DISTRIBUTED SYSTEMS 2

UNIVERSITY OF TRENTO

DEPARTMENT OF INFORMATION ENGINEERING AND COMPUTER SCIENCE

Implementention of a Broadcast-Only Communication Model Based on Replicated Append-Only Logs in Repast Symphony

Authors:
Andrea Traldi (ID: 213011)
Constantin Popa (ID: 211986)

Date: November 30, 2020

Abstract:

This document is intended to describe how the broadcast-only communication model based on replicated append-only logs[1] proposed by Christian F. Tschudin can be implemented using the Repast Symphony framework. We will start with briefly summarizing some of the key concepts of the paper and what is our understanding of the entire protocol. Then, we will discuss the architecture of our application, going deeper and deeper into details, until we get to the small technical details of the project such as the simulation parameters. Finally, we analyse the performance of the proposed protocol, presenting also our findings and suggestions of how the algorithm could be improved.

1 Protocol summary

Nowadays, the point-to-point communication is by far the most popular form of exchanging information among nodes, since it is used by the largest network in the world, the Internet. In his paper[1] instead, the author proposes a different approach, suggesting a broadcast-centric world as an alternative. What this new solution offers is the possibility to send a message without specifying a destination and it eliminates the need of having a routing algorithm, since the information keeps propagating continuously and in all the directions unless it is blocked or it reaches its maximum range.

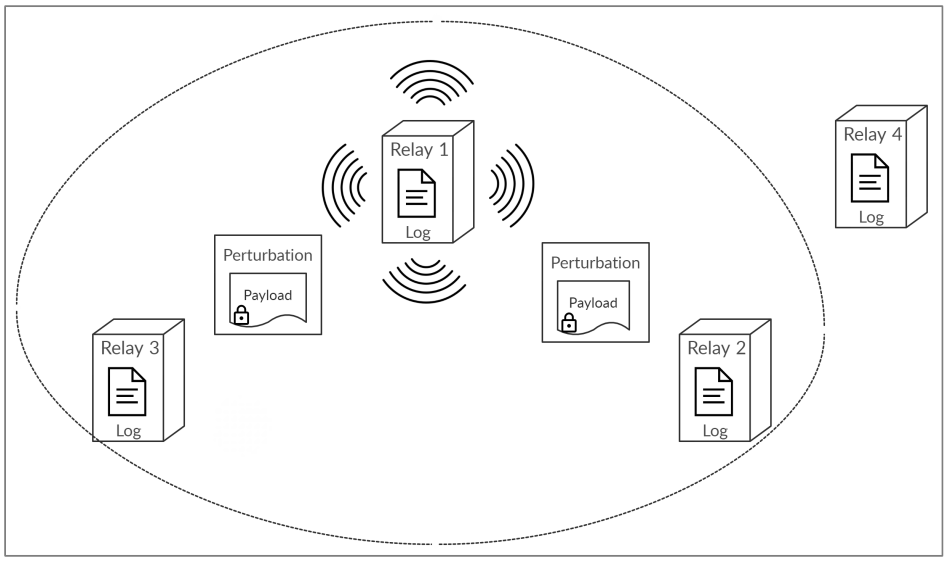

Figure 1: Example of a relay generating a perturbation with an encrypted payload

They key elements of this model can be seen in *Figure 1*, and they are the *relays*, their *logs* and the *perturbations* they generate. The first element refers to the nodes of the network, also called *observers* when they behave as passive agents which sense the pertur-

bations emitted by other relays. The perturbation is a metaphor inspired to the physics field that the author of the paper chose to refer to when talking about information dissemination. Hence, whenever a relay generates a new perturbation, it propagates in all the directions in a wave-like form and it can be sensed by the other observers that are in the same local domain broadcast (the dashed curved line in the picture) as the source.
After picking up the signal, the relays forward the perturbation so it can reach further relays. In the above example specifically, the *Relay 1* starts emitting a new perturbation which will soon propagate so *Relay 2* and *Relay 3* will sense it. *Relay 4* is too far to be reached by this perturbation; however this is not a problem since *Relay 2* will forward the perturbation, so *Relay 4* will be able to pick it up successfully this time.
In order to preserve ordering and safety, each relay stores internally the next expected perturbation reference per source. When a new wave is sensed, it will be delivered if and only if it is the exact next perturbation expected for that source; otherwise, based on the protocol variant, it gets discarded or stored in a buffer awaiting to be delivered later. There might be cases in which the next expected perturbation won't arrive, so the relays must implement an automatic retransmission mechanism in order to guarantee the liveness of the protocol. Therefore, for each source in the log, relays will broadcast at regular intervals a request to their neighbours, asking for the next expected perturbation.
The log of each relay consists of a chain of perturbations which were delivered by the relay itself. Even though the length of these logs might be different at a certain point in time, the perturbations have to be delivered and appended to the log in the same order as they were sent by the source, and processed only once by all the relays. Relays must have globally unique identifiers, but perturbations are not bound to this rule, hence each relay has an internal clock which automatically increases when a perturbation is generated. This implies that perturbations can be uniquely identified by combining the source identifier and the perturbation identifier.
In the most simple case, a perturbation is a tuple <*src, ref, val*> containing the source identifier, the perturbation identifier and a value. The payload of the perturbation doesn't always have to be a value and it can be replaced by more complex payloads in order to build more advanced communication primitives:

- plain private messages, in the form <*src, ref, <dest, msg>*>
- encrypted private messages, <*src, ref, <encrypted_payload>*>
- group communication, <*src, ref, <group_id, msg>*>
- publish/subsribe messages, <*src, ref, <topic_id, msg>*>

As can be seen, the payload is not a simple value anymore, but it now includes both meta data about the content, as well as the value that needs to be delivered.

2 System architecture

For our implementation, we have chosen the agent-based modelling as a paradigm for simulating the proposed protocol. Furthermore, the components were designed by keeping in mind best practices such as modularity and reusability, and they were grouped in the various packages based on their logical role. Let's now have a more in depth look at each of them and their relationships.

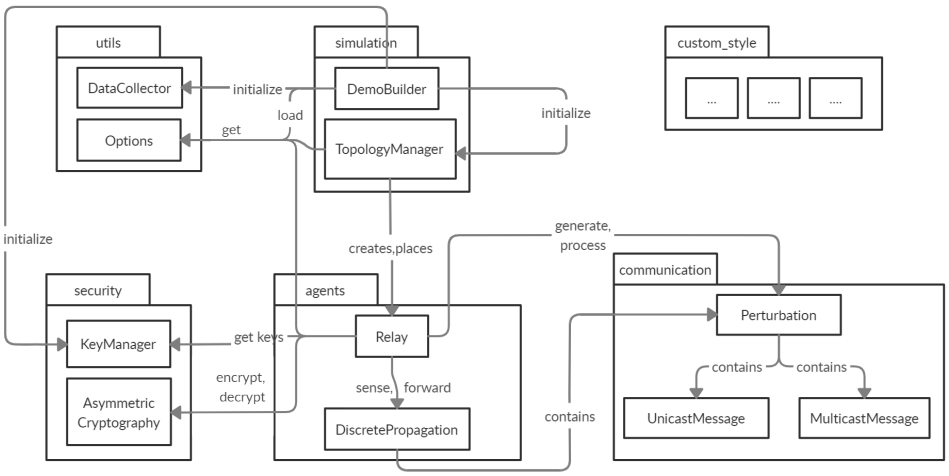

Figure 2: Components and main interactions among them

2.1 Agents

The agents of our model are the *Relay* and *DiscretePropagation* classes. The main functions of a *Relay* consist of generating and picking up perturbations, processing them and eventually forwarding them in order to reach new observers. Being an observer is a role of the Relay, hence it will be implemented as internal behavior, without the need to create a whole new agent.
The purpose of *DiscretePropagation* agent is to carry the particles that form a wave, helping the perturbation to propagate. Therefore, it will contain the perturbation and some other information related to the direction it propagates along, expressed as an angle measured in radiant, as well as the distance it has traveled w.r.t the position of the relay it was forwarded by.

2.2 Communication

Since there is no direct mapping of the perturbation wave in Java or even in Repast Symphony, there is the need to emulate this behaviour using manual mechanisms. Having many point-to-point connections among the relays would have changed too much

the idea proposed in the original protocol, hence we decided to not follow this path and create something that resembles better the concept of a broadcast, rather than simplifying it with multiple unicast messages send to the neighbors.
Therefore we have thought of abstracting this wave which propagates in all the directions and in a circular form, with a number of points that are picked from this virtual circumference. Naturally, by increasing the number of points, the shape will resemble more and more to a circle, at the cost of increasing the computational complexity of the whole simulation and decreasing the performance of the simulator itself, based on the available hardware resources. For instance, representing a circle with 16 points instead of 8 offers surely a better visual effect, but it will introduce exponential complexity w.r.t the number of items to be processed from the environment.

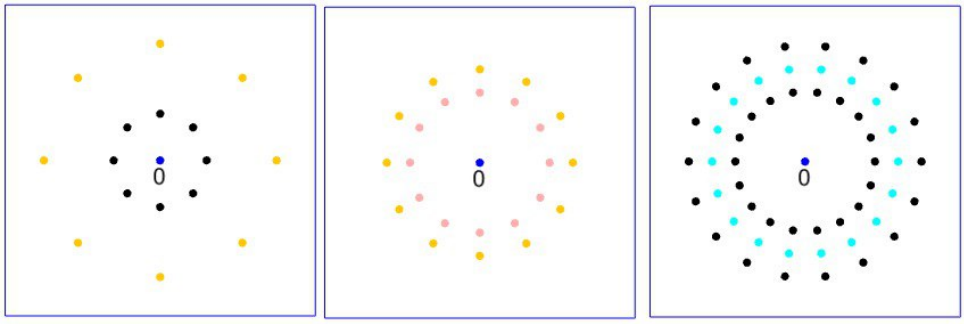

Figure 3: Comparison of perturbation waves made of 8, 12 and 16 points

The *Perturbation* class represents the concept with the same name described in the original paper, hence it will contain information about the source, an identifier for the perturbation itself and finally a payload. As we said though, we need to propagate this piece of information in multiple directions, and this is where the *DiscretePropagation* agent described earlier comes in handy.
The payload of a perturbation can be more than a simple value. The private messages were modeled with the *UnicastMessage* class. Since the group and the Publish/Subscribe messages are very similar because they are addressed to a certain group and optionally have a topic, they have been represented with the same class *MulticastMessage*, which suits both the usages.

2.3 Security and privacy

Unicast messages, also called as private messages, can be sent as plain or encrypted. In the latter case, there is the need of a cryptosystem which performs key management (generation, use, exchange, etc.), as well as encryption and decryption methods. These operations are encapsulated inside the package *security*. The class *KeyManager* generates a pair of keys, one private and one public, for each relay. Later, when a relay needs to encrypt a message, it asks the *KeyManager* for the public key of the destination. Similarly, when relays need to decrypt a message, they retrieve their private key

from the same class. *AsymmetricCryptography* component contains the actual methods for encrypting/decrypting payloads, given a payload and a key as input.

2.4 Simulation parameters and data collection

The *Utilities* package groups together the user-defined parameters for the current simulation and the methods used to collect and report statistics about its outcome. *Options* class reads these input parameters and make them available to the other components (*Relays, TopologyManager*, etc.). *DataCollector* is responsible for collecting information about the simulation at runtime and output them to a permanent file, which can later be used for generating graphs.

2.5 Simulation environment and customization

The components inside the main package (*simulation*), are responsible for boostraping the simulation and coordinating the other components. The *DemoBuilder* is the entry point of the simulation and it starts up the *KeyManager* the *TopologyManager*, as well as the components in the *utils* package. *TopologyManager* encapsulates all the logic related to the creation of the topology and its maintenance. For our implementation we have chosen to support 4 topology types: random, extended star, line and ring.

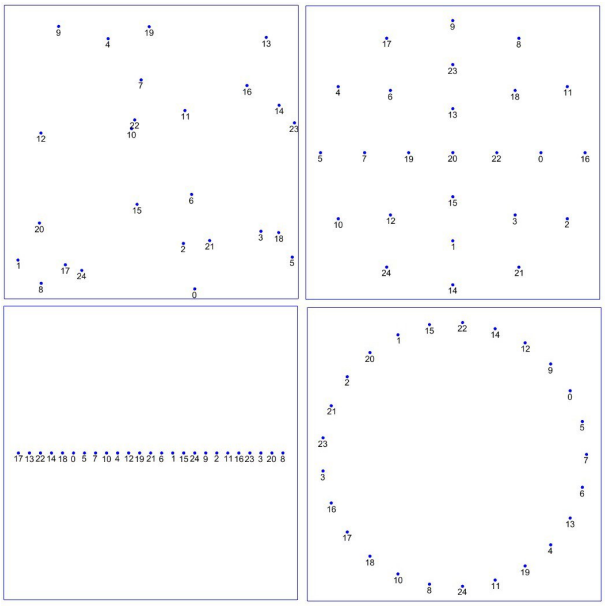

Figure 4: Example of topologies with 25 relays

Therefore, given one of these 4 topologies and certain environment dimensions, this component decides how the relays should be placed in order to form that topology and have equidistant locations (for the last 3 types naturally). Then, when a relay leaves the

environment (i.e. it crashes), *TopologyManager* handles this event, as well as the one when a new relay is joining the simulation, assigning it one of the available locations. Finally, the relays and perturbation types need to be graphically customized so that it becomes easy to identify a shape category based on its fill color. This task is achieved by the *custom_style* package, whose component names were not listed in order to avoid crowding the architecture diagram. More specifically, we have chosen this semantic for our simulation:

- **Blue dots**: relays
- **Orange dots**: perturbation containing a value broadcast
- **Cyan dots**: Perturbation containing a multicast message, suitable for both group and publish/subscribe
- **Pink dots**: plain unicast message
- **Black dots**: encrypted unicast message
- **Red dots**: retransmission request
- **Green lines**: edges between relays. These links are logical and they are established when the perturbation of a relay is sensed by an anonymous observer, in order to highlight the fact that these relays can reach each other with the perturbations they generate.

3 Implementation

Now that we have an overview of the architecture and what is the role of each component, we are going to explain how they achieve their goal. The implementation follows closely the algorithms and techniques proposed in the original paper. Some parts of the *Relay_II* algorithm were commented out of the final version of the source code, since they were replaced by the *Relay_III* implementation. However, it is possible to go back anytime to the previous implementation, simply by uncommenting a few lines of code and commenting others.

3.1 Topology management

When the simulation starts, the *TopologyManager* creates the amount of relays specified by the user, assigns them a globally unique identifier, and places them according to the chosen topology in the continuous 2D space. In case of the structured topologies (i.e. line, extended star, ring), the *TopologyManager* uses mathematical formulas in order to calculate locations that fill the entire available space and are equally distributed as

well. Otherwise, in case of the random topology, each relay will be assigned a pair of random coordinates and the density of relays might vary across the space. Finally, this component asks the *KeyManager* to generate new pairs of keys for all the relays, including the ones that will join later during the simulation.
Simulations start with a number of relays equal to the capacity chosen by the user, therefore at the beginning there are no available locations for new joining relays. However, a location is freed up when a relay crashes. The *TopologyManager* has the task to remove it from the context and mark its location as available. Please note that saving the specific location is only necessary for the structured topologies, since random topologies can place relays anywhere. As soon as new locations are made available, the *TopologyManager* simulates the arrival of new relays by generating them based on the probability parameter (more on this in the section dedicated to the parameters), and applying the same bootstraping operations which were listed for the initial relays.

3.2 Relay lifecycle

Relays are brought to the simulation environment by the *TopologyManager*. At each tick, they randomly generate a new perturbation with a given probability chosen by the user. Since there are 4 perturbation categories (retransmission requests excluded), this probability value is divided in 4 equal parts in order to obtain an uniform distribution of perturbations (e.g: if the the probability was equal to 1%, then the relay would have 0.25% probability to generate a plain unicast message, 0.25% to generate an encrypted unicast message, 0.25% for multicast and finally 0.25% for a broadcast). The expected amount of generated perturbations at each tick is hence the following:

$$n\_perturbations = probability * n\_relays \tag{1}$$

Going back to the 1% example, in a simulation with 200 relays, we are expecting to have 2 new perturbations at each tick. Multicast communications are simulated by picking randomly some of the relays and subscribing them to a mock group and a couple of mock topics.
Relays keep track of the all the perturbations that are still alive after every tick. Please notice this might include perturbations generated by both the relay itself or those received from somebody else. This operation is necessary in order to calculate the available bandwidth at each tick. Just like in the TCP/IP stack, we assume our perturbations are all of equal size (parameter which can be changed through configuration, 10 *units* by default). For example, given a bandwidth of 30 *units/tick* and a relay which currently has 3 perturbations forwarded by itself, each of them will propagate by 1 *unit* during a tick:

$$perturbation\_bandwidth = \frac{relay\_bandwidth}{n\_perturbation * perturbation\_size} = \frac{30 units/tick}{3 * 10 units} = 1 unit/tick \tag{2}$$

However, the propagation speed has an upper bound in order to prevent perturbations propagating almost instantly when the bandwidth is very large, and have a more realistic simulation. Finally, when a perturbation reaches its maximum range and it disappears from the environment, it is removed from the relay's list. As a consequence, the remaining perturbations will have more bandwidth available.
Relays implement also an automatic retransmission mechanism which has the goal to recover lost perturbations. At regular intervals and for each source in the log, relays send a retransmission request specifying the next expected perturbation for that source. Relays who happen to have that perturbation in their log will forward it. The original protocol suggests to not forward the retransmission requests but simulations have shown that relays might become isolated in presence of churn and if the forwarding is avoided. We therefore modified this aspect and enabled the forwarding of this kind of perturbations until they reach a relay which has the specific perturbation in their log. The automatic retransmission mechanism activates at a regular interval, which is equal for all the relays. However, they don't choose all the same tick to perform it because this might lead to flooding the environment with perturbations and consequently, block the execution. What they do instead, is choosing the tick following a common rule: the last digit of the tick should be equal to the last digit of the relay's identifier. For example, relays with identifier equal to 1, 11, 51, 121, 431 will all activate their automatic retransmission mechanism at tick 1, 11, 21, ... , 91, 101, 111 and so on (the interval is constant, 10 ticks). Similarly will do all the other relays which have the last digit in the identifier different than 1. This way, we are going to have an uniform distribution of retranmission requests and avoid slowing down the simulation. Relays have a certain probability to crash, which can be configured through the parameters. This event could occur at anytime during relay's lifecycle. When a relay is crashed, the *TopologyManager* will remove it from the environment and free up the location for a new relay.

3.3 Perturbation polymorphism and propagation

All perturbation categories have in common an identifier and a source identifier attributes. The first identifier derives from the internal logical clock of the source, which gets incremented every time a new perturbation is generated. The payload instead is a generic *Object* which can take various form and therefore define a specific perturbation type. In its simplest form, a payload is assigned a *String*, so the perturbation becomes the broadcast of a value. In case of unicast messages, the Object becomes an *UnicastMessage* encapsulating a destination and a *String* value. Furthermore, if the payload is encrypted, such as in the case of encrypted private communication, the payload will become an instance of the *SealedObject* class. Finally, group and publish/subscribe messages are represented using the *MulticastMessage* class which contains a group identifier, optionally one topic identifier and a *String* with the value.
When a perturbation is generated, it is originally placed in the same location as its source. Afterwards, at each tick it propagates with a given propagation speed and fol-

lowing a certain angle. The propagation speed is defined by the available bandwidth of the relay it was generated by, while the angle is assigned so that the dots which compose a perturbation will form a circle. For example, in a simulation with 8 dots per perturbation, each of them will be assigned an angle multiple of 45 degrees (360 degrees / 8). Propagation delay is simulated through a parameter which defines the probability of slowing down the propagation of a perturbation at each tick. A perturbation will propagate until it reaches the boundaries of the environment or until it reaches its maximum propagation range. Relays can pick up the perturbations when they get near them. There is a specific parameter for defining how near a perturbation needs to be in order to be sensed. Finally, is it worth noticing that picking up a perturbation won't remove it from the environment, so other relays will have the chance to pick it up too.

4 Performance analysis and parameter tuning

During this section, there will be many references to charts which are not present in the report. Whenever this happens, the image will be available inside the "Charts" folder.

In order to properly analyze this implementation, the results will be extracted from many different simulations. For each of these, there will be constant parameters that will define the specific baseline of the evaluation trying to represent a *realistic* scenario. The actual values taken into account to perform the simulations are the following:

- **Dimension of the environment:** 50 (measured in Repast Simphony space units)
- **Number of relays:** 20
- **Range of perturbation:** 8 (measured in Repast Simphony space units)
- **Relay's range of sensing a perturbation:** 5 (measured in Repast Simphony space units)
- **Probability of generating a new perturbation:** 0.05
- **Message size:** 10 (measured in any desirable unit as long as coherent with the bandwidth)
- **Maximum speed of perturbation:** 1.5 (measured in Repast Simphony space units over one tick)
- **Probability of random delays:** 0.0
- **Ticks of execution:** 1000
- **Random Seed:** 63.528.437

In this scenario, the topology and the bandwidth of the network were progressively changed in order to highlight the properties of this protocol. Static and dynamic networks have both been considered and the simulations have been performed by adjusting the bandwidth with respect to the specific needs of each topology.

4.1 Static network case

4.1.1 Broadcast latency

In order to measure the broadcast latency, the implementation selects the two most distant relays in the network; by doing so, the latency between these two relays will represent the latency of the network. In fact, any perturbation sent by one of these relays will be deliverable by the other only if the perturbation itself was able to pass through the whole network. For this specific evaluation, the random topology won't be considered, since the results depend strictly on how the nodes are positioned in the environment.

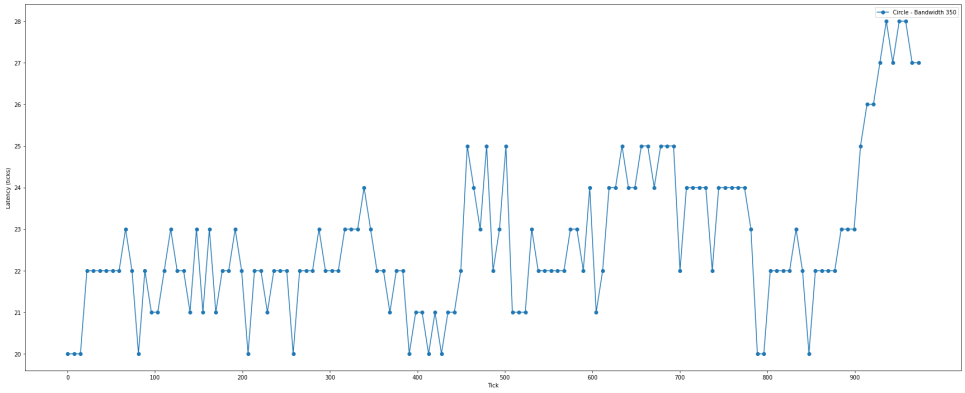

Figure 5: Broadcast latency w.r.t. the time of the execution by fixing a specific value of bandwidth

Let's consider Figure 5, where the latency is measured in a ring topology with bandwidth equal to 350. By looking at the chart, it is possible to see that the average latency is approximately 22 ticks; however, ups and downs respectively highlight that there are low and high traffic time intervals using these parameters. This means that for this protocol, the ring topology is an option only when the traffic that will travel throughout the network is well known in order to choose carefully the bandwidth. The extended star topology has a similar behaviour (Figure "Latency-Tick-Star-460") but with a much lower average latency. However, in this case, a higher bandwidth is needed in order to let the simulation finish in a reasonable time. Finally, the line topology (Figure "Latency-Tick-Line-420"), shows a null variation on the latency that stays for the whole execution at value 10. Even though it needs a higher bandwidth w.r.t. the circle topology in order to finish the simulation in a reasonable time, it exploits the environment dimension by

reducing the gaps between relays; by doing so, any new perturbation can be almost instantly sensed by the nearby relays. This achievement, however, comes at the cost of a large unused area of the environment.

