be lost and not reach all the relays.

The *Automatic Repeat reQuest* mechanism implemented in each relay solves this problem by periodically asking all relays for the next expected perturbation for each source, according to the log.

This mechanism was implemented so that ARQ requests are sent every 200 ticks, and analysis in section 4 will show how this tends to produce quite a high amount of perturbations.

When a relay receives an ARQ request, it checks whether it has the requested perturbation in the logs and broadcasts it if present. Eventually, the perturbation will reach whoever asked for it.

3.6 Encryption in P2P communication

To preserve the confidentiality of point-to-point communication, encryption was implemented with the asymmetric encryption algorithm *RSA*.

Each relay is associated with a private key that is kept private, while every relay is provided with the public keys of all the other relays. To simplify the implementation, we assumed that all the relays always have immediate access to all the public keys instead of implementing a key exchange mechanism.

Since private keys are only known by the corresponding relay, decryption will succeed only on the destination relay for a particular perturbation, implicitly providing a way to specify the destination of a message by using *RSA* keys.

It should be noted that this is a naive implementation, therefore, for example, it doesn't handle sender authentication and could also be vulnerable to replay attacks.

3.7 Runtime parameters

The simulation of the model can be tuned by adjusting several parameters through the *Parameters* tab of Repast GUI.

Here's a summary of the available parameters:

- *Relays count*: the initial number of relays that are created in the space. This should be set before starting the simulation, since changing the value during the run has no effect;
- *Topology*: the arrangement of relays in the space. Possible values are *random*, *ring*, *star* (ring with a relay in the middle), *star+* (two rings with a relay in the middle). Changing this value during the simulation has no effect;
- *Spawn probability*: the probability that a new relay will be added to the space every 10 ticks, only when the topology is *random*. This value can be changed during the simulation;
- *Kill probability*: the probability that a random relay will be removed from the space every 10 ticks, only when topology is *random*. This value can be changed during the simulation;
- *Perturbation spread speed*: defines how much the perturbation circle should grow at each tick, in units. This value can be changed during the simulation;
- *Message size*: the average message size, used to estimate the total amount of data travelling at a given time. This value can be changed during the simulation;
- *Perturbation probability*: the probability that a new perturbation will be created every 10 ticks for each relay. This value can be changed during the simulation;
- *P2P probability*: the probability that a new perturbation will be of type point-to-point. This value can be changed during the simulation;
- *PubSub probability*: the probability that a new perturbation will be of type publish/subscribe. This value can be changed during the simulation;
- *Broadcast probability*: the probability that a new perturbation will be of type broadcast. Computed as $1 - (P2PProbability + PubSubProbability)$.

3.8 Displays

We created 2 displays in the Repast GUI:

- *Broadcast Display*, that shows all relays and perturbations except for ARQ requests and retransmissions;
- *ARQ Display*, that only shows relays, ARQ requests and retransmissions.

We decided to split the representation of the perturbations because every 200 ticks ARQ requests take place and the many retransmissions that are usually generated cover the entire display.

We also made sure to use a grid and a space with a `PointTranslator` set to `InfiniteBorders` so that perturbations don't wrap around borders.

Finally, we tried to make the simulation easier to understand by introducing a *Repast Network* with edges between relays, but a bug in Repast appeared to leak memory so we couldn't use this feature.

4 Analysis

The implementation of the model was tested by trying many combinations of the runtime parameters to observe how the model behaves in different conditions and to find out if there are any critical configurations that should be avoided.

To analyse the performance of the model we relied on Repast features, in particular on data sources and charts. We also created some custom data sources to collect some metrics that couldn't otherwise be computed with built-in metrics.

4.1 Relays Count

This chart represents the current number of relays in the simulation. This is usually constant unless the random topology is chosen and the *Spawn probability* and *Kill probability* parameters are set to a value greater than zero.

Figure 3 shows the chart with *Spawn probability* and *Kill probability* set to 0.1. As we would expect, the number of relays more or less remains about the same as the initial one, on average.

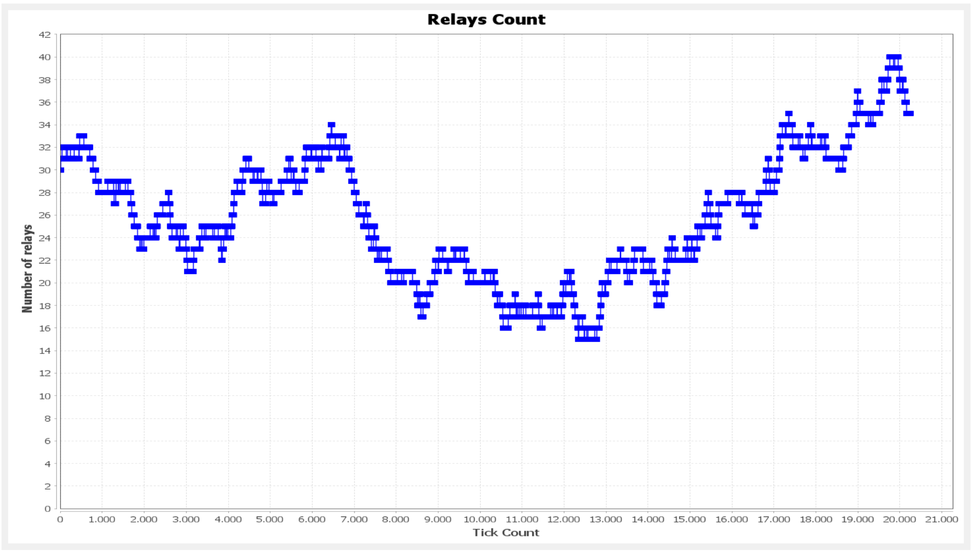


Figure 3: Relays count chart with default parameters, except for *Spawn probability* and *Kill probability* which are set to 0.1.

4.2 Propagation Latency

This chart represents the time it takes for perturbations to be delivered. The value is computed for each perturbation at delivery as follows, and then stored in a circular buffer of 50 values.

$$Latency = CurrentTick - PerturbationCreationTick \tag{3}$$

More in detail, the *Propagation Latency* chart shows the following three time series:

- *Average latency*: the average of the mean propagation latency of each relay;
- *Min average latency*: the minimum of the mean propagation latency of each relay;
- *Max average latency*: the maximum of the mean propagation latency of each relay.

These metrics are computed through three custom data sources (in particular, `LatencyMinDataSource`, `LatencyMeanDataSource`, and `LatencyMaxDataSource`), that implement the `AggregateDataSource` interface. In these data sources we compute the values by making sure to skip relays that have still to receive perturbations and shouldn't therefore be included in the computations.

As mentioned, latencies are computed whenever a perturbation is delivered to the destination, which means that in the case of a P2P perturbation only the recipient relay will compute it. This fact could lead to the wrong conclusion that a simulation with only P2P perturbations will present a higher average latency, but in reality this value will just converge slower in comparison with normal broadcast. In fact, a broadcast can be seen as many P2P communications from the same source to all the other relays, and since the propagation speed does not change with respect to the perturbation type, the *Average latency* will eventually be the same.

Similarly, changing the number of relays in the simulation does not affect the latencies, but it helps reaching a "stable" value more quickly, at the expense of making the simulation heavier to run.

Obviously, increasing the spread speed of the perturbations will lower the latency (the average latency has an inverse linear correlation with the spread speed).

Finally, some considerations can be made about the behaviour in different topologies. In the *random* topology, we observe that the *Min average latency* is lower in comparison with other topologies, probably because the random placement of relays makes it more probable to have very close relays. This does not happen with other topologies since the relays are arranged in a fixed way.

For the same reason, we observe that with the random topology the *Max average latency* is slightly higher with respect to other topologies. We find that this is the case because in the random topology relays are more scattered and thus more distant on average.

