Distributed Systems 2 - First assignment

Georgiana Bud
University of Trento
Trento, Italy
georgiana.bud@studenti.unitn.it

Massimiliano Fronza
University of Trento
Trento, Italy
massimiliano.fronza@studenti.unitn.it

Enrico Soprana
University of Trento
Trento, Italy
enrico.soprana@studenti.unitn.it

***Abstract*—**

***Index Terms*—distributed systems, protocol, broadcast, unicast, multicast, simulation**

I. Introduction

In this report, we propose our implementation of the broadcast-only algorithm presented in the article *A Broadcast-Only Communication Model Based on Replicated Append-Only logs* [1]. We decided to develop the third proposed variant of the protocol: the dynamic network case with recovery from loss based on append-only logs and a pull mechanism for missed perturbations. On top of the basic broadcast algorithm we also implemented the unicast and multicast functionalities. The implementation provided is written in the Java programming language, with the usage of Repast Simphony, an agent-based system, as a simulation and visualization framework.

In the next section, we will introduce the overall architecture of the system. Then, we will underline the main design choices taken, with a more detailed explanation of the implementation. Following, we will describe the design of the visual interface, with the visualization of the network and the configurable parameters of the simulation. Next, another section will explain the experimental results of the analysis we have conducted to understand the performance of the protocol and of our implementation. We will conclude with some notes on lessons learned and possible extension of the work.

II. Architecture

A. Layered architecture

From an architectural perspective, we decided to follow a modular approach and to split the simulation in 3 layers: physical, which abstracts away the wireless communication, logical, which implements the actual protocol, and the application layer, which we can use to inject messages and to more easily get statistics on the received ones.

- The **physical layer** is implemented in the *nodes* and *network* packages. The main agent is *Machine*, which represents the relay described in the article, on which the protocol runs. This agent is uniquely identified by an *Address*. The exchange of messages at the physical layer is implemented under the *network* package which simulates wireless-based communications. The implementation choices done for the network will be explained in more details in section III.
- The **logical/protocol layer** is implemented under the *protocol* package. The main class that holds the functionalities of the relays presented in the article is *RealProtocol*. It extends the parent class *Protocol*, which has more general methods to map an event type and its data with a corresponding handle function marked with a specific java annotation. *RealProtocol* sends broadcast messages and handles the various types of *Perturbations* that it receives. It holds a log of the complete history of *Solitons* per source, including itself. All the functionalities related to multicast and unicast are collected together in *MulticastProtocol* and *UnicastProtocol*, which are accessible from *RealProtocol* as fields. Different extensions of *Perturbation* were implemented to represent the different type of messages that are sent by the protocol. These extensions can be found in *protocol.messages* package. The data that they can hold is to be found under *protocol.messages.data*.
- The **Application** is under the *application* package. The *RealApplication* is the main agent here, and it is responsible for generating broadcast, unicast and multicast events when requested by the simulation; it is also the final destination of the messages being exchanged. This simulates a real application, in which a user would trigger these events that have to be managed at a protocol level.

B. Simulation

Every *Event* of the simulation is coordinated from the *simulation* package, with the help of agents under the *utility* package.

The *Events* that can happen are placed in the relative packages, according to their classification into *SimulationEvents*, *NetworkMessages*, *LocalEvents* or *ApplicationEvents*. The coordinator agent is *Oracle*, which implements the scheduled `step()` function of Repast. At every tick the Oracle calls the relevant `handle(Event)` functions of the relevant classes. These events can be:

- *SimulationEvents*, through which we can create/kill nodes with a certain probability and distribution;
- *ApplicationEvents*, through which we can compel an *Application* to generate messages of a certain type - broadcast, multicast or unicast;
- *LocalEvents*, through which the *Oracle* can remember to the *Protocol* that the time interval between two *ARQSoli-*

tons expired or that it decided to retry to broadcast at a later time.

- *NetworkMessage*: it simulates the reception of a message of a wave, at a physical level. It is exchanged by *Machine*, and it is used to draw network links in the visualization. It has a probability of being lost due a bad reception or due a collision.

All these *Events* are contained in a *Timestamp* and ordered according to this timestamp. Different schedulers take care of ordering the events of different types while the Oracle only has to pick the one with the earliest timestamp from the schedulers. The Oracle then delivers the event to the appropriate class for its handling

(e.g: In the case of an ApplicationEvent it will delivered to an Application, if it is a NetworkMessage or LocalEvent it will be delivered to a Protocol while in the case of a SimulationEvent it will be handled directly in Oracle).

III. Implementation choices

A. Data exchange

There are different data packets sent at the various layers. They have only been shortly mentioned above, and here we explain them into more detail. We have:

- *Perturbation*: it is the main agent exchanged at the protocol level, implemented as an abstract class which holds a *source* field. *Soliton*, *ARQSoliton*, and *ARQHistoryReply* inherit from it the source field. Besides the source, the agent *Soliton* holds as *ref* the sequence number of the given perturbation instance, and as value an (abstract) *ProtocolMsg*, which in turn holds an *ApplicationData*. The types of *ProtocolMsg* can be *BroadcastMsg*, *UnicastMsg* or *MulticastMsg*.
  - *Solitons* are forwarded and processed according to the functionalities indicated in the paper.
  - *ARQSoliton* is a perturbation with a source *Address* and the next expected reference for that address. We have decided to extend the implementation given in the article by adding a custom value for the source field, i.e. *null*. This is sent as the first message when a node joins the network, in order to request a general update on all the messages sent so far. It indicates that it does not know anyone, and it will trigger as a reply an *ARQHistoryReply*.
  - An *ARQHistoryReply* is triggered when a node receives an *ARQSoliton* with as source, *null* value. It contains all the messages of a source that the *RealProtocol* running on the receiving relay has. This is done to save time and not send every single message individually.

  We decided to instantiate the types of *Perturbation* as different agents, and then the *ProtocolMsg* contained in them as well, in order to have a smoother processing at the protocol level.
- *ApplicationData*: this abstract class is used as a tag to recognize the data which should be consumed by the Application. *Id* extends it and is the dummy data actually used in the simulation. *Id* holds a uniformly increasing integer id, used to keep track of the messages and their distribution in the network.

B. Protocol

We have already seen the message types that can be handled and sent by the protocol, and the fact that the RealProtocol performs the broadcast of *Perturbations*. In the following, we will explain in more detail the functionalities of *UnicastProtocol* and *MulticastProtocol*. These two agents are accessible from the *RealProtocol* and they separate the data and functionalities for point-to-point communication and group communication respectively.

The *UnicastProtocol* contains an updated list of all the peers which ever communicated with the protocol. This is equivalent to the list of *Addresses* that show up in the log of the *RealProtocol* underneath it, which is progressively updated during network discovery. The list of peers is used to perform an informed unicast (i.e. a direct message can only be sent to a known node). Moreover, only the peers that are alive can be the destination of a unicast. Unicast messages are broadcasted by calling the `sendBroadcastMsg(Solitons)` of the underneath *RealProtocol*. When a *Soliton* holding a *UnicastMsg* is received, this agent checks if it is the final destination of the message, and handles it accordingly.

For what concerns the *MulticastProtocol*, it holds a list of topics to which a node is subscribed. At initialization, a random number of topics, from 1 up to 5 different ones, is chosen and assigned to every *MulticastProtocol*. The subscriptions can be changed at runtime. The handling of incoming and outgoing *MulticastMsgs* is done in a similar way as in the *UnicastProtocol*.

Another modification we did to the implementation specified in the paper, was to change the reply of a node to an *ARQSoliton* when it has more than one new message for the requested source. The proposed algorithm in Relay_III was sending only one newer message, whereas we decided to send all of them, to make sure that the requesting node was up to date faster.

C. Wireless Communication

Our implementation uses a wireless network as a network model with randomly arranged nodes, all operating on the same channel.

1) Distance: As the size of the square simulated is normalized to 1x1, to regulate the behaviour wrt. distance, we can regulate the propagation speed by setting it to the ratio $speedOfLight/desiredSize$. For this reason, the default propagation speed used in the simulation represents the simulation of a square which is 250x250m.

2) Packet loss: One of the issues related with the decision of implementing a wireless network is how to relate the distance between two nodes and the packet loss between them. There are many models that can be used to model the behavior of packet loss in a wireless network depending on the type of radio technology used and the environment you want to simulate. We have chosen to model the packet loss by first modeling the path loss using the log-distance path loss model [2], which uses the following formula for the strength of the signal received at the destination:

$$PL(d) = PL_0 \ + \ 10\gamma \ \log_{10} \frac{d}{d_0} \ + \ X_g$$

The parameters can be described as follows:

d_0	The reference distance. In the simulation parameters is called *NORMAL_DIST*. As this is linked with the screen size (which is normalized to 1) this can be interpreted as a percentage of the screen.
PL_0	The strength of the signal at the reference distance d_0.
γ	The path loss exponent which represents how much obstruction/loss there is per unit of space.
X_g	A Gaussian random variable with zero mean used to reflect the fading of the signal. This variable can be adjusted to change the amount of variance there is in the behaviour of the model given a distance. In the simulation *VAR_NOISE*.

As simulating the packet loss of broad set of environments was not the goal of our endeavour we decided to fix γ to 2.6, a value which is usually used to simulate an office environment with soft partitions.

