Lightweight Probabilistic Broadcast

Luca Zanella
Student ID 207520
luca.zanella-3@studenti.unitn.it

Daniele Giuliani
Student ID 203508
daniele.giuliani@studenti.unitn.it

Giuliano Turri
Student ID 207553
giuliano.turri@studenti.unitn.it

***Abstract*—In the last decades, gossip-based algorithms have become increasingly popular. These kinds of algorithms trade reliability for scalability, which is becoming even more important given the use of distributed systems to parallelize very complex tasks. This paper will examine our implementation of Lightweight Probabilistic Broadcast [1] using the agent-based framework Repast Simphony. We first describe the main architecture. Then, we present the model used for visualization. Finally, we provide an analysis of the performance achieved by our implementation.**

***Index Terms*—Gossip-Based, Broadcast, Probabilistic, Repast, Java**

I. Introduction

In order to evaluate the performance of the Lightweight Probabilistic Broadcast algorithm described in [1], we have implemented a simulator using Repast Simphony[1], a popular actor-based modeling framework.

The simulator provides several parameters to customize the execution mode. It allows users to execute the algorithm both in synchronous and asynchronous mode, simulating network delay and message loss. We have also implemented optimizations, such as age-based event purging and frequency-based membership purging, that can be enabled and disabled in order to analyze how they influence the behaviour of the protocol. We also decided to implement the routines necessary to perform the recovery of lost events. This behaviour was briefly described in the original paper, but never implemented, nor considered, in the analysis presented in [1].

The simulator and data collection components are written in Java, using Repast Simphony Java API in order to interact with the framework. The scripts used to perform data analysis and plotting of graphs are written in Python[2]. All the code is available in our Github repository[3].

II. Architecture

The main agent of our architecture is implemented in the `Process` class. At the beginning, the class `LpbCastBuilder` creates N instances of this agent. As there is no centralized memory, the interaction between these processes is realized through the exchange of messages. In order to mimic this behaviour, each process stores a queue of incoming messages. Three types of messages can be shared among processes:

[1]https://repast.github.io/
[2]Additional libraries might need to be installed
[3]https://github.com/LucaZanella/lpbcast

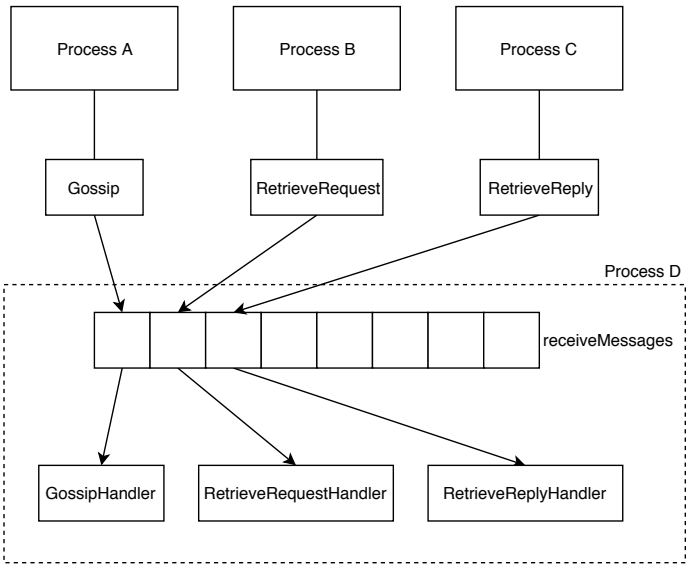

Fig. 1. A representation of the dispatching of messages to the corresponding handlers.

- `Gossip`: a message to periodically spread the information between processes.
- `RetrieveRequest`: a message to require the retransmission of a missing event.
- `RetrieveReply`: a message to send an event to a process that required it.

In order to simulate network delays, each message carries a field, named `tick`, whose value denotes the simulation tick during which the message has to be processed. When the simulator is run in synchronous mode, each message sent at tick X will be processed at tick $X+1$ (without the possibility of being lost).

At each time step, each process iterates through the queue containing the incoming messages and dispatches them to the handler responsible for their processing as shown in Fig. 1. Since Repast is not a message-based framework, this queue may be simultaneously accessed by several processes. In order to avoid concurrency issues, an unbounded thread-safe queue was used in the implementation.

Differently from [1], where the retrieval of missing events is only described but never implemented, our implementation leverages two buffers to accomplish this task:

- `retrieve`: the buffer, originally described in [1], used

to collect missing events that might need to be recovered.
- `activeRetrieveRequest`: the buffer used to maintain the state of the requests which have already been issued in order to recover a missing event.

When a process detects a missing event, it first adds it to the `retrieve` buffer. If no gossip message carrying this event is received within a certain number of ticks, the process starts the recovery phase. The missing event is then removed from the `retrieve` buffer and added to the `activeRetrieveRequest` buffer. From that moment, the active request traverses four states:

- The missing event is requested to the sender of the gossip message which allowed the process to detect the loss of the event.
- If no reply is received, the missing event is requested to a process selected randomly from the view.
- If no reply is received, the missing event is requested to the process which originally broadcast the event.
- If no reply is received, the event is considered lost.

The receiving of a recovery request and a recovery response are handled by the `retrieveRequestHandler` and `retrieveReplyHandler` methods respectively. In particular, the process responsible for replying to a `RetrieveRequest` message checks for the presence of the missing event inside its `events` and `archivedEvents` buffers.

In order to analyze the performance of the protocol, our implementation leverages a particular agent, named `Collector`, which is used to gather information from processes. For example, it collects information regarding the number of times each event is delivered or the tick during which each process becomes aware of a new subscriber. This agent is used in combination with some custom data sources that we have implemented based on Repast Simphony guidelines.

The visualization of the model exploits an agent, named `Visualization`, to collect the actions performed during every tick by each process and visualize them on a display.

III. Real time visualization

The Repast framework provides APIs which allow users to display in real-time how the simulation evolves and how agents interact with each other. In our simulation, a network of nodes is displayed in order to visualize how processes relate in terms of message passing. To better analyze the behaviour of the protocol, the visualization shows the propagation of a single event at the time through the network, this way the interface remains clear and easy to interpret. A timeout is provided in order to establish when the visualization should start displaying a more recent event. The network is displayed by means of a directed graph in which the nodes correspond to the processes and the edges represent how processes are linked together and how they exchange messages. There are different types of edges according to what they represent:

- *Light gray edges*: represent the view of a process. This kind of edges allows users to see how processes are connected and how their view changes over time, they are particularly useful to detect the presence of nodes isolated from the rest of the network.
- *Colored edges*: represent the gossip of that particular event. The appearance of a new color represents the fact that another event is being considered. Analyzing this type of edges, at each tick, allows users to see how the event propagates through the network.
- *Thin red edges*: they represent the `RetrieveRequest` messages.
- *Thin blue edges*: they represent the `RetrieveReply` messages.

In addition to the visualization of the propagation of the various messages, the real-time visualization model allows users to see some action performed by a process:

- *Delivery*: when a process delivers the displayed event, it becomes of the same color used to represent the gossip message. After a specific amount of time has elapsed, the color of each node is reset to the default one and the visualization proceeds to show a new event.
- *Submission*: the processes joining the network are briefly flashed in a bright red color.
- *Unsubmission*: when a process performs a submission, its color changes to light gray and after a short amount of time is removed from the view.

IV. Parameters of the simulator

During this section, we will explain all the parameters that can be changed in order to personalize the simulation. Given the modular structure of our software, many different analyses can be performed by tweaking the parