Broadcast-only communication model

Simone Degiacomi
211458
simone.degiacomi@studenti.unitn.it

Davide Tessarolo
211457
davide.tessarolo@studenti.unitn.it

Davide Zanella
211463
davide.zanella-1@studenti.unitn.it

I. Abstract

In this report we describe how we implemented the concepts presented in the "A Broadcast-Only Communication Model Based on Replicated Append-Only Logs" paper [1] assigned for this task, and an analysis of its performance depending on different configurations and scenarios. This protocol, and its variations, aim to provide different network features (point-to-point communication, recover from losses and other features built on top of those) by using only a communication primitive: the broadcast of a Perturbation, hence the name "Broadcast-only".

II. Implementation

This project has been implemented using the Repast Simulation Framework, with the Java programming language. With these technologies it is easy to design and implement an actor based system to perform the simulation. Repast, as contrary to Akka, includes the concept of positions of actor in a space grid. This allowed us to model the perturbations as actors, that propagates trough space at each simulation tick. The other actors we implemented are: the stations that broadcast perturbations, the relays that receive and re-broadcast perturbations, and finally a simulation manager, that crashes and spawns relays, and drops perturbations to simulate network conditions according to the simulation configuration.

A. Difference between Stations and Relays

We assume that stations and relays are different actors: stations generate perturbations, while relays receive and *relay* (propagate) them. This was done because the paper makes an important distinction between the two: for relays, it says that they are "capable of picking up solitary waves in one media and propagating them to another media", while the stations are the triggers of the perturbations.

B. Perturbation is an actor

Why did we choose to model perturbations as actors? Initially we did not: instead, stations and relays passed perturbation objects to their neighbours. However, by doing so we had to re-implement somehow the concept of distance between relays: passing the perturbation to near relays took otherwise only one unit of time (tick), independently of the distance between the relays. For this reason we decided to exploit the spatial concepts available in Repast to create an actor that represents a perturbation: a circle whose radius increases as the time passes, and whose center is the position of the station that generated it (or the relay that propagated it). This allows us to better simulate the behaviour of a wave in a wireless environment. We can summarize the lifetime of the perturbation actor with the following steps:

1) a station creates a new Perturbation and adds it to the Repast context. The initial radius of the perturbation is 0 spatial units;
2) at every simulation tick, the Perturbation increases its radius. The speed with which the perturbation enlarge its radius decreases with every tick (the reason for this is explained in the next section). After having increased its radius, the Perturbation checks if some relay has now received it: this is done by checking if a relay lies in the incrementation ring (that is, the ring computed by the subtraction of the new perturbation area and the previous one). If that is the case, the perturbation calls the *onSense* method on the Relay, which will then process the perturbation according to the current configuration, and then propagate a new Perturbation object (with its new lifetime);
3) when the perturbation reaches a radius greater that the size of the Repast simulation grid, the Perturbation gets removed from the Repast context. Thus, it is also important to note that each perturbation is global;

C. Propagation speed decreases over time

In the second step of the perturbation lifetime, we said that we enlarge the radius less and less as time passes. Specifically, we compute the increment using the following formula:

$$increment = 0.5 - sigmoid(tick)$$

Where sigmoid is defined as:

$$sigmoid(x) = \frac{1}{1 + e^{\frac{x}{10}}} - 1$$

We slow down the perturbation to satisfy the requirement of the original paper that says that, in the case of a dynamic network (where new relays join the network while the protocol is running), it could be the case that adding a new relay decreases the latency of perturbation propagation. If we kept the propagation speed constant, this would not be true: imagine a scenario where we insert a new relay C between station A and relay B. If the propagation speed remains constant, adding the relay C would not improve the latency of a perturbation sent by A, as the re-propagated perturbation from C would propagate together with the original propagation from A.

Decreasing the propagation speed also allows us to better model the behaviour of a wave in a wireless environment. The formula was taken from a real-scenario wireless wave speed decrease, but it was adapted to better suit the needs of our simulation.

D. Simulation Manager

The SimManager class, or Simulation Manager, is an external actor that manages the aspects that are out of the control of other actors. This includes creating or crashing relays, randomly dropping perturbations and initializing some aspects of the simulation, such as topics for the Group Publisher/Subscriber communication variant of the protocol or private and public keys for the privacy preserving one. Especially for the initialization cases, SimManager allows us to have a powerful tool to manage these aspects, without having the complexity that would be due to its absence.

E. Stations

Station is a rather simple actor which sole purpose is to generate perturbations for the relays to receive. While perturbations are different in the various protocol variants, there is only one implementation of the Station actor (unlike relays). The adaption of the perturbation generation, for the various variants of the protocol, is done by changing the flags of the Station actor, which in turn changes the perturbation it generates directly in the *sendPerturbation* method.

F. Relays

The original paper describes three main protocols for relays. To implement this specification avoiding duplication of code, and to simplify the initialization of the simulation, we created the Relay abstract class, with three different implementations.

To simulate the concept of bandwidth, the Relay abstract class throttles the propagation of perturbations according to two parameters: the size of a packet and the transmission speed. When a Relay implementation calls the *forwardPertubation* method, the abstract Relay will add the Perturbation to the *perturbationsToSend* queue. Then, at every tick, two scenarios are possible:

- the Relay can start to send a new Perturbations if no other Perturbation was already being sent;
- the Relay can continue to send a Perturbation, and removes it from the queue once it has been sent entirely;

The number of ticks needed to send a perturbation can be computed by dividing the size of a packet by the transmission speed.

1) RelayI: Perfect conditions: This implementation is intended to be used in a simulation where there are perfect conditions: no relay is spawned or crashed during the simulation, and no perturbation gets dropped. The concept of the *frontier* is implemented using a HashMap, that maps a station id to the next expected perturbation ref from that Station. When a Perturbation is received, the Relay checks that it is the expected next perturbation from the sender station, and then propagates it.

2) RelayII: Dynamic Network: This advanced Relay implementation can tolerate changes to the topology of the network, which can lead to out-of-order reception of Perturbations. This Relay will propagate immediately Perturbations received in the expected order, but will delay the propagation of OOO (out-of-order) ones, by storing them in the *bag* collection. RelayII will propagate postponed perturbations as soon as the missing previous perturbations are received, and will try to empty the bag as soon as possible.

3) RelayIII: Recovering Loss: This last Relay implementation can withstand unstable network conditions where perturbations can be lost, both for being dropped by the SimManager or due to the crash of other relays. This is accomplished by a protocol that tries to recover from losses thanks to Automatic Repeat Requests (ARQs): periodically, each Relay will propagate ARQs that ask for the next expected perturbation from all the known stations. Relays that receive an ARQ will propagate the answer if they already received the requested perturbation, otherwise they will just ignore the request and will not propagate it. Differently from the bag of out-of-order perturbations, this implementation stores all the received perturbations forever, increasing the memory requirements as time passes.

4) Point-to-Point communication: We added to the RelayIII implementation the capability of Point-to-Point communication. This is achieved by adding a unique identifier to stations and relays, and adding two fields in the Perturbation actor: the sender and receiver identifiers. When running a simulation with the Point-to-Point capability enabled, the RelayIII will continue to forward the received perturbations as it did before, but it will process only the ones whose destination id corresponds to its identifier.

5) Privacy-Preserving communication: One of the biggest flaws of our Point-to-Point communication is that every relay in the network can read the content of every message, even if the Perturbation is not addressed to them. We solved this problem by integrating asymmetric encryption of the messages. The generation of private/public key pairs is taken care by the SimManager, and stations encrypt the message inside a perturbation using the public key of the destination relay.

6) Group Publisher/Subscriber communication: Through a simple modification of the message carried by perturbations, and by filtering messages in the relay, we implemented a variant of the protocol that allows relays to subscribe to topics published by stations. A topic is identified by a string, and this act as an identifier of multiple relays: the group of relays interested in the topic.

G. Log collection and data analysis

To efficiently run many simulations and summarize the results into graphs, we used the batch run feature of Repast together with Python scripts. Specifically, we defined fifty-four types of batch configurations: nine types of network configuration (number of stations and relays, probability of relays being added or crashed and probability of perturbation

drops) multiplied by our six protocol variations. We run each batch configuration twenty-five times, and all the results were collected into CSV files, using Repast File Sinks. Once all the logs were collected, three different Python scripts were used to draw the plots:

- *create_plots.py*: this script can operate in different modes. The first generate a plot that shows the latencies of every single perturbation of a simulation; The second mode, given a log of a batch simulations, computes the mean latencies of every perturbation of a run, preparing a CSV file that will be used by the second script;
- *plot_multiple_runs.py*: this script uses the file prepared by the first script and plots the mean latencies of perturbations of multiple configurations in a single plot;
- *launch_script.py*: this script combines the first and the second one, to generate plots using only one command for network configuration;

At each execution of each batch run, the position of each station and relay is decided in a pseudo-casual way by Repast: a random seed is issued for every run of the system, determining how the simulation is generated. The high number of runs for each configuration serves the purpose of generating a relevant dataset, thus eliminating the problem of outliers caused by particular topologies[1].

III. Running the simulation

Running our project in Eclipse using the default simulation window, we can see the scenario tree with the default data loaders, datasets, display and sinks already selected. Switching to the parameters tab allows us to customize the different parameters for the next run. The parameters are:

- *Default Random Seed*: the random seed to be used in the simulation initialization and management. Using the same seed in multiple runs will, for example, result in stations and relays having similar positions (randomly generated using the same seed);
- *Maximum number of relays*: the maximum number of relays that can be present on a given run;
- *Minimum number of relays*: the minimum number of relays that can be present on a given run;
- *New relay probability*: the probability that a new relay will be spawned in a period of 100 ticks;
- *Relay crash probability*: the probability that a new relay will be deleted in a period of 100 ticks;
- *Number of relays*: initial number of relays;
- *Number of stations*: initial number of stations;
- *Packet size*: size of packets sent via perturbations;
- *Transmission speed*: available bandwidth per tick;
- *Perturbation drop probability*: probability that a perturbation is dropped;
- *Protocol version*: defines which protocol version (from those presented in the paper) the simulator will run;
- *Stop at*: the simulation will stop once it reaches the indicated amount of ticks.

Starting a new simulation creates a grid with the indicated number of stations and relays. Stations will then start generating perturbations as indicated in the parameters and according to the protocol version, as can be seen in *Fig. 1*. The simulation can be run automatically (in this case it is advisable to set a high value for the "Schedule Tick Delay" option in the "Run

[1]Possible outliers could be an incredibly low latency value caused by a topology where all stations and relays are near to each other, or a very high latency value caused by a topology where stations and relays are all situated at the edges of the grid.

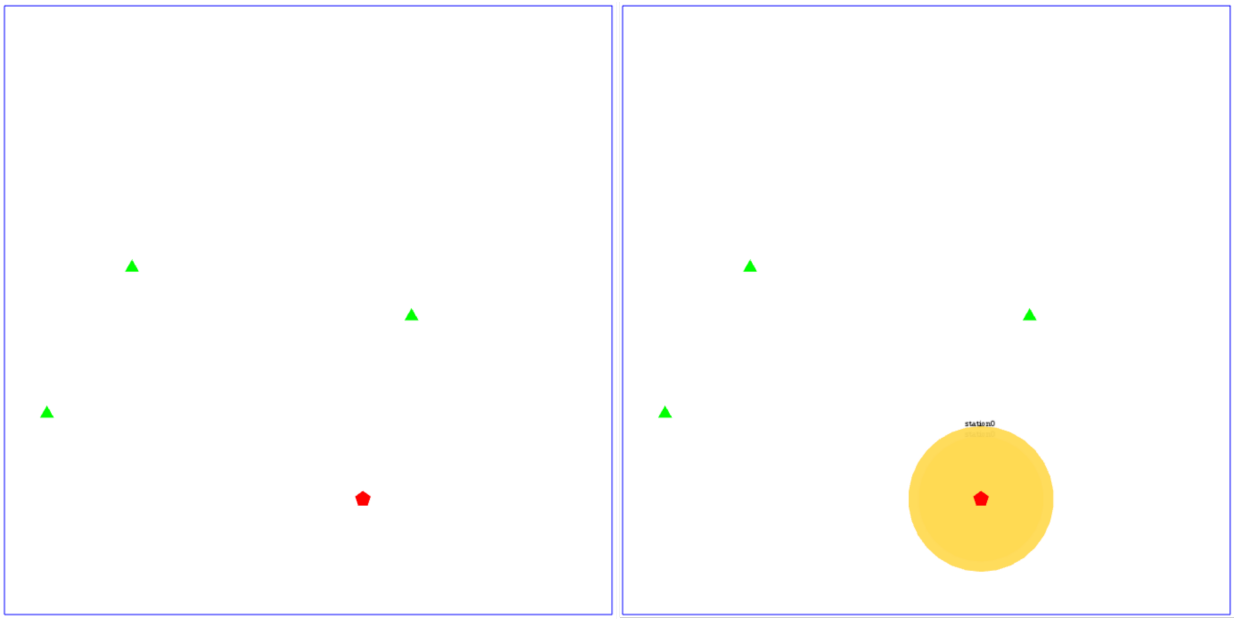

Fig. 1. Left: a newly initialized simulation with one station (red dot) and three relays (green triangles). Right: The same simulation a few ticks after the station has emitted a perturbation.

Options" menu) or tick-by-tick for examining it with a better precision.

IV. Results

While the original referenced paper does give implementation details about the model, it doesn't tackle the possible performances its different variants could present. In this section we will analyze the latencies and the number of reached relays by every perturbation for each Relay implementation, each one in the nine different scenarios. The chosen parameters are reported in *Table I*.

Here we can see how each scenario incrementally adds variables to the simulation: in *Scenario 0* we find a rather simple simulation, where the scene is static (no new relays are added/removed and perturbations are not dropped). *Scenario 1* adds the possibility for new relays to appear and crash, assuming an upper limit of 60 and a lower limit of 40. *Scenario 2* and *Scenario 3* test different packet sizes and transmission speeds, while in *Scenario 4* and *Scenario 5* we return to a more simple but more dynamic scene. From *Scenario 6* the number of stations gets higher, while in *Scenario 8* we add the possibility for perturbations to be lost, with two different probabilities.

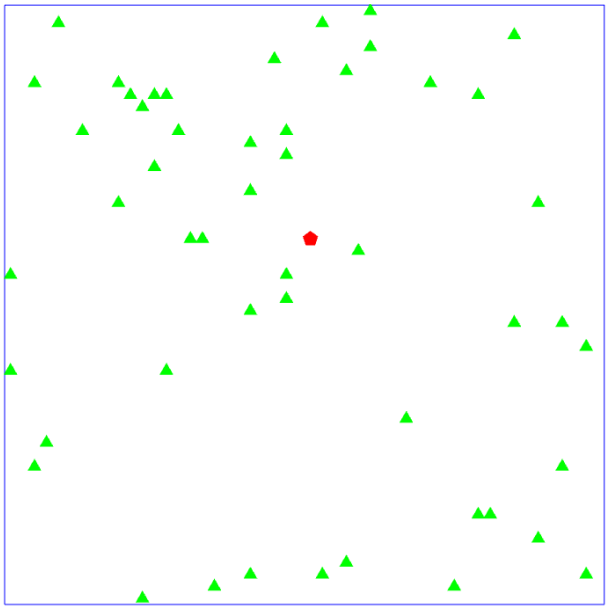

Fig. 2. A sample Scenario 0 where the is a single station (red dot) and 50 relays (green triangles).

A. Data gathering and analysis

Data gathering was done via logs: each station and relay were in fact able to record to two separate log files (one for stations, one for relays) the send and receive events of perturbations. These logs were then summed up via the *create_plots* Python script to a single CSV file containing all variants of the model. Finally, the generation of the plots was made using the *plot_multiple* script and the matplotlib library. Two different graphs were made for each one of the nine scenarios, one plotting the latency of every perturbation for the variants of the model and one plotting the number of relays reached by that perturbation in that variant.

B. Scenario analysis

1) Scenario 0: As already noted, Scenario 0 uses 1 station and 50 relays in a static environment, where relays cannot crash or be spawn. In this scenario, as show in *Fig. 3*, the *Perfect Condition* and *Dynamic Network* variants of the model behave in the very same way, even if the latter has some small overhead due to its relays keeping logs for a short amount of time. The *Point to Point*, *Privacy Preserving* and *Group Communication* tend instead to show more swinging values due to them being non-uniform (in a broadcast environment all relays are always reached, while in a point-to-point one the distance between the station and the relay changes between the perturbations, leading to changes in latency values).

In this (and all subsequent) scenario we can see how the *Recovering Loss* variant of the model performs slightly worse than its *Dynamic Network* and *Perfect Condition* counterparts. This is due to the periodical ARQs, which increase the traffic in the network, even when an ARQ does not get answered.

2) Scenario 1: In this scenario we start to have a more realistic environment (although still far from a real one), where relays can crash and be spawn. The probability is still rather low (a relay will crash/get spawned with a probability of 10% every 100 ticks). We can see how this does not affect too much the latency in the *Dynamic Network* scenario: the price of getting OOO perturbations is usually really low, as it can be seen on the graph in *Fig. 5*. This is because OOO perturbations usually come only a few ticks before the ones they are preceding, slowing down the delivery by a really short amount of time.

Fig. 6 shows how the Relay's versions that have high latencies, are the ones that reach more relays, so this two values are directly correlated.

3) Scenario 2: This scenario is very similar to its predecessor, but the size of the packets has been doubled. We can see how this has little to no effect on the *Perfect Condition* and *Dynamic Network* variants. The explanation for this behaviour is that, to forward a perturbation, a relay needs two ticks instead of one, but this extra tick is just a little delay compared with the ticks needed by the perturbation to travel across the space.

Even the *Point to Point* and *Privacy Preserving* models behave similarly to the previous scenario, while the *Group Communication* and *Recovering Loss* ones have peaks in different period of the simulation, maybe due to some random crashes of the relays since we are averaging only 25 runs, but the number of reached relays is almost the same, as shown in *Fig. 8*.

4) Scenario 3: Scenario 3 is Scenario 2 with the packet size doubled an additional time (arriving at a size of 400 per

TABLE I
LIST OF THE PARAMETERS OF EVERY TESTED SCENARIO

Scenario	N° stations	N° relays	New relay and crash relay probability	Packet size	Transmission speed	Perturbation drop probability
0	1	50	0%	100	100	0%
1	1	50 ± 10	10%	100	100	0%
2	1	50 ± 10	10%	200	100	0%
3	1	50 ± 10	10%	400	100	0%
4	1	50 ± 10	30%	100	100	0%
5	1	50 ± 10	30%	200	100	0%
6	5	50 ± 10	10%	100	100	0%
7	10	50 ± 10	10%	100	100	0%
8	5	50 ± 10	10%	100	100	50%

packet). With a transmission speed of 100, it means that the single station in the scenario (and subsequently all the relays), take four ticks to send a single perturbation, as it can be seen from the plots in *Fig. 9*. This is noticeable in the graphs where, particularly in the *Perfect Conditions* and *Dynamic Network* scenarios, the latency is basically the same of Scenario 2, but with four additional ticks of latency.

The behaviour of the *Recovering Loss* and *Group Communication* variants and the reason why the latencies decrease after the initial part of the simulation is explained in the next scenario, because it is more evident.

The number of relays reached by the different protocols is the same as the Scenario 2, as demonstrated in *Fig. 10*.

5) Scenario 4: Scenario 4 highlights, thanks to a higher relay crash/spawn probability, how the first perturbations have higher latency values with respect to the ones that are sent later in the simulation. This happens with the *Recovering Loss* and *Group Communication* variants that, thanks to the RelayIII implementation, try to recover all past perturbations when a new relay is spawned. This creates a situation where the newly spawned relays receive old perturbations late in the simulation, driving the latency of earlier ones up. The effect can be seen on *Fig. 11*.

Analyzing Fig. 12, we can derive how, for every Relay's protocol, the number of reached relays, is bigger than in the previous Scenario, due to the higher probability of crash/span of the relays.

6) Scenario 5: This scenario is similar to Scenario 4, but relays take two ticks to propagate a perturbation due to an increase of packet size.
In *Fig. 14* we can see that this change does not affect the number of relay receiving the perturbations, and, as expected, in *Fig. 13* we can see slightly higher latencies. The increase in latency is small for the same reasons explained in the Scenario 2.

7) Scenario 6: Scenario 6 is the first one with multiple stations. From *Fig. 15*, we can see how the effect shown in Scenario 4 (high latency during the first perturbations) is more highlighted here, even with a low probability of 10% for relays spawn and crash (due to simulation having five stations, and so five times the amount of perturbations). On the other hand, *Perfect Conditions* and *Dynamic Network* variants show basically the same latency values. This is due to the lower traffic in the network, showing once again how cost-efficient is the *Dynamic Network* protocol. Fig. 16 shows how the number of reached relays by the *Recovering Loss* and *Group Communication* protocols is reduced in a specific moment (the same when the latency decreases sensibly).

8) Scenario 7: In this scenario, there are 10 stations, all of them sending one perturbation, starting from a random moment, every 100 ticks. The result is that we have 10 times the number of ARQ requests and their answer perturbations with respect to Scenario 1, where we have only one station. The behaviour of this scenario, as shown in Fig. 17, is driven by the phenomenon already noticed in the Scenario 4 and 6, but emphasized by the higher number of stations. Comparing the plot reporting the number of reached relays of this scenario, Fig. 18, with the one of the previous scenario, we can observe how the minimum value of the *Recovering Loss* curve is lower in this scenario.

9) Scenario 8: Scenario 8 has been used to simulate a busy network with a very high probability of losing perturbations. This could be, for example, the case of a network with a high amount of background noise. As shown in Fig. 19, *RecoveringLoss* is able to maintain good latency even with bad network conditions. Most importantly, in Fig. 20 we can see that perturbations are still able to reach an high number of relays.

In this scenario, analyzing the *Perfect Conditions* and *Dynamic Network* variants doesn't give us many information. In our simulations they still achieve good results, but this is due to the high number of relays. In fact, if only one perturbation gets lost before it reaches any relay, not only this perturbation will never be delivered, but also all the next coming from that station: those two type of relays propagate and deliver only perturbations having the expected perturbation *ref*; once a perturbation is lost, there will never be a perturbation with the expected ref coming from that station.

V. CONCLUSION

In this report we presented our implementation of the protocols proposed in the assigned paper [1], and we run

several simulations to measure its performance and compare the different variations. After having conducted our analysis, we discovered that:

- the *Perfect Conditions* relay, as expected, is the most suited in high reliable networks, since it provides low latencies and does not involve any overhead;
- the *Dynamic Network* relay does not add additional traffic, but can work properly even with OOO perturbations;
- the *Recovering Loss* relay is suggested only in networks that are not reliable and may drop perturbations: we do not suggest to use it in other scenarios, since it comes with additional network overhead if perturbations are assured not to be dropped;
- the *Group Communication* relay seems to behave slightly better than the *Recovering Loss*, involving less relays with slightly lower latencies;
- the *Point to Point* relay is a good solution if we are interested only in a unicast communication, and the latencies are affordable;
- the *Privacy Preserving* relay behaves like the Point to Point relay, but assuring privacy between the involved agents. In this simulations the encryption overheads were ignored while collecting metrics;

Even though *Perfect Conditions* had good performance in our simulations, we do not encourage to use it in a network that may drop perturbations, as any perturbation dropped may cause a crash of the entire protocol.

Looking at the graphs and results obtained, we believe that twenty-five runs for each scenarios may be a number too small, so running more simulations may be a good first step to continue this study.

How to install

In order to install our simulator, you have to follow these steps:

- Download the *installer.jar* file from google drive. You can find the download link in the Readme file of our repository on GitHub;
- Execute the *installer.jar* file. From the console you have to type *java -jar installer.jar*;
- An installation wizard will pop-up, just follow it;
- Once installed, you can execute it by opening the *start_model.bat* file or the *start_model.command* one. Adjust the parameters and then enjoy the simulations!

References

[1] Christian F. Tschudin, "A Broadcast-Only Communication Model Based on Replicated Append-Only Logs", ACM SIGCOMM Computer Communication Review, Volume 49 Issue 2, April 2019.

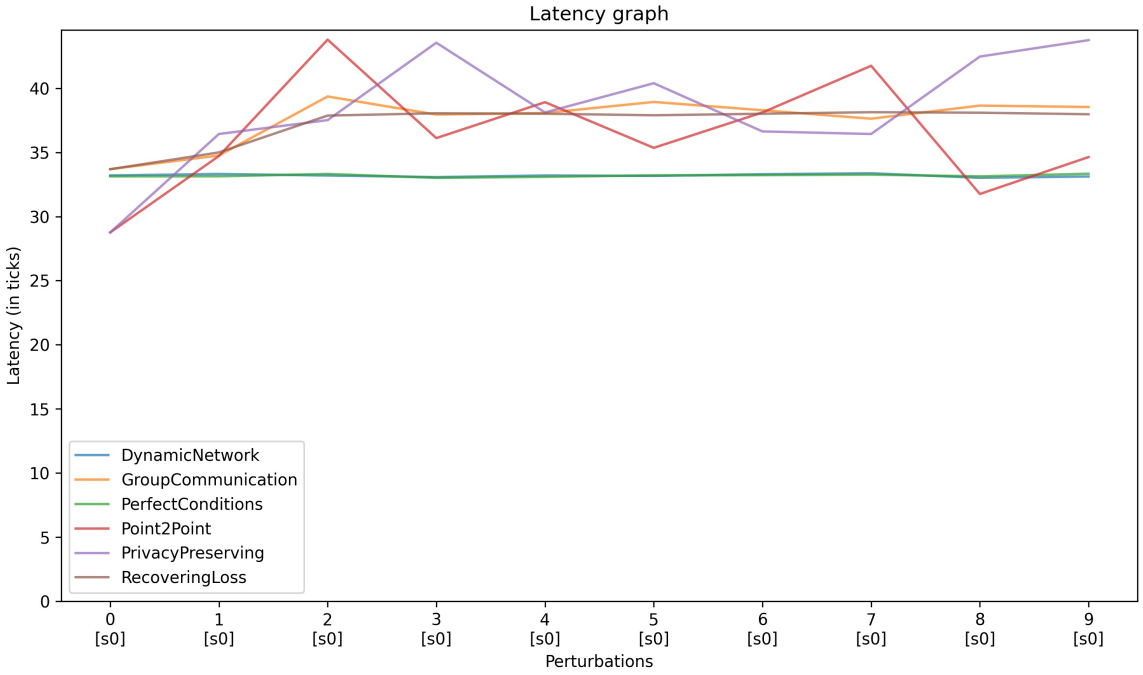

Fig. 3. Latency plots for Scenario 0

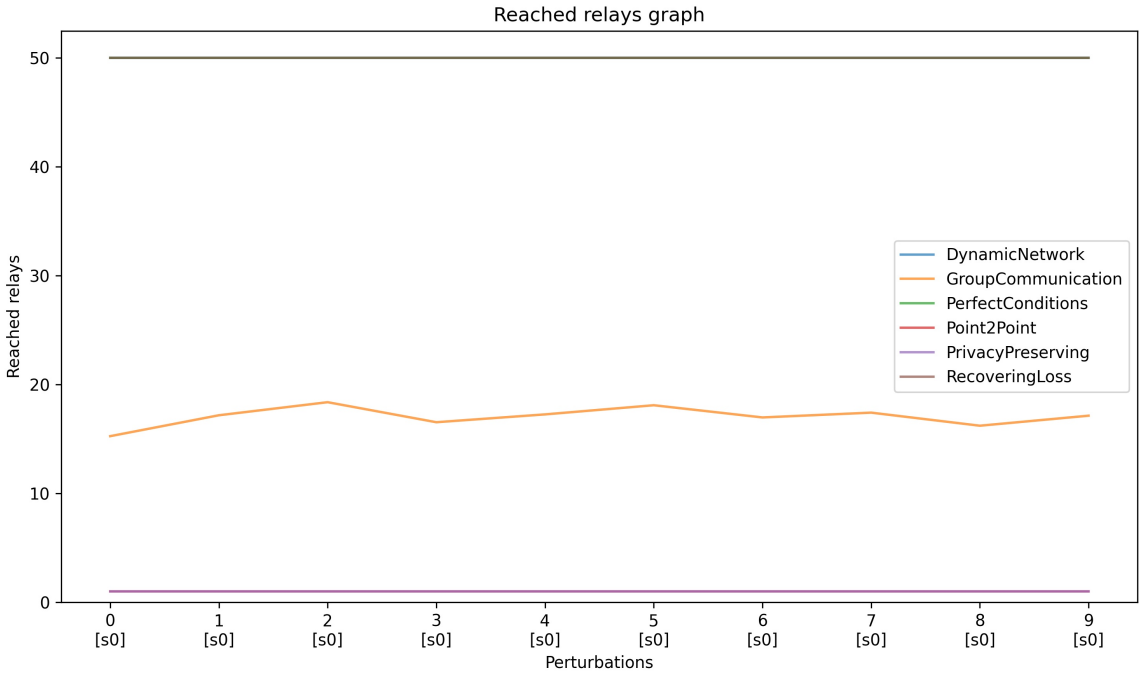

Fig. 4. Plot showing the number of reached relays by every perturbation in Scenario 0

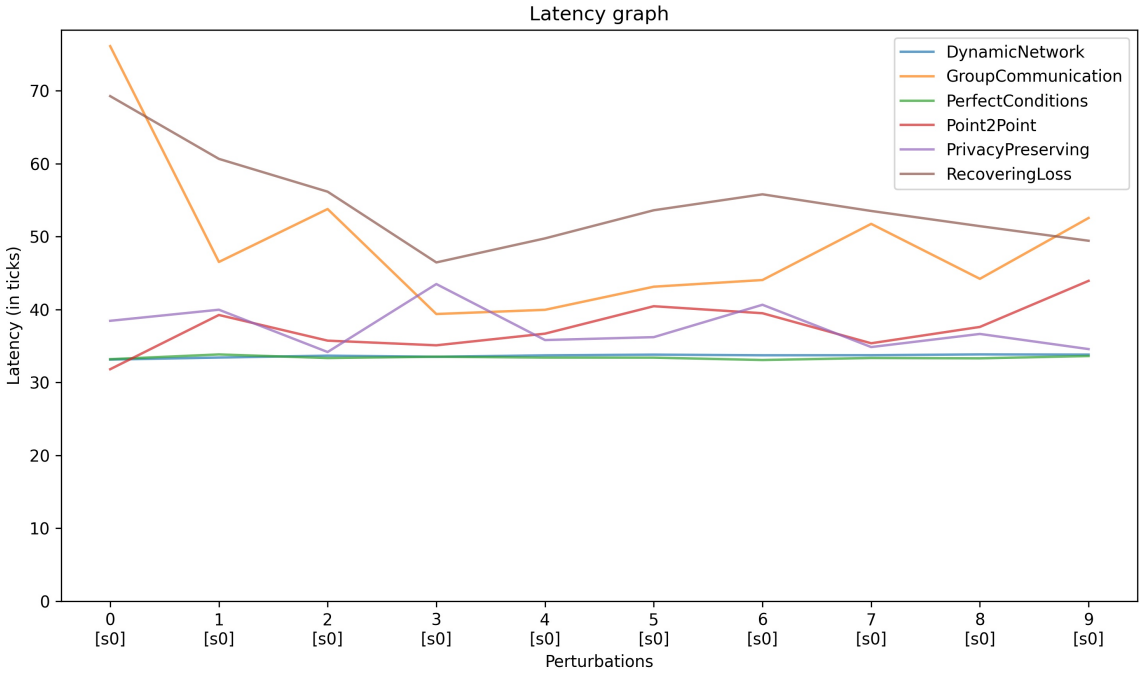

Fig. 5. Latency plots for scenario 1

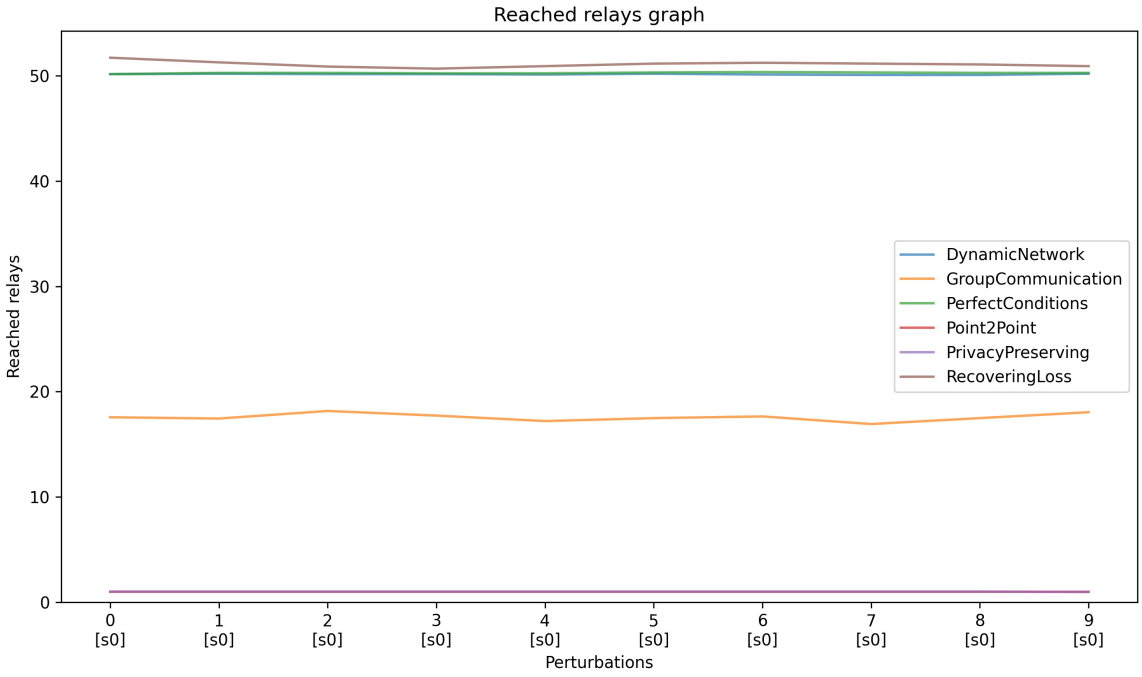

Fig. 6. Plot showing the number of reached relays by every perturbation in Scenario 1

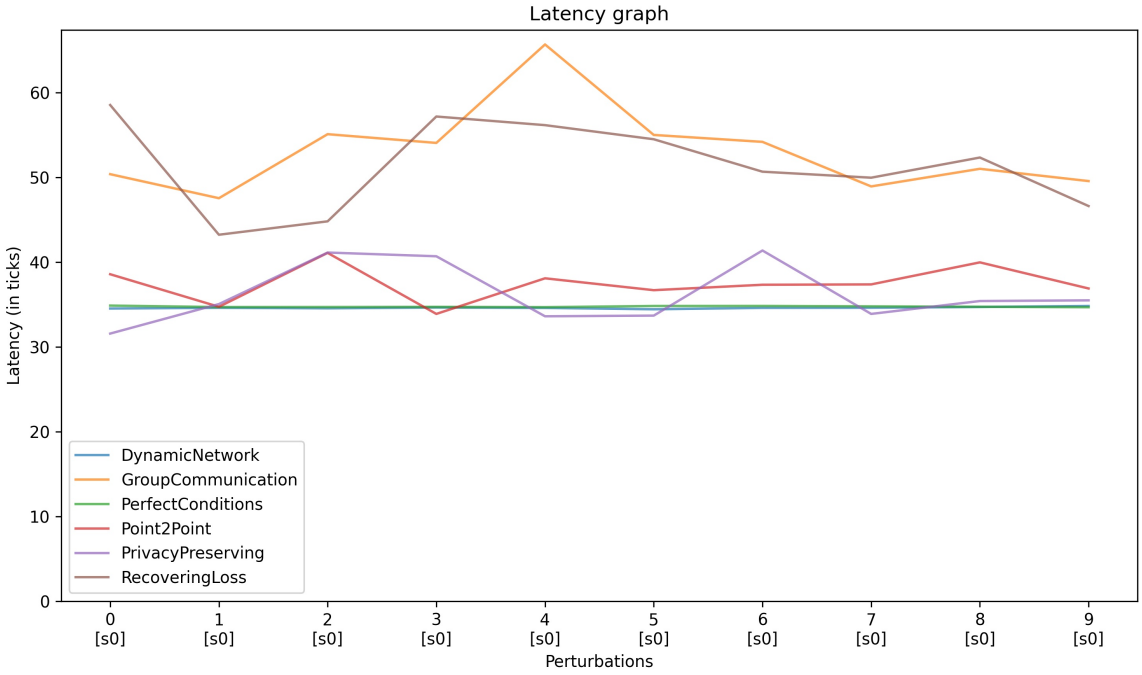

Fig. 7. Latency plots for scenario 2

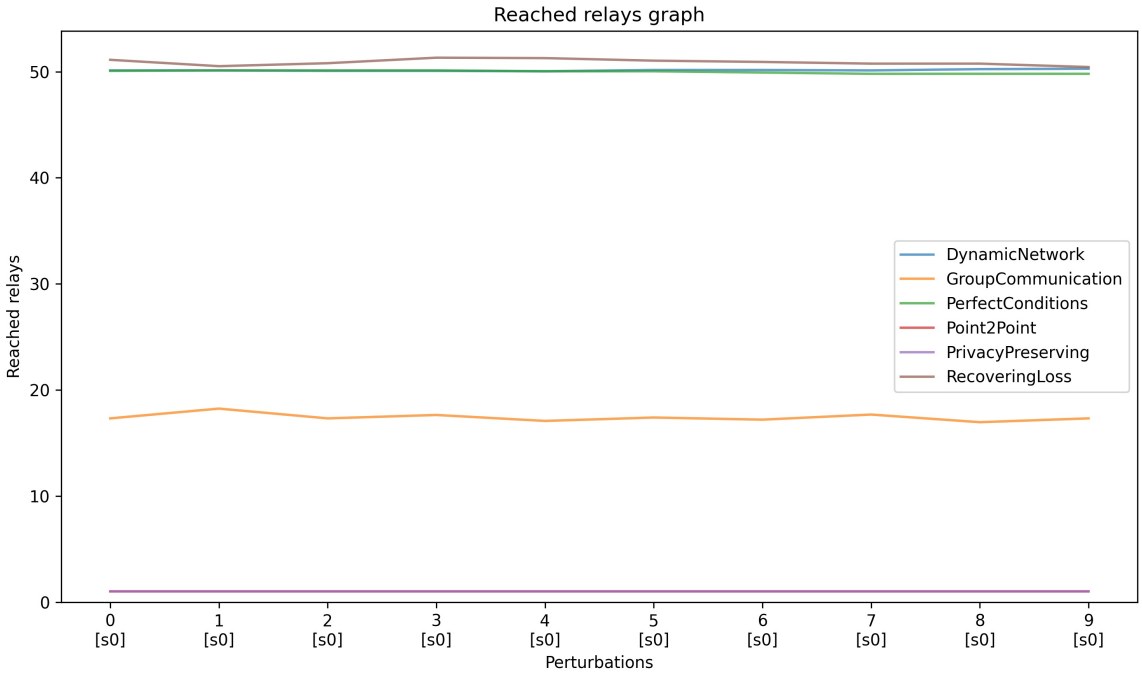

Fig. 8. Plot showing the number of reached relays by every perturbation in Scenario 2

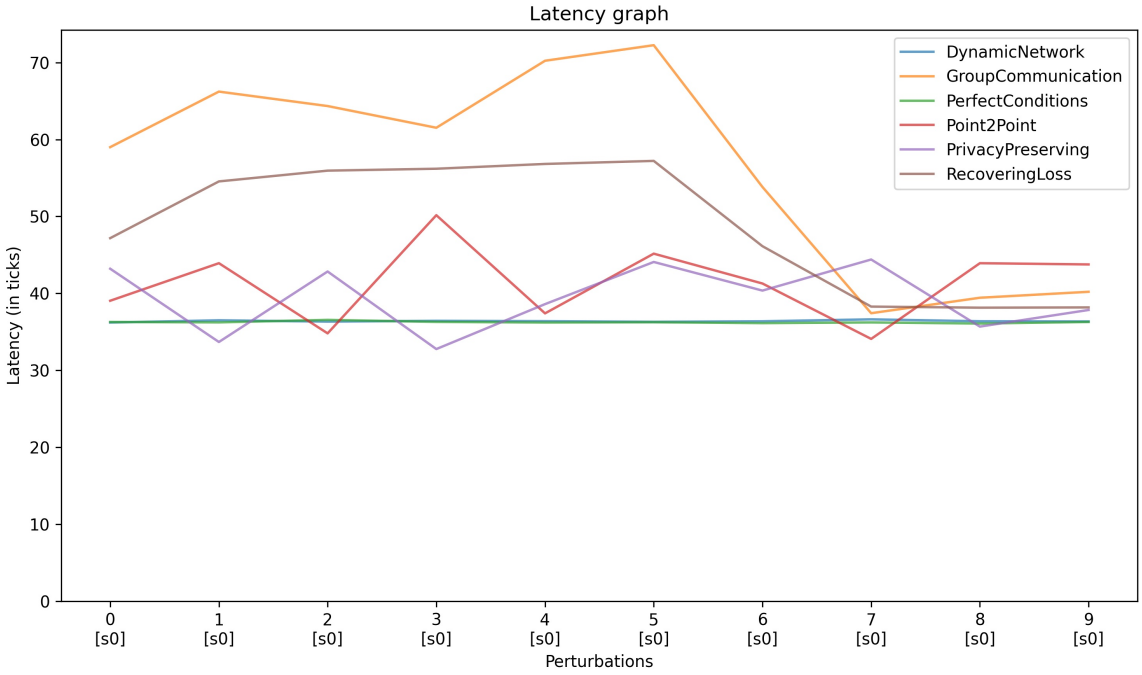

Fig. 9. Latency plots for scenario 3

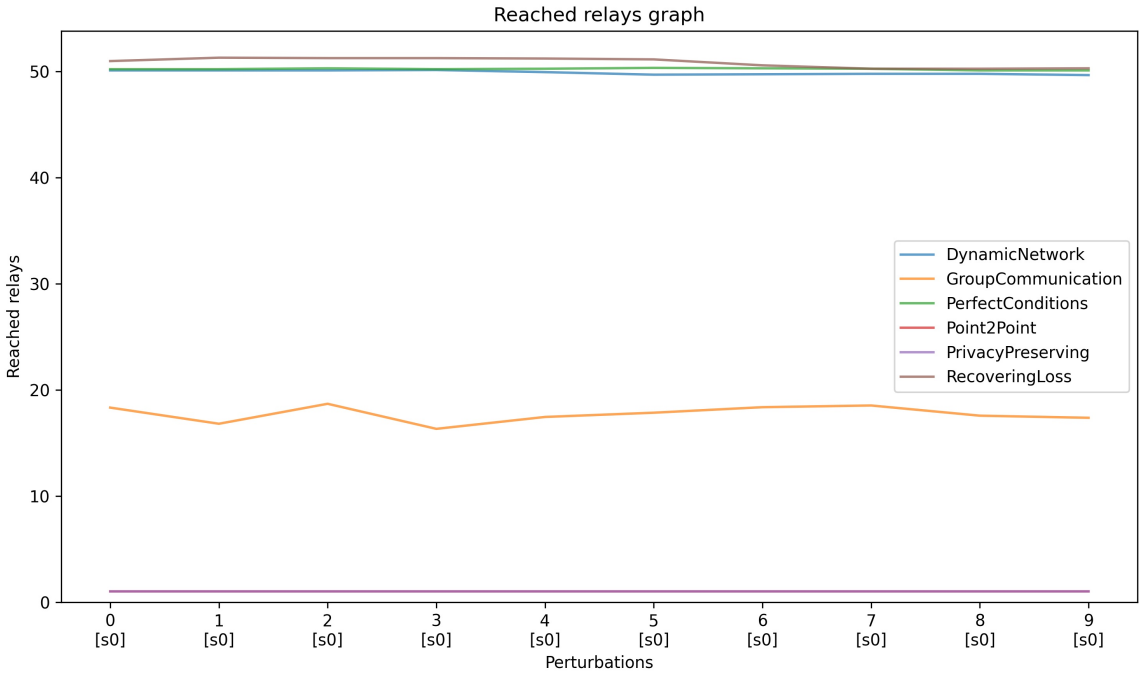

Fig. 10. Plot showing the number of reached relays by every perturbation in Scenario 3

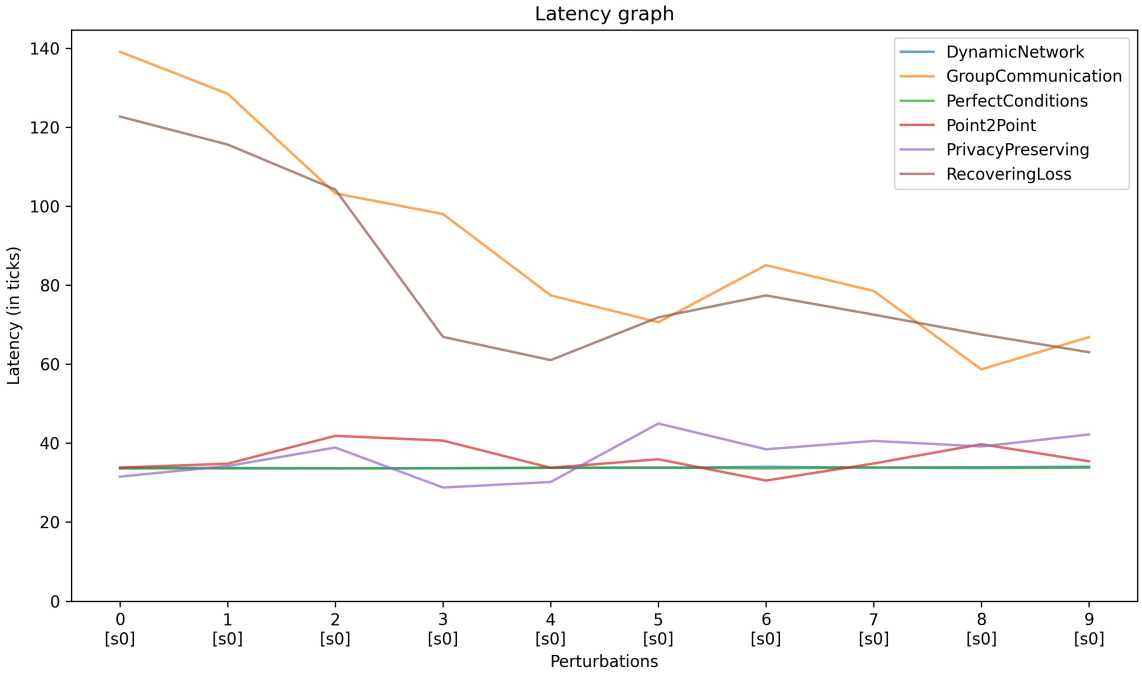

Fig. 11. Latency plots for scenario 4

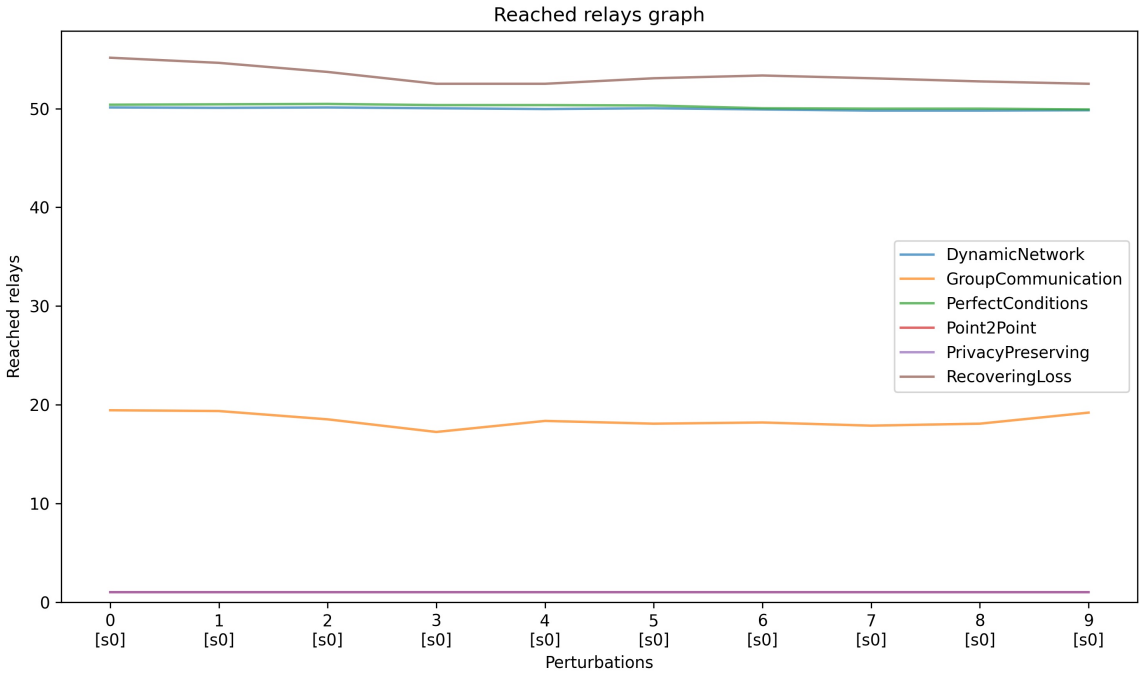

Fig. 12. Plot showing the number of reached relays by every perturbation in Scenario 4

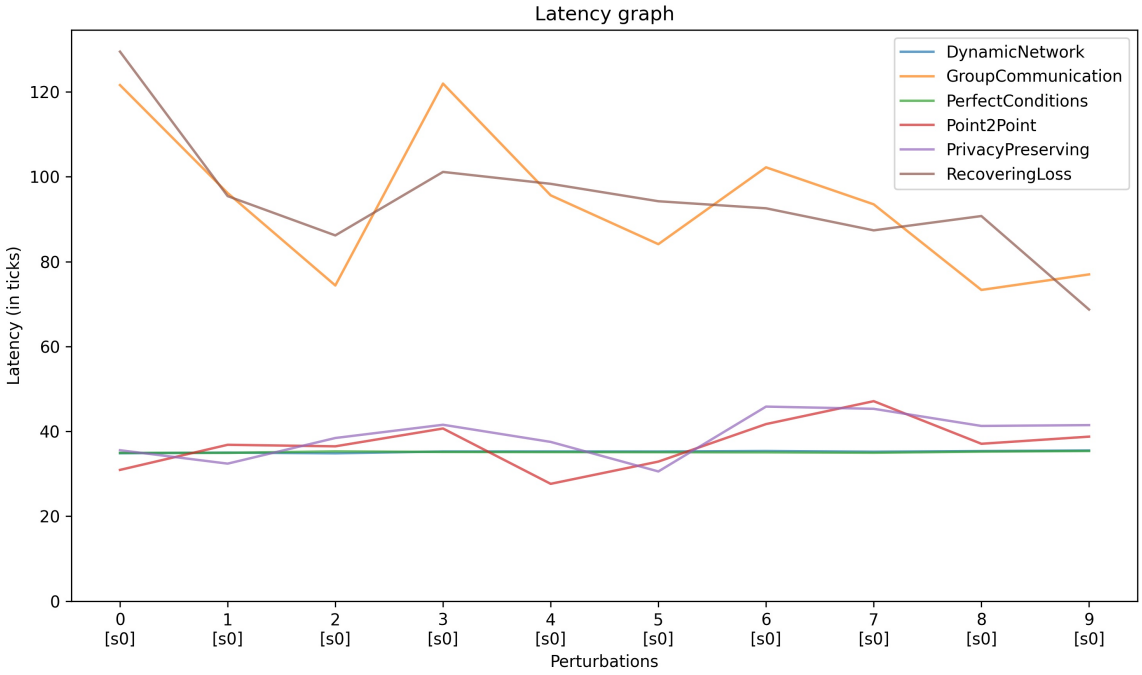

Fig. 13. Latency plots for scenario 5

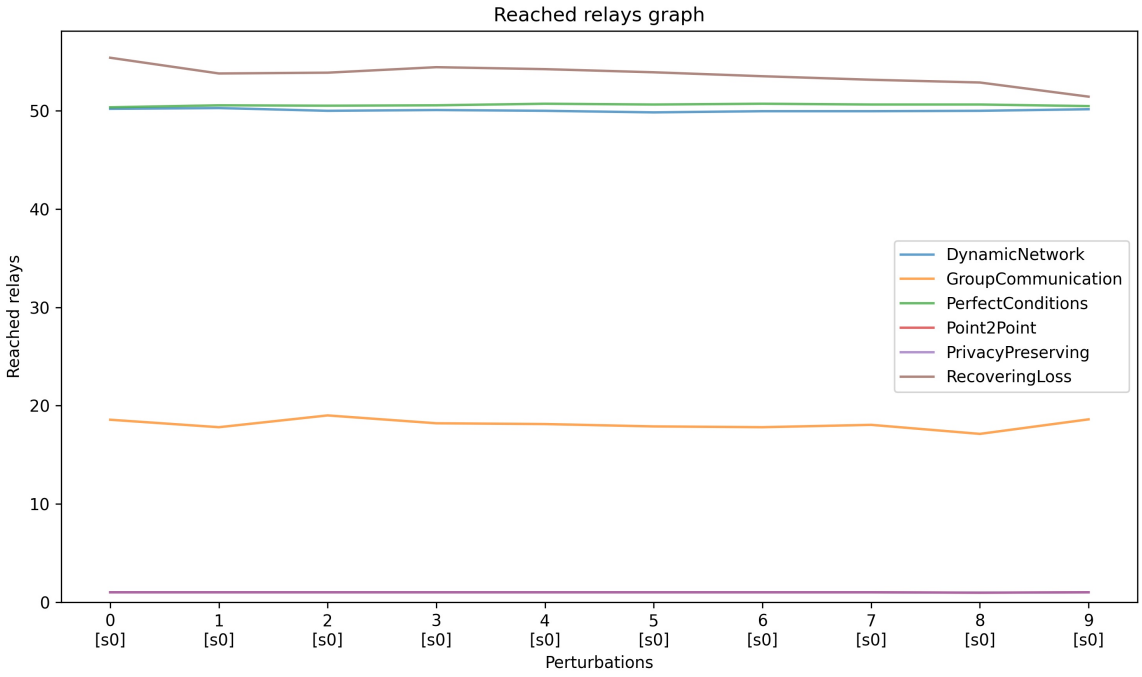

Fig. 14. Plot showing the number of reached relays by every perturbation in Scenario 5

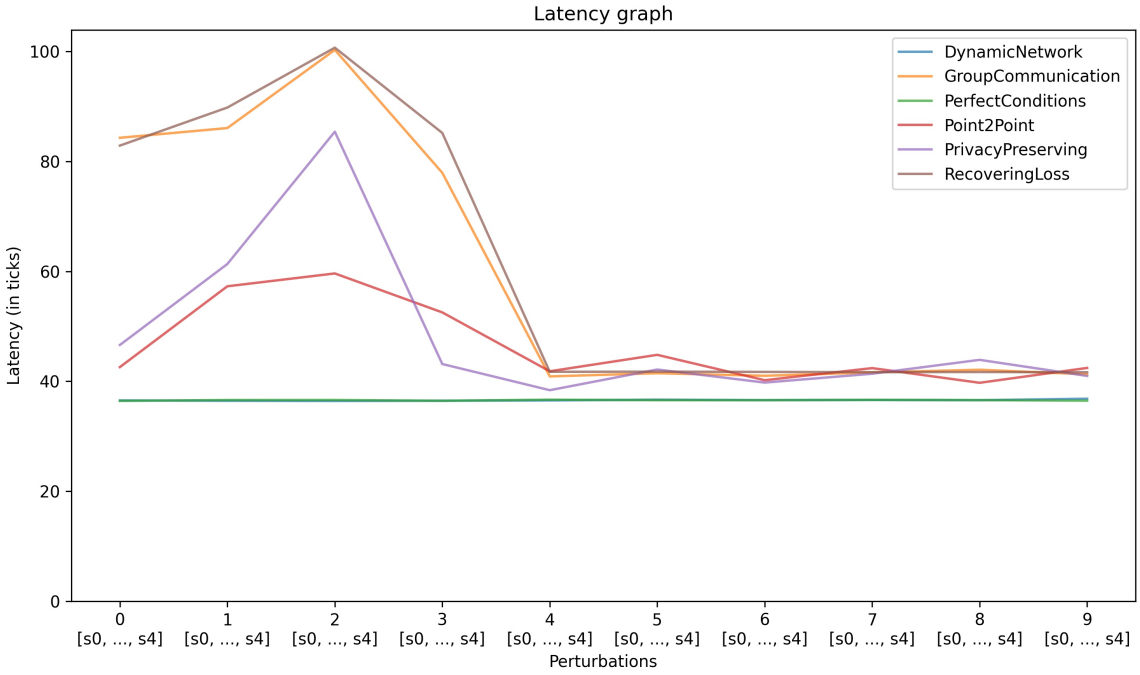

Fig. 15. Latency plots for scenario 6

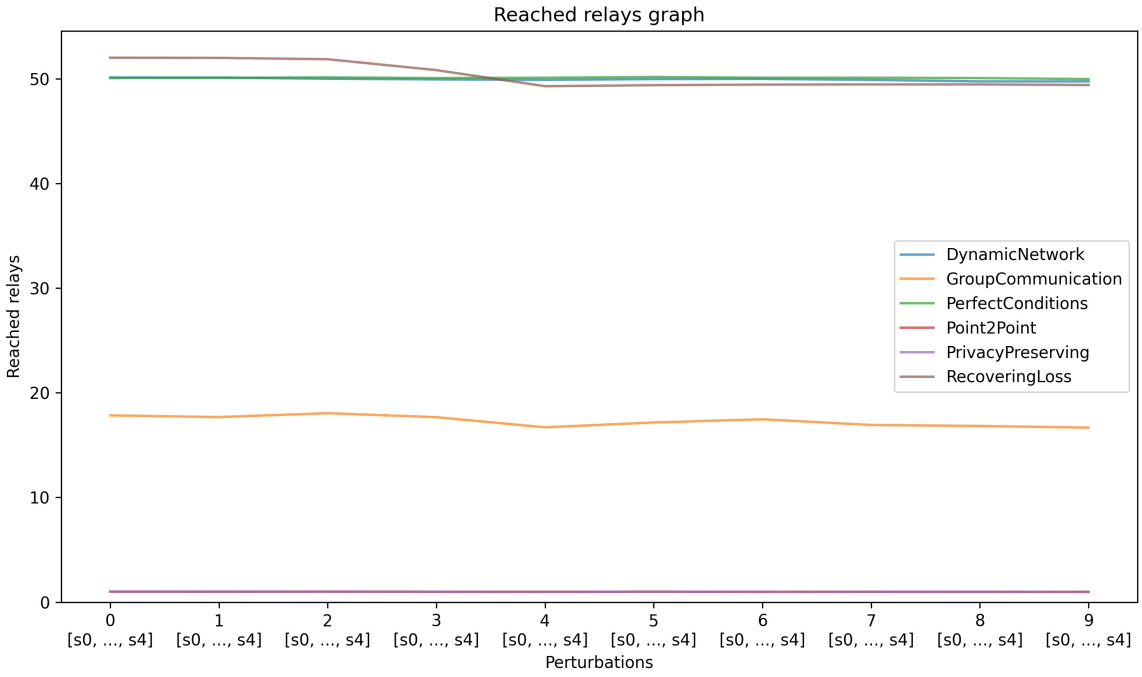

Fig. 16. Plot showing the number of reached relays by every perturbation in Scenario 6

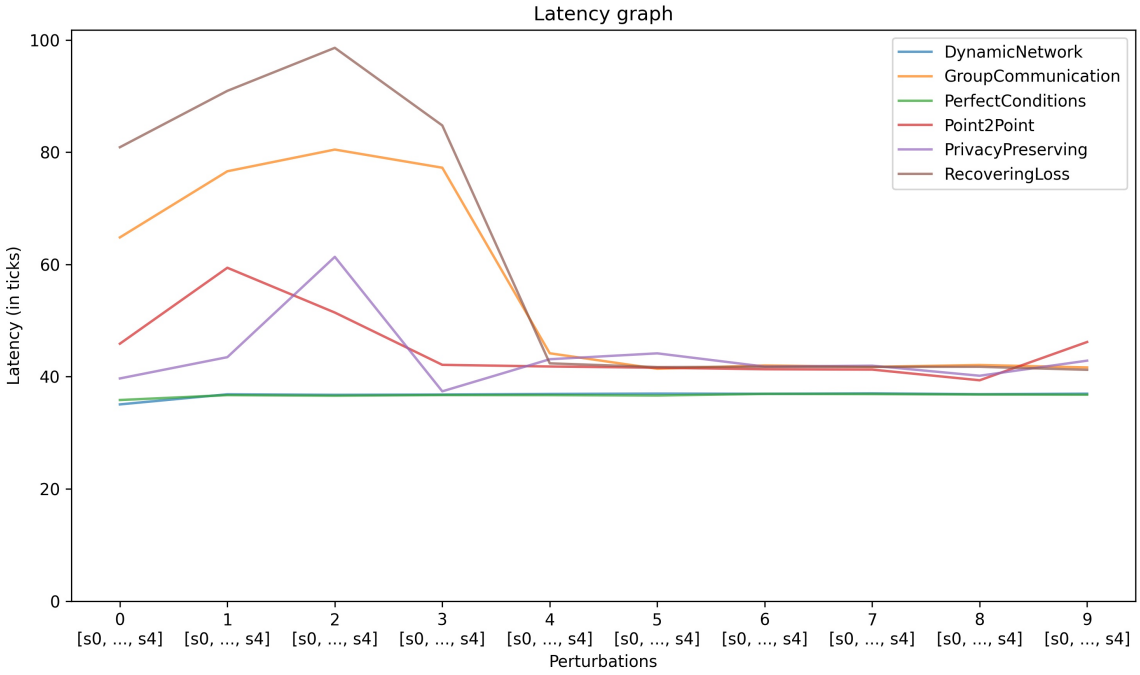

Fig. 17. Latency plots for scenario 7

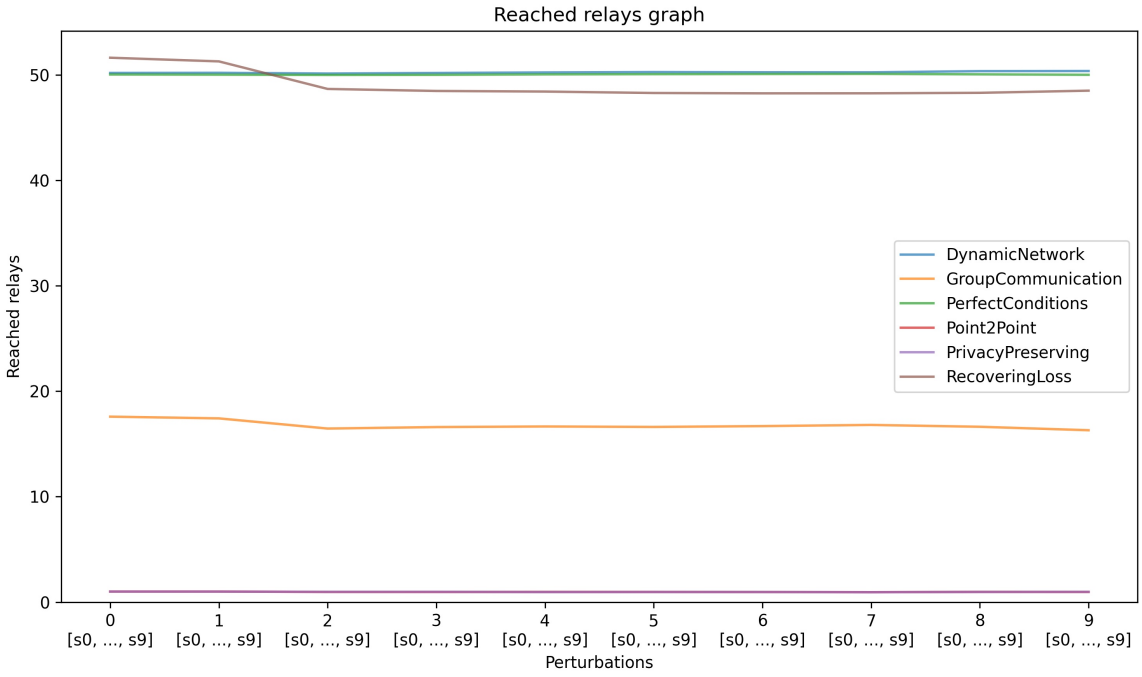

Fig. 18. Plot showing the number of reached relays by every perturbation in Scenario 7

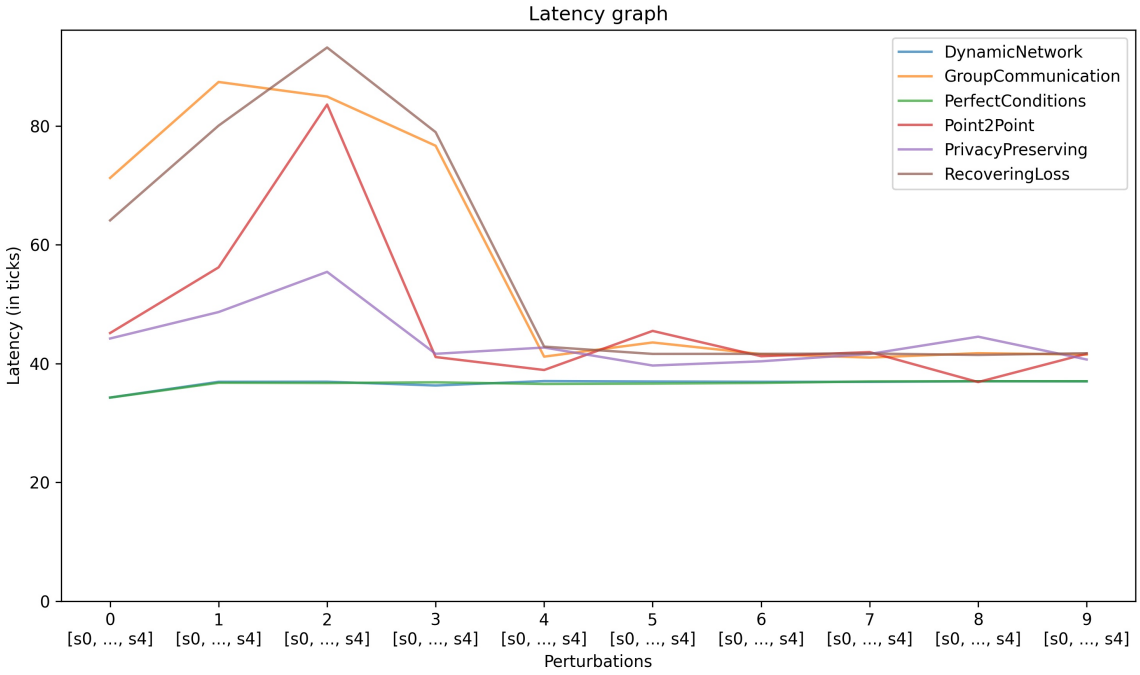

Fig. 19. Latency plots for scenario 8

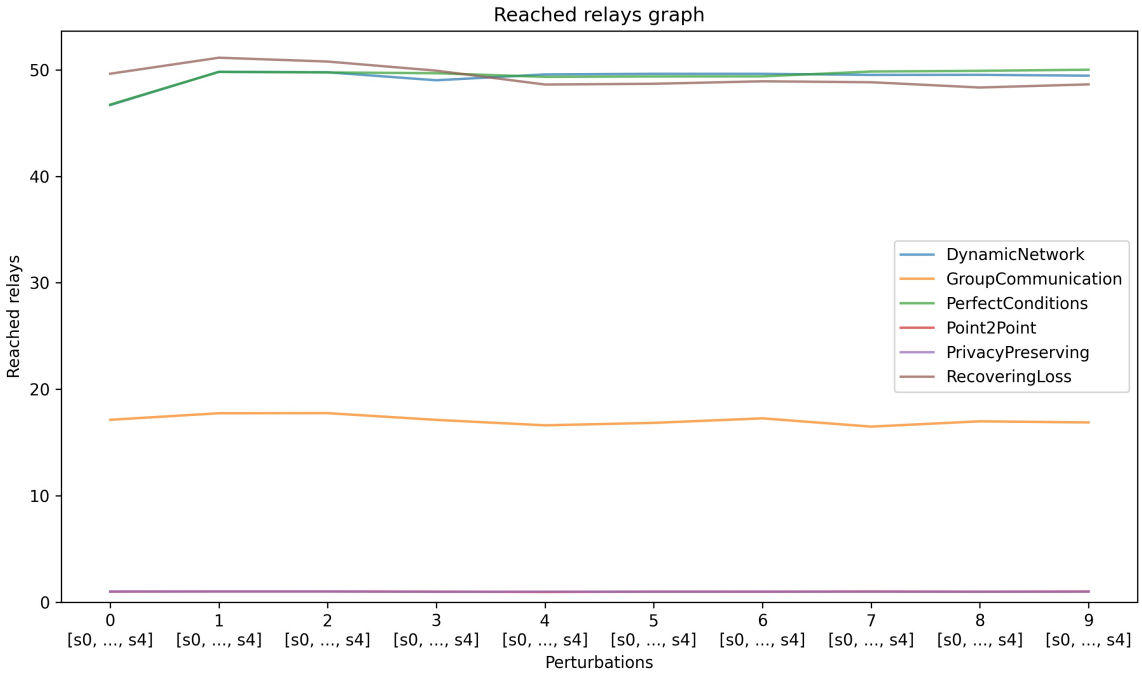

Fig. 20. Plot showing the number of reached relays by every perturbation in Scenario 8