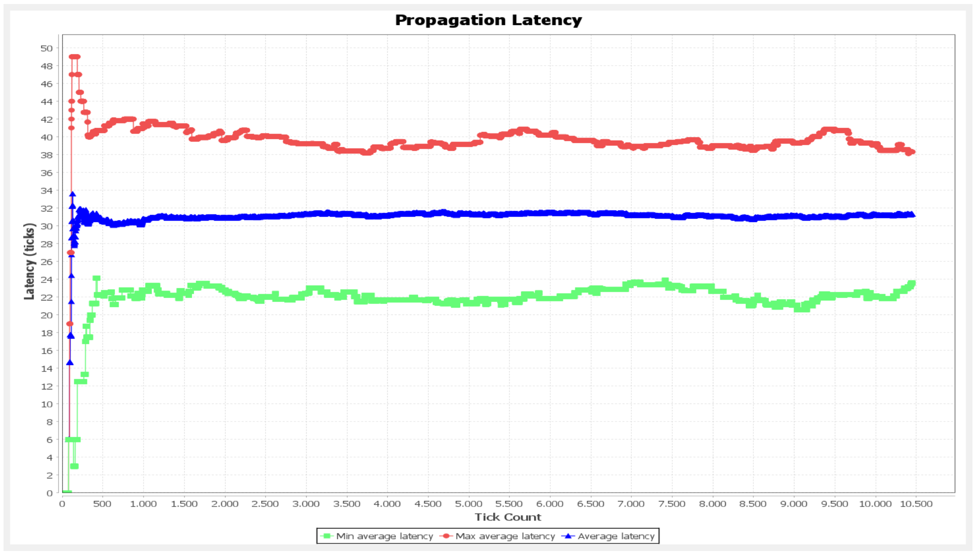


Figure 4: Propagation latency chart with 30 relays arranged in a random topology. The parameters are set to the default values, but *Spawn probability* and *Kill probability* are set to 0 to avoid “jumps” in the chart.

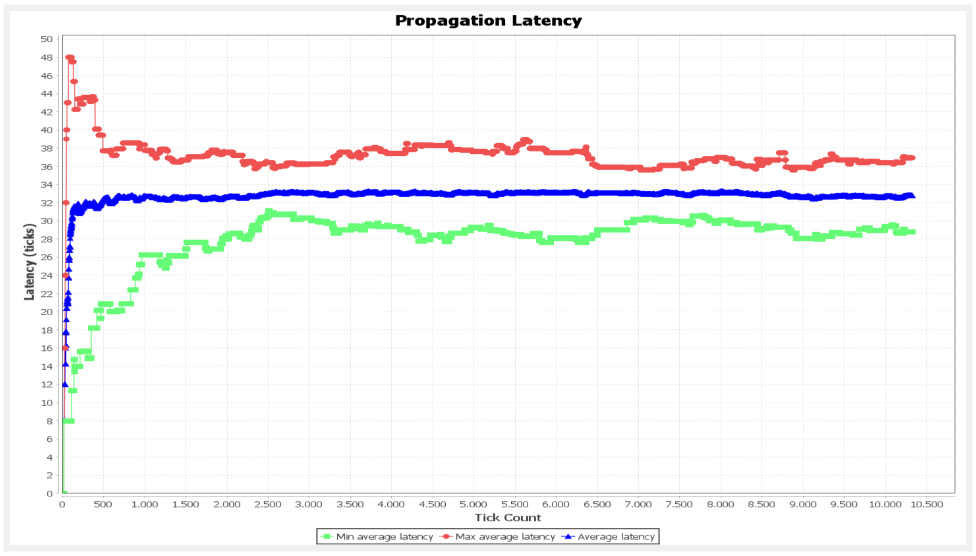


Figure 5: Propagation latency with 30 relays arranged in a *ring* topology. The chart is very similar with the *star* and *star+* topologies.

4.3 Perturbations Count

This chart represents the current number of perturbations of different types in the simulation.

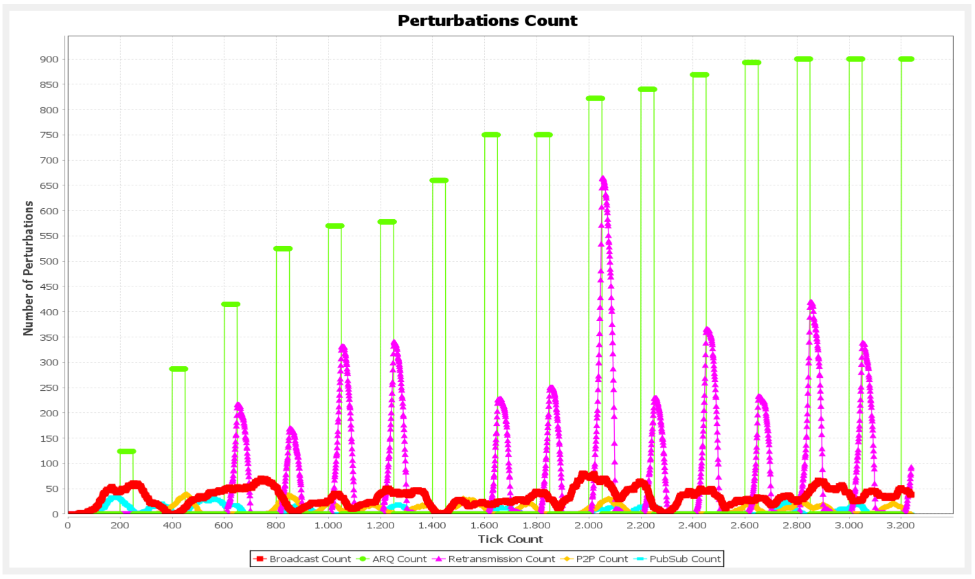


Figure 6: The number of perturbations of different types in a simulation with default parameters.