After getting the result of this model we can then apply another model [2], which models the PRR (Packet Reception Rate) given the path loss. Among the ones proposed in the cited paper we decided to use the one proposed for the modulation technique NCFSK which uses the following formula:

$$p(d) = \left(1 - \frac{1}{2} e^{\frac{-y(d)}{2}}\right)^{8 \ packetSize}$$

with

$$y(d) = P_t - PL(d) - P_n$$

As PL_0, P_t, P_n are all constants they were all aggregated in the simulation parameter *SNR*.

The choice of this model is not really important as the behaviour between different techniques is similar and we are not interested in simulating a specific radio technology.

Another parameter *CUT_OFF* was added ignore all the nodes after a multiple of *NORMAL_DIST*, to reduce the number of nodes considered for the path loss model and to put a limit to the distance at which a transmitting node can cause interference for another.

3) Channel capacity/collisions: The channel capacity is modeled though the time it takes to transmit a packet (which has a fixed size of 1000bytes) and the bitrate which can be chosen through the interface. This acts as a limit to the amount of data which is possible to move through the network. In fact, if the nodes try to send more data than allowed by the bitrate some of the packets will inevitably be lost due to collisions, which will render the colliding packets not readable by the receiving nodes. It is important to note that, as the nodes are not coordinated in any way, there will be some collision even if the limit set by the bitrate is not reached.

In our model a collision happens if:

- Two partially overlapping messages are received by the same node[1].
- A message is received while the receiving node was transmitting.

As our physical layer is quite simple, it does not implement any recovery process for lost packets or collisions. We decided to keep it that way as this is in part already addressed by the ARQ requests of the implemented protocol. Even if there is no recovery process implemented, some precautions are adopted to keep the number of collisions to a minimum.

This was done by using two techniques:

1) The node listens for activity on the network and only when there are no messages currently being transmitted (or at least it appears so due the delay due the distance between the two nodes) the node actually transmits.
2) As per our model, the node cannot detect when a collision is taking place (and thus inform other nodes). This means that it is not possible to use an adaptive behaviour which led us to add a simple random delay uniformly distributed in [*PROCESSING_TIME*, *PROCESSING_TIME+MAX_RANDOM_DELAY*]. This was done to prevent the nodes from synchronizing their re-broadcasting of a message as this would mean that all the messages sent would collide with extremely high probability as the delay added by the distance between the nodes is too small compared to the transmission time of a packet.

As it can be quite difficult to guess the behaviour of this model, a jupyter notebook was provided in `./scripts/notebook.ipynb` to more easily see the effect of these variables. The chosen model is summarized in Fig. 1.

IV. Visualization

The visualization-related part of our project has its main management performed by the *DisplayManager* class. This is initialized in the *Oracle*, and its main function, *graphic_propagation*, is called after the whole step computation.

For each step, we first clear the space form all graphic elements previously created by removing them from the context, or the network in case of the edges; then we proceed to analyze the passed *NetworkMessage*, dividing the outcomes in this way:

- if the destination of this packet is dead, it will disappear from the space and there won't be any graphic event;
- message not received: drawing of a red edge going from the source to the destination of the message, and a red

[1]As the transmission time of the packets is the same (they all have the same bitrate and packet size), this is done by checking that the distance in time of the reception of the last bit of the two messages is greater than the transmission time

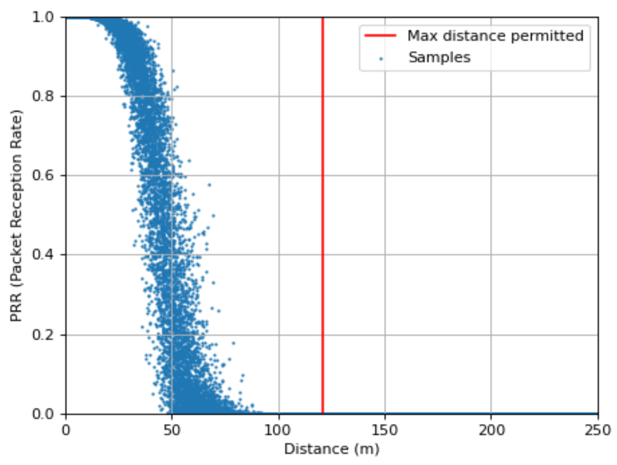

Figure 1. PRR (Packet Reception Rate) vs distance

cross on the latter. There will not be any other shape on screen;

- message of *Soliton* instance: if the node (re-)broadcasting the information is still alive, a green edge will go from the current source to the destination receiving in that moment. In this case, if it is the first time the destination is receiving the message, a circle centered on the original source contained in the Soliton will also appear, to simulate the growing propagation of the wave originating from the original node whose message is being transmitted. The color of this circle depends on the type of protocol message: green in case of Broadcast, Orange for Multicast and Pink for Unicast. In this way, circles will only be shown when nodes actually process the message, and the overhead, duplicated messages that will be received because of retransmission will pass unnoticed.
- message of *ARQSoliton* instance: if the node re-broadcasting the information is still alive, an orange edge will go from source to destination;
- message of *ARQHistoryReply* instance: as in the previous case, but with a magenta edge;
- if the original source of the message is still alive, there will be a light-blue triangle on that node;

Several classes were created to represent edges, circles, crosses and the origin triangles, each one as a custom element. To change colors, weights and radius values in a dynamic way for edges and circles, we created two style classes that implemented the *EdgeStyleOGL2D* interface. At the beginning, we wanted to use *DefaultEdgeStyle2D*, which then turned out to be used in Repast Simphony 2.7, but deprecated since the 2.8 version. These classes call methods from *DisplayManager* to get the proper values of the objects they represent, and since we kept having problems in making Repast use them instead of the *DefaultEditedStyleData2D* counterparts, we had to manually edit the */Ds2Assignment1.rs/repast.simphony.action.display_1.xml* file to make it work. Also, edges needed a class implementing an *EdgeCreator* to instantiate them. One additional note about circles is that we wrote an auxiliary function to compute the euclidean distance between source and destination nodes whenever it was necessary to obtain the radius of the object.
For crosses, origin triangles and the nodes themselves the situation was easier, because their colors and sizes were not dynamic, so to describe their properties the *DefaultEditedStyleData2D* was enough. Style files for these objects were stored in the */Ds2Assignment1.rs/styles/* folder.
Nodes are represented as small blue circles, under which there are labels identifying them.
Speaking of parameters, only DISPLAY_SIZE affects the visual part of the simulation, by re-sizing the space dimensions.

V. Parameters

Our simulation is highly configurable. Here we provide a list of all the parameters that can be set and changed in order to obtain different behaviours, and a quick explanation. Please note that all the parameters starting with MEAN and VAR are doubles, and that they refer to the parameters of a normal distribution, which indicates how the corresponding events are spaced in time. The time is meant in (simulation) seconds, and it is not directly related to the tick count. The numbers at the beginning of the parameter, shown on the graphical interface, are only a trick to group the parameters together and to display them in the desired order.

OUTPUT_PATH This is the path of the folder in which data is written. It is a string, with default value results.

ARQnull_OPTIMIZATION It holds a boolean, which enables or disables our implementation with null ARQs sent when a node joins the network. The default is true.

HANDLE_ARQ_OPTIMIZATION It is as well a boolean, to enable or disable the optimized sending of newer messages, when an older message is requested and the replying relay has newer messages for that source.

MEAN_CREATE - It is the average time between the creation of two nodes. With 0, we disable the creation of nodes after that the network has reach the initial size. To start with a static network, we have let 0 as default.

VAR_CREATE - Standard deviation of the time between the creation of two nodes. The default value is 0.

MEAN_KILL - Average time between the removal of two nodes. The default is 0, which means that it is disabled, for a more static network.

VAR_KILL - Standard deviation of the time between the removal of two nodes. The default value is 0.

MEAN_BROADCAST_EVENT - Average time between the creation of two broadcasted events. With 0 it is disabled. The default value is 3.0.

VAR_BROADCAST_EVENT - It is the standard deviation of the time between the creation of two broadcasted events. The default value is 0.5.

MEAN_MULTICAST_EVENT - As the mean for broadcast events, but for multicast events. By default, it is disabled (0).

VAR_MULTICAST_EVENT - As above, with default value 0.

MEAN_UNICAST_EVENT - As above, but for multicast events. By default, it is disabled (0).

VAR_UNICAST_EVENT - As above.

INITIAL_NODE_SIZE - Size of the network at time 0, when broadcast can start. It is of type integer, with a default value of 25. We discourage smaller values when using a large display, because nodes may be too far away for the broadcast to reach them.

MAX_NODE_SIZE - Maximum number of nodes allowed in the network. It is of type integer, with default value of 100. TOPIC_NUM - Integer, holds the maximum number of topics from which a node can pick when subscribing.

STOP_TIME - It is the time at which to automatically stop the simulation. It is of type double, with default value 5000.

PROPAGATION_SPEED - Speed of propagation of the perturbation. It hold by default a value of 1198837.

BITRATE - Bitrate in kbps. It is of type double, with a default value of 54000 (54Mbps). PACKET_SIZE/BITRATE is the actual needed time to transmit a single packet.

PROCESSING_DELAY - Delay needed by a node to process and then retransmit a packet. It is of type double, with a default value of 0.01.

MAX_RANDOM_DELAY - Maximum random delay before sending, according to a uniform distribution. The default value is 0.15. It indicates delays that may happen on a PS.

NORMAL_DIST - Reference distance for packet loss model. It is of type double, with a default value of 0.22.

CUT_OFF - Cut off distance in NORMAL_DIST*x after which no interference or reception can happen. It is of type double, with a default value of 2.2.