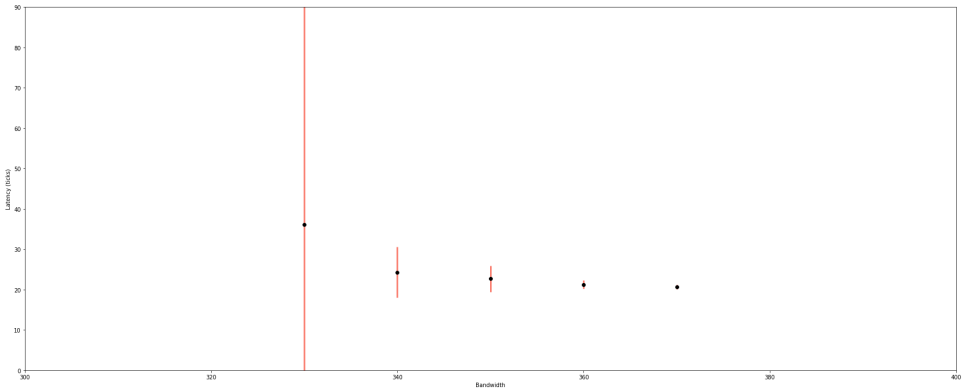

Figure 6: Broadcast latency w.r.t. the bandwidth variation in a circle topology

Starting from the Figure 6, it's possible to get a better overview on the impact of the bandwidth in the ring topology. The red lines, indicating the variation of the latency grows exponentially when the bandwidth decreases. With a value lower than 330, the protocol does not give any guarantees about the time it takes for a perturbation to go throughout the network. With a value greater than 370 instead, the perturbations can travel with a speed high enough to eliminate any possibility of congestion of the network. The extended star topology similarly (Figure "Latency-Bandwidth-Star"), just in a different bandwidth span and with a different overall latency. However, it is the line topology that shines again, by showing a clear 10-ticks overall latency that stayed constant until the execution could not be performed anymore due to the insufficient hardware resources (Figure "Latency-Bandwidth-Line"). The exponential increase of variance in the circle and extended star topology is caused by a double dependency: perturbations get slower and slower when they become too many w.r.t. the bandwidth and, at the same time, they increase in number since none of them will be able to reach the maximum range and disappear, releasing the bandwidth for new ones. This means that if the bandwidth is not large enough, the perturbations will eventually become too many for the relays to process them in a sustainable speed, which will cause new retransmission requests to be generated, slowing down the whole network even more.

4.1.2 Relay isolation

Another aspect that has been considered in this evaluation is the number of isolated relays w.r.t. the execution time. Isolation is a property indicating that a relay is not logically connected to any other relay in the network, hence no perturbation will reach the isolated relay. Since only static networks are considered in this section, we expect to have a strictly non increasing number of isolated relays, as shown in Figure 7

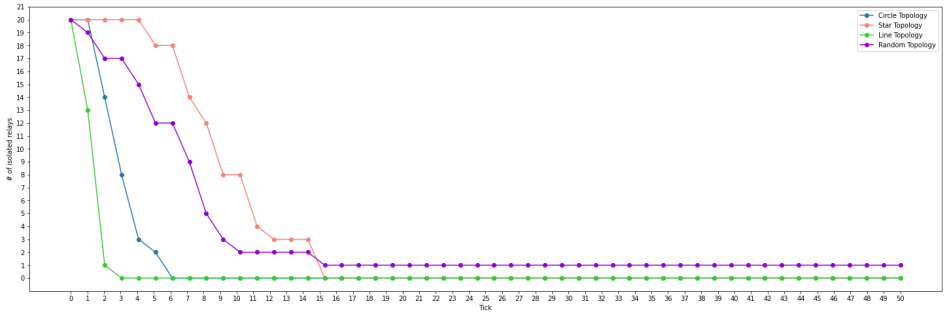

Figure 7: Number of isolated relays w.r.t. the time of execution

As shown, only the random topology has a chance to end up having a number of isolated relays greater than 0 (not considering limit cases, e.g. where the scenario is composed by an enormous environment filled by just a few nodes). In fact, by placing the relays in random positions, it is possible that some of them end up out of range of the others. These results were obtained by fixing the bandwidth at a value that could avoid any kind of network congestion; based on these assumptions, each topology would reach complete connection at different tick count. Once again, line topology created a fully connected network in just 3 ticks, overcoming the others results, for the very same reason of the small gap between the relays.

4.2 Dynamic network case

Many issues may arise when considering a dynamic environment. First of all, any evaluation made previously could change, since each topology reacts differently to relays' crashes. Moreover, the bandwidth needed to sustain the simulation (in order to make it finish in a reasonable time) will generally be higher. This is due to the fact that temporary disconnections require more perturbation exchanges in order to reflect the status of the network.
In these simulations, the probability of a node to crash (or join) has been set to the value of 0,001.

4.2.1 Broadcast latency

The way this implementation measures the broadcast latency is the very same it was used the static network version, making an exception and considering the two selected nodes as ”immune” to crashes for simplifying the tests.
In Figure 8 it is possible to see a completely different behaviour compared to Figure 5. First of all, the bandwidth needed to execute a simulation in a reasonable amount of time with a ring topology is greater by 100 units. Moreover, there is a noticeable difference between two portions of the chart: up to tick ˜800 the execution proceeds normally; however, after that point, there is a spike in latency that moves from the

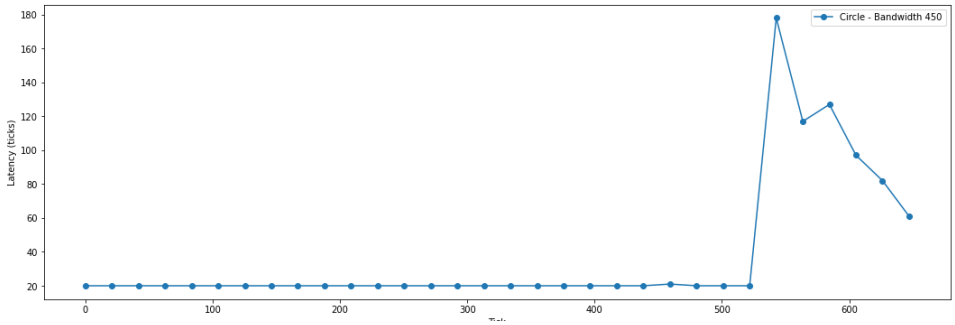

Figure 8: Broadcast latency w.r.t. the time of execution by fixing a specific value of bandwidth (in a dynamic network)

average of 22 ticks to almost 180 ticks. This increase is due to the fact that the circle topology has been interrupted by the crash of at least 2 relays, that created 2 isolated sub-networks. Once the topology will be reconstructed by the arrival of new relays, the perturbations could propagate again through the whole network, but with a large latency, since they have to catch up on the old perturbations for the amount of time the sub-networks were isolated. The extended star topology needs a higher bandwidth as well (File ”Dynamic-Latency-Tick-Star”); however, this topology shows in this case an important property: robustness. With the same amount of crashes, the overall latency is much lower compared to the circle topology. By having a high amount of logical connections, the extended star topology is harder to ”break” into several isolated sub-networks and can easily find alternative paths among relays. This difference is highlighted by Figure 9 and Figure 10. Note that, in Figure 10 the variation seems not to be coherent with the bandwidth; this is due to the fact that the bandwidth was selected based on the hardware limitations and Repast Symphony performance. Hence, the variation here does not depend on the bandwidth, but on the way the relays crashed during the simulation, which is different each time the execution is performed.

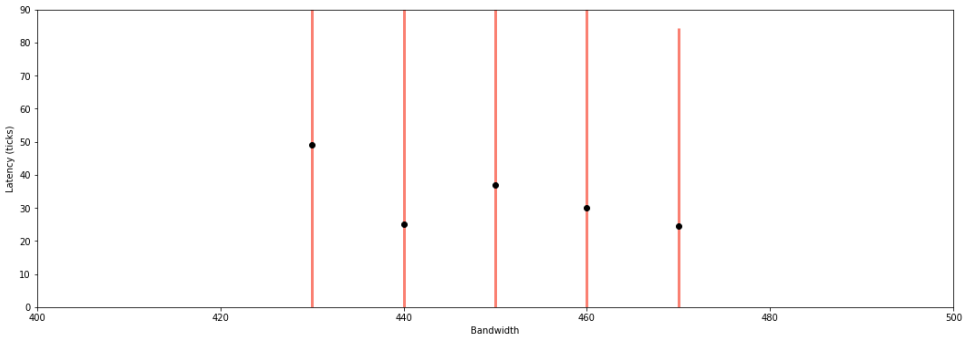

Figure 9: Broadcast latency w.r.t. the bandwidth variation in a circle topology (in a dynamic network)

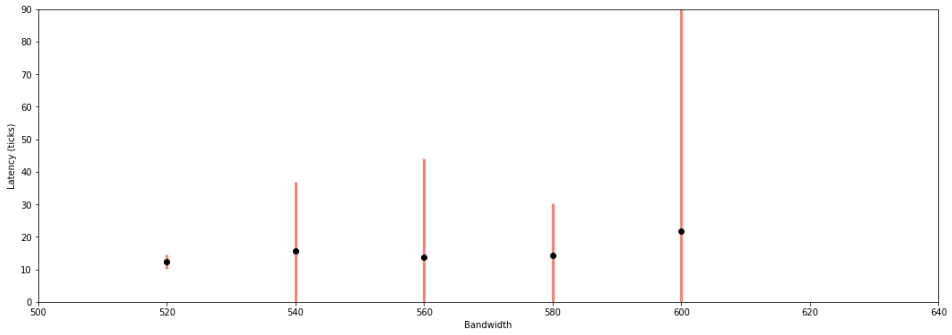

Figure 10: Broadcast latency w.r.t. the bandwidth variation in a complete star topology (in a dynamic network)

Finally, we get to review the line topology. Unfortunately, Repast Simphony couldn't handle the simulation at any useful value of bandwidth. The reason behind this is that the line topology, even if it shines in a static network, has no robustness. Even with a single relay's crash, the topology is "broken", and each time a new relay joins, a huge amount of messages has to be traded in order to update the network.

4.2.2 Isolated relays

In the dynamic case the simulation execution was extended from 50 ticks to 1000 ticks. In fact, while in the static case the number of isolated relays could not grow, in this case might happen. Therefore, it is relevant to see how this value modifies in time.

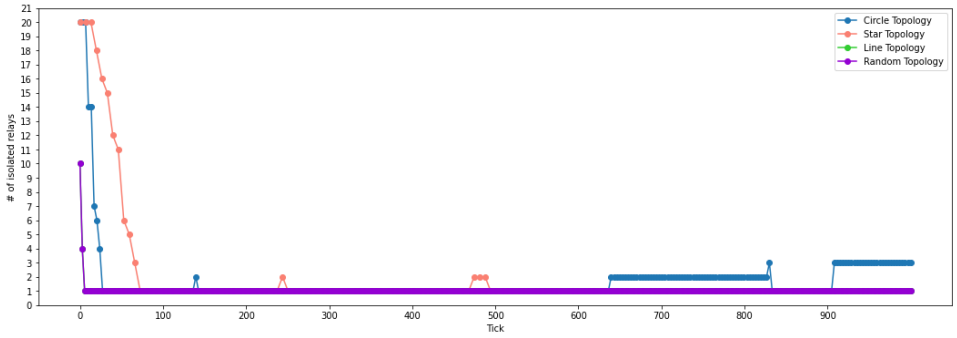

Figure 11: Number of isolated relays w.r.t. the time of execution (in a dynamic network) (Note that the line topology is not visible because has the very same behaviour as the random topology, that is covered by)

Taking into account the Figure 11, it is possible to understand that, in the dynamic case, this value is not really relevant. In fact, each topology behaves similarly, and nothing could be deduced from this chart. The reason behind this is that the main issue with dynamic networks are unconnected sub-networks and not isolated relays.

5 How to install the simulator

Once the *setup.jar* file was downloaded from the link found in the repository's README, it's possible to follow these steps in order to install and execute the simulator:

- Make sure JRE 11 is installed on your system.
- Run the *setup.jar* using the system GUI or by typing *java -jar setup.jar* in the terminal. Follow the wizard steps in order to accept the licence and choose the installation directory.
- Launch the simulator by executing *start_model.bat*. A Java application will be displayed on your screen. If you are using an Apple Mac device then you have to execute *start_model.command*.
- Set the values of the parameters from the *Parameter* window. Once you have chosen the parameter values, it is possible to save them for further executions.
- Finally, you can launch the simulation by clicking on the *Initialize Run* button placed on the upper bar. You can start the simulation by clicking on *Start Run* button or by clicking on the *Step* button. Simulation execution can be slowed down by increasing the *Schedule Tick Delay* in the *Run Options* window.

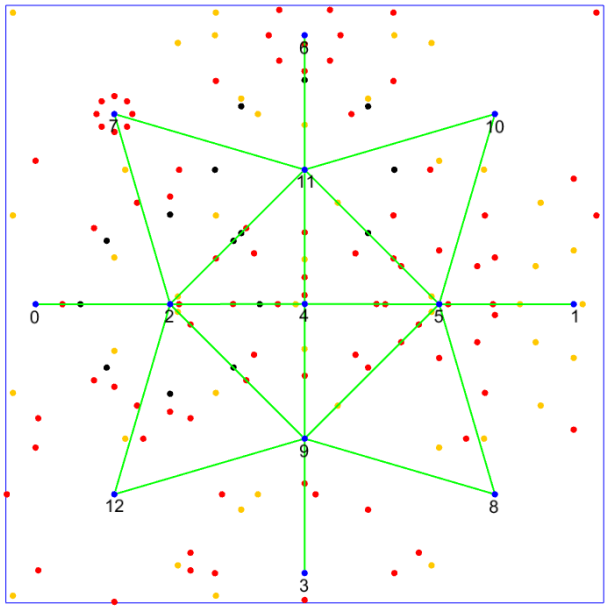

Figure 12: Example of an execution using this simulator

Conclusions

With this simulator all 3 Relay variants were implemented, as well as all the communication primitives proposed in the original paper [1]. The implementation is resilient to arbitrary network dynamics, delay , and losses. In the context of perturbation recovery in a dynamic network, we have learned that it's necessary for the relays to forward the retransmission requests, as opposed to what was suggested in the original paper. If we didn't do so, isolated networks would have remained as such permanently. Furthermore, we consider that the performance of the protocol could be greatly improved if multiple retransmission requests were piggybacked inside a single perturbation, instead of generating separate perturbations for each request. The retranmission requests represent a good percentage of the whole traffic, especially in presence of churn. By piggybacking these requests, the traffic is decreased significantly, as well as the processing that needs to be done by the relays. Finally we've run several tests on the 4 topologies with different scenarios and compared the results. Each topology has its own strong and weak points, and they are often defined by key aspects such as the available bandwidth, the relays density and the churn rate.

References

[1] Christian F. Tschudin. A broadcast-only communication model based on replicated append-only logs. pages 2, 17