The peaks of ARQ requests and retransmissions are due to the fact that all the relays perform the requests at the same time and each relay that receives an ARQ request will reply with the requested perturbation immediately, generating a lot of traffic (Figure 6).

If we analyze the worst possible scenario, each relay will generate n ARQ requests at the same time, where n is the number of relays in the simulation. Every relay will reply to an ARQ request with a *RetransmissionPerturbation*, which leads us to the following total counts of ARQ requests and retransmissions:

$$ARQRequestsCount = RelaysCount \cdot RelaysCount \tag{4}$$

$$RetransmissionsCount = ARQRequestsCount \cdot RelaysCount \tag{5}$$

For example, if we take 30 relays, we can theoretically generate up to 2700 *RetransmissionPerturbations* at the same time. This scenario is more plausible if the *Perturbation probability* parameter is set to some high value (Figure 7).

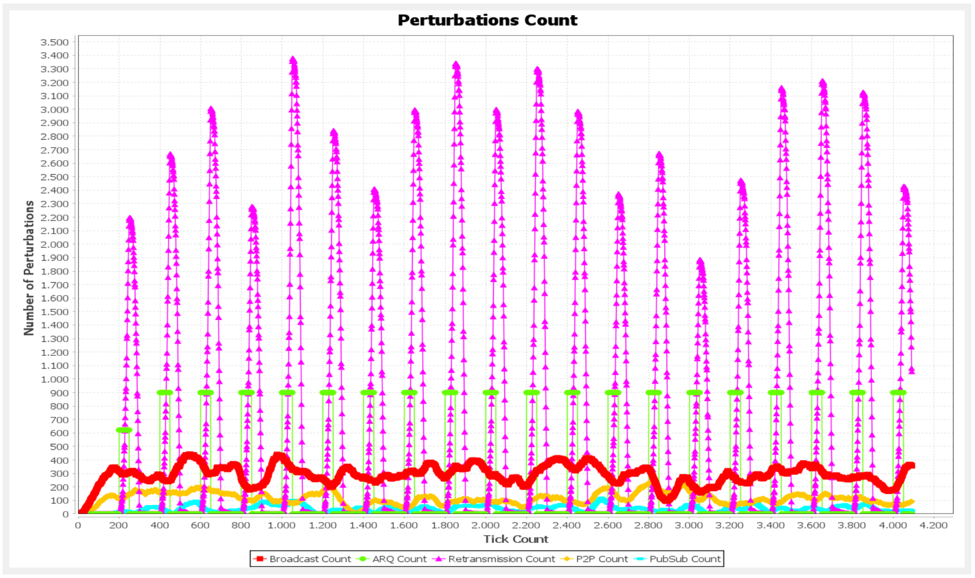


Figure 7: The number of perturbations of different types in a simulation with the Perturbation probability parameter set to 0.1

Finally, we can observe that if we increase the *Perturbation spread speed* parameter, far less retransmissions will occur. This is because more relays will receive perturbations before ARQ requests.

4.4 Perturbations Logs

After observing that so many retransmissions were happening during the simulation, we decided to compare in a chart both the number of perturbations in the logs and the number of duplicate perturbations, which correspond to the total number of retransmissions.

As mentioned before, the *Perturbation spread speed* and *Perturbation probability* parameters are inversely proportional to the number of retransmissions.