SNR - Base signal to noise ratio that determines the probability of a packet of being received. It is of type double with a default value of 15.

VAR_NOISE - Standard deviation of the random noise. It is of type double with a default value of 1.3.

PACKET_SIZE - Size of the packets, this influences both transmission time and packet loss. It is an integer, by default set to 1000.

DEBUG_COLLISIONS - Disables or enables the output for detected collisions. It is a boolean set by default to true.

ARQ_INTERVAL - Interval after which a new ARQ is triggered. It is of type double, with a default value of 30 seconds, one every 10 broadcast messages sent.

DISPLAY_SIZE - Size of the display, used by the visualization tool. The type is double and the default value is 10.

VI. Analysis

In order to perform the following analysis, we collected data from the simulation while it was running different times, with different parameters. We used a *Logger* agent in order to print, both to screen and to different files, information about the events, exchange of messages and message processing. We parsed the output logs to obtain the relevant information for each analysis by using bash commands. Further processing of output was done in Python, and graphs were generated both in Python and in Excel.

A. Latency and distribution without bitrate limiting

We made this analysis as to have a reference point of how the protocol behaves without external limitations. This is useful as we can see the point after which the protocol starts to behave abnormally due the limitations imposed. To do this we used as bitrate 54Mbps.

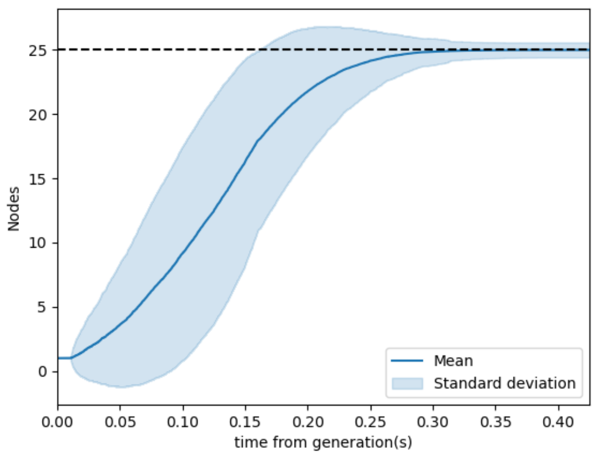

Figure 2. Average number of nodes receiving a message after a given time

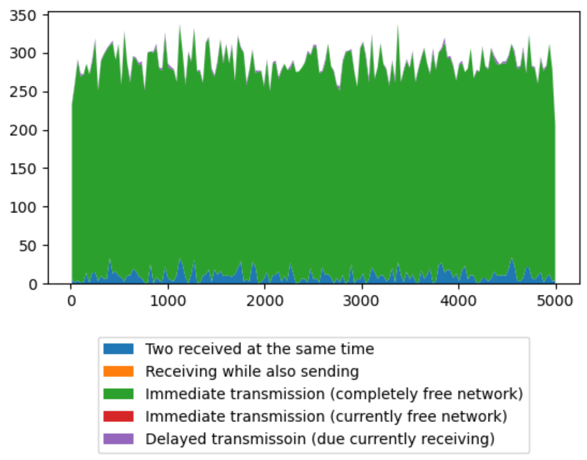

Figure 3. Average number of “collisions” for each type

As we can see from 2, in this configuration the messages can reach all the nodes within 0.4 seconds. This is in line with what we expected given that the random delay added of the broadcast plays a major role in the time needed for the message to propagate when not lost due bad reception or collision. In fact, we can see the 0.4s result can be justified by considering that at each broadcast we add a uniform random delay of [.01s, 0.16s] (and thus a mean random delay of 0.085s) and each broadcast can reliably travel about fifth of the width of the network. This means that it will take about 5-6 hops for the message to reach all nodes and on average each hop will take 0.085s adding to a total of 0.425s or 0.5s. This value is dependant on how the nodes are arranged and on the "luck" each broadcast has wrt. the possible collisions and the variable reception rate due the random noise.

As we can see from 3 with a bitrate 54000kbps collisions are and most messages can be send immediately without having to wait for the channel to be free.

B. *Latency and distribution vs bitrate*

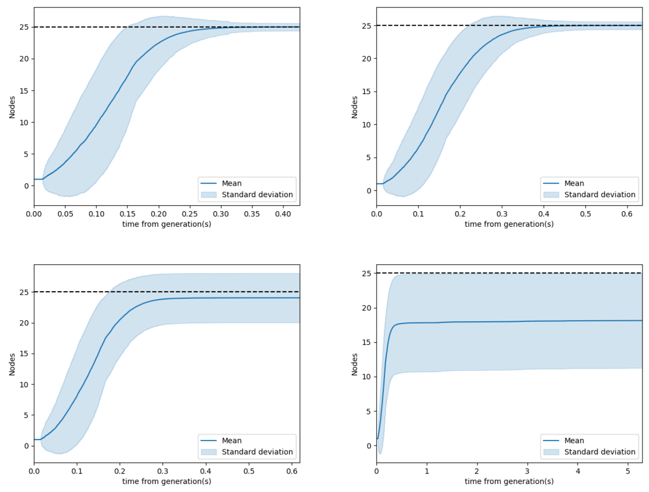

Figure 4. Average number of nodes receiving a message after a given time vs bitrate (in order 2100kbps, 1500kbps, 1400kbps, 400kbps)

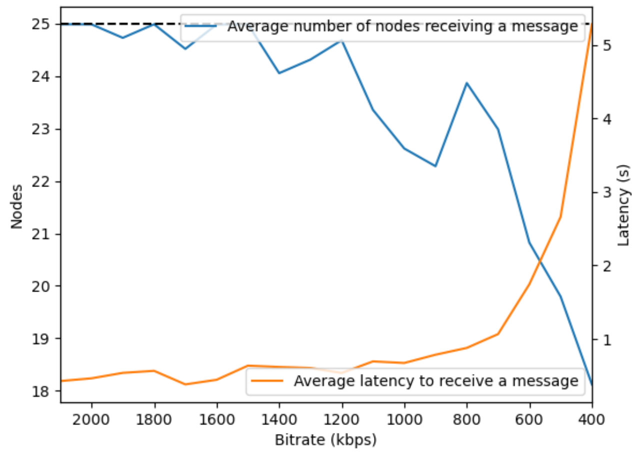

Figure 5. Average number of nodes receiving a message and latency vs bitrate

An analysis we wanted to make was seeing the minimum amount of bandwidth that is needed for the protocol to work correctly and with an acceptable latency. To do this we ran and logged a simulation with a static network of size 25 and generated a broadcast message every, on average, 3 seconds to create some load in the network.

The bitrate was then varied in increments of 100kbps. We can see that under a bitrate of 2100kbps the delay needed before reaching all the nodes starts to increase and that under a bitrate of 1500kbps the protocol can no longer reliably deliver the messages to all nodes[2] while at the same time having an ever increasing latency. This inability to deliver the messages can be attributed to the protocol not being able to withstand the large amount of collisions and due the increasing number of messages to be sent (to recover the lost packets) which makes a successful delivery of a message more and more improbable.

This values are influenced by the random topology assumed by the network, the random values chosen by the path loss model and the scheduling of the broadcasts and should thus not be considered as exact values at which a specific change happens but more as a guidance on where depending on the conditions the change can be expected.

Some images of the behaviour can be seen at Fig.4

The general behaviour can instead be seen at Fig. 5. We can see that the latency has a sharp increase after 700kbps due the network no longer being able to handle the load, while we can see the average number of nodes receiving slowly dropping after 1400kbps

C. *"Collisions" vs bitrate*

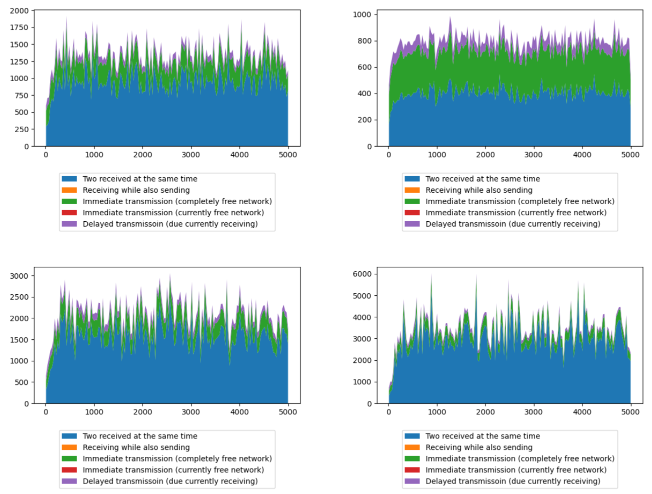

Figure 6. Collisions per 25s vs timestamp

As in our model the capacity model is tightly connected with the occurrences of collisions and delays due an occupied channel, we wanted also to log and show their behaviour to see how this correlates with the results of the previous test.

As we can see from 6, most of the collisions are due two

[2]Sometimes some nodes might still not receive a message but it can be attributed due the packet loss due the distance and thus the random "topology" of the network which leaves a node isolated and not the bitrate

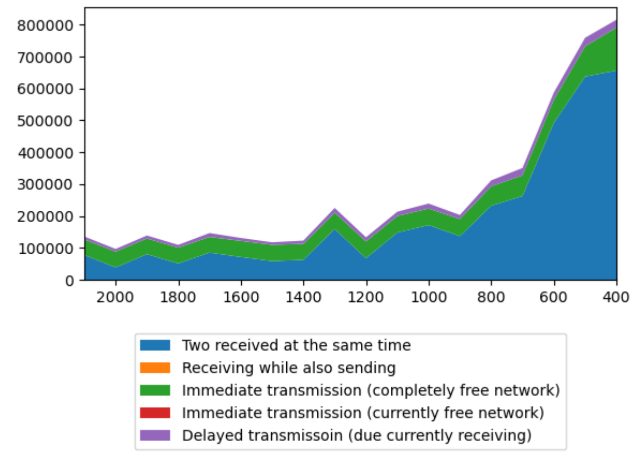

Figure 7. Number of collisions of a given type vs bitrate

packets being received at the same time, while the are next to none "receiving while sending" collision (with 2 total occurrences with 600kbps, 800kbps, 1000kbps and 4 total occurrences with 1300kbps).

As we can see from 7 the number of collisions has a sharp increase under 700kbps after which, like hypothesised in VI-B, the number of collisions becomes unmanageable and this is reflected in the latency and in the percentage of nodes which receive the message.

D. Message recovery time in dynamic networks

In order to verify the correctness and efficiency of our optimized implementation, we conducted an analysis on the recovery time of older messages when a node joins the network.

Three simulations were conducted, with the same parameters for the WiFi transmission and for node creation: the default parameters were used, with a channel bandwidth of 54Mbps, 25 initial nodes, static network (no crashes), with node creation every 250 seconds. This means that 19 nodes were created during the simulation, which run for about 5000 seconds. The ARQnull_OPTIMIZATION and HANDLE_ARQ_OPTIMIZATION were set differently, as explained in the following paragraphs.

In the first simulation, we tried out our implementation of both optimizations, in order to check that the expected behaviour of an immediate recovery was indeed satisfied. In this implementation, a new node joining the network sends an immediate null ARQ request, asking for all the messages that the neighbours have. In fact, the percentage of message recovered is of 100%, with an average time of 0.12 seconds, as you can see in the upper plot in Fig.8

In the second, the ARQnull_OPTIMIZATION parameter was set to false, and the HANDLE_ARQ_OPTIMIZATION to true. The expected behaviour in this case is that a node will incrementally ask for old messages of a source, after that a

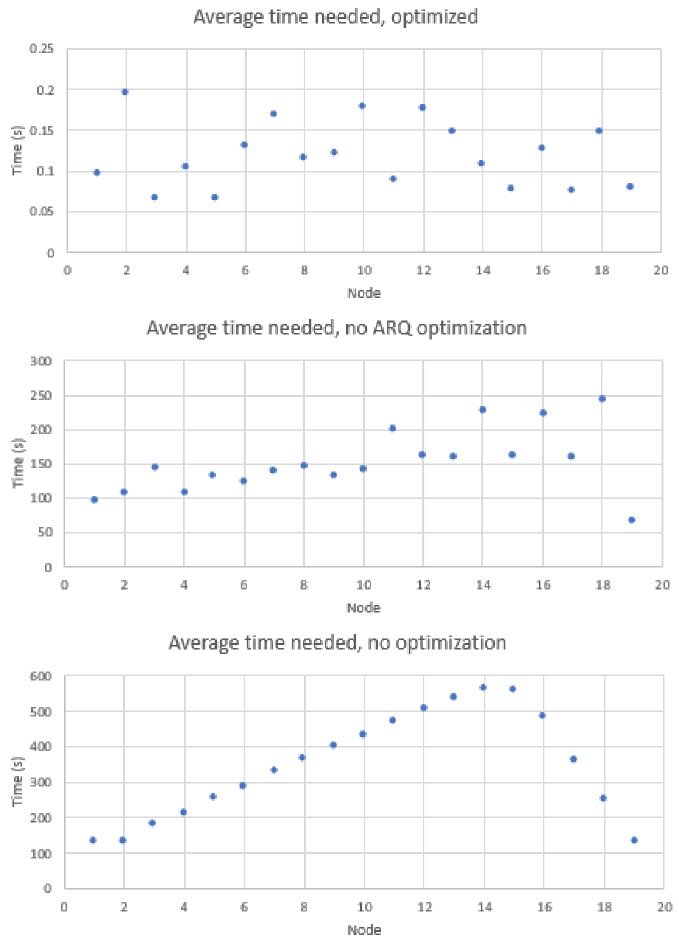

Figure 8. Average time needed by the new nodes to recover old messages. In the order 1: optimized; 2: no ARQ optimization, 3: no optimization at all, plain implementation of paper.

message from that source was received. Our implementation of the protocol, in fact, adds a new source to the known peers as soon as a message from it is received, even if the message is not in order. This way, the message is not logged nor processed, but the node knows that a certain source exists and it will ask for its first message. The second optimization parameter represents the optimization of the Relay_III algorithm, where the reply for an ARQ requesting an older message provides also the newer messages than the one requested, if any. As we see from the corresponding line in Fig.10, 17 out of 19 nodes manage to recover almost 100% of the old messages. The last 2 nodes joining the network instead were not able to receive all the messages in the time that they have before the simulation ended. In fact, this implementation required to run the simulation for almost 2 hours, and it did not arrive to 5000 seconds, thus the drop in the central graph for node 19. In this case the time for recovery drastically increases, with a mean of 153.02 seconds, and with peaks up to 244 seconds. This can be seen in the second plot in Fig.8 and in the orange plot of Fig.9.