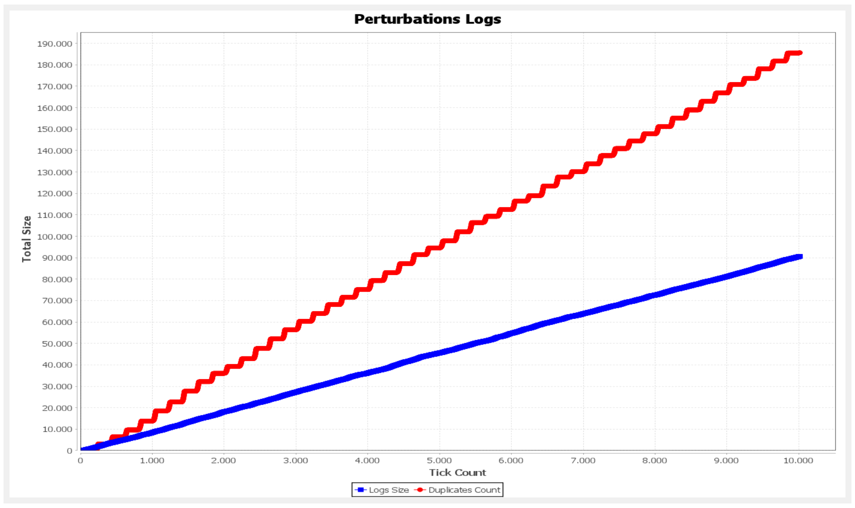


Figure 8: Chart with the total log size and the duplicates count for a simulation with 30 relays, a random topology, and a *Perturbation probability* of 0.1. It can be observed that almost immediately the number of retransmissions exceeds the total number of correctly received perturbations, while after $10k$ ticks it reaches almost $200k$ retransmissions.

4.5 Current Total Bandwidth

One fact that follows from the high number of retransmissions is that at regular intervals there's a lot of activity in the system, producing a potentially large amount of data traffic.

We estimate the amount of traffic in the system with the following formula:

$$TotalBandwidth(KB) = PerturbationsCount \cdot MessageSize \tag{6}$$

Where the *MessageSize* is a runtime parameter expressed in KB.

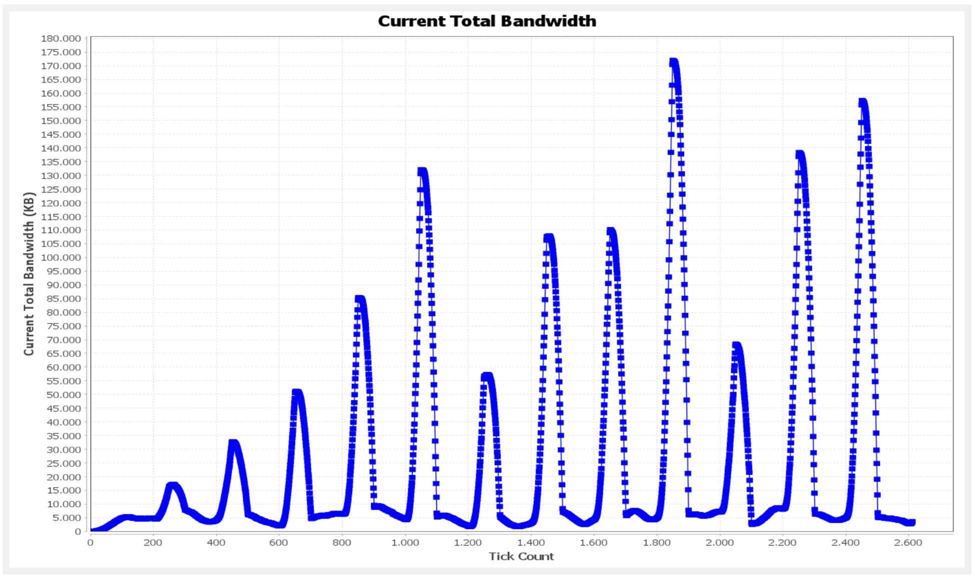


Figure 9: In this simulation with 100 relays and default parameters, we can observe peaks of almost 175 MB of traffic travelling in the system.

5 Conclusions

The implementation of the model with Repast was successful, but we found the confirmation that broadcast-only models tends to generate a lot of useless traffic.

In particular, the basic ARQ mechanism presented in the paper has the effect of generating duplicate retransmissions that could be avoided. This happens because at regular intervals each relay sends ARQ requests without having a way to know if another identical request was already sent. In the paper, a relay that receives two identical ARQ requests reply to both of them.

This situation could be improved by scattering the ARQ requests in time and therefore by reducing duplicate retransmissions.

A further problem derives from the fact that new relays can be added at any time, which initially have no knowledge of the system. Therefore, they need to wait for a perturbation by each other relay (that will be probably discarded), just to know that they exist.

After that, the new relay needs to catch up with the history of perturbations by using ARQ requests. This mechanism however is very slow because it requests only a single perturbation per source every 200 ticks, so the new relay could actually never be up-to-date.

A way to prevent this could be to implement a protocol that new relays execute to retrieve the full list of logs.

References

[1] Christian F. Tschudin. *A Broadcast-Only Communication Model Based on Replicated Append-Only Logs*. URL: `https://ccronline.sigcomm.org/wp-content/uploads/2019/05/acmdl19-295.pdf`.