The last case is that of no optimization at all, with both

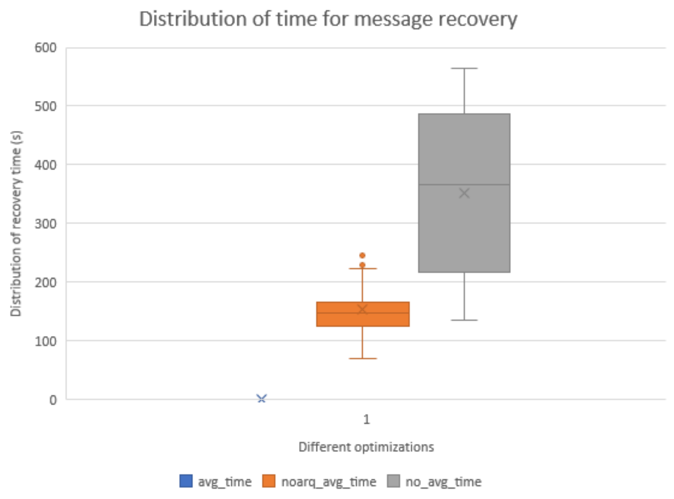

Figure 9. Comparison of distribution of average time needed to recover a message.

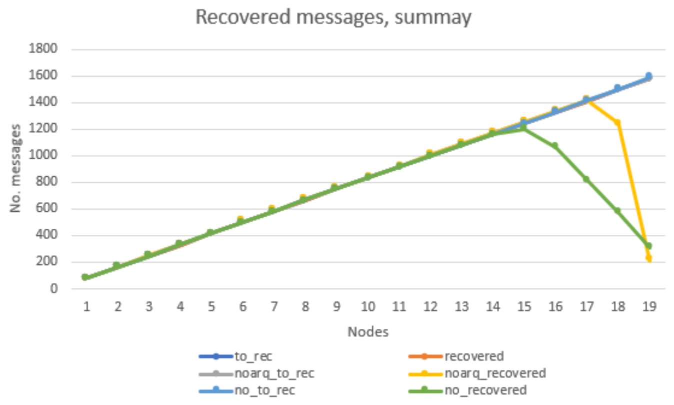

Figure 10. Comparison between recovered messages in the three simulations

parameters set to false. The results show a linear increase in the average time of recovering the old messages (third plot in Fig.8), which is also proportional to the number of messages that a new node has to recover. We can see in `no_to_rec` and `no_recovered` in Fig.10 that the first 14 nodes manage to recover 100% of the old messages, whereas the remaining ones are not able to recover everything in the time given by the simulation. In fact, from the 15th node we see a decrease in the percentage of the messages that were recovered, which also corresponds to a decrease in the time needed. In the comparison in Fig.9 we can see how the average time is more than double with respect to the previous case, with peaks up to 570 seconds.

VII. Limits, conclusion and further work

The analysis that we conducted were just a subset of all the ones that would have been interesting to consider, and the limited time retained us from covering more cases than the ones presented. For example, it would be interesting to analyse in more depth the dynamic network case and also the performance of the protocol when it is needed only for unicast and multicast purposes. Generally, since the two functionalities rely upon a generic broadcast, we believe that there would not be many differences. It would also be interesting to see the behaviour the protocol wold exhibit with some modifications to limit the collisions which we analyzed. This could be in two parts: 1) in which a collision detection system to *Machine* is added so to implement an adaptive behaviour to intelligently limit the number of collisions when in high load simulations and to push for a lower random latency 2) limiting the flooding behaviour that currently the re-broadcast has by rebroadcasting only with a certain probability to reduce the usage of the channel while retaining the convergence proprieties of the protocol (as rarely a node receives a message from a single machine)

References

[1] C. Tschudin, “A broadcast-only communication model based on replicated append-only logs,” *ACM SIGCOMM Computer Communication Review*, vol. 49, no. 2, pp. 37–43, 2019.

[2] M. M. I. Mamun, T. Mahmud, S. Debnath, and M. Z. Islam, “Analyzing the low power wireless links for wireless sensor networks,” *CoRR*, vol. abs/1002.4838, 01 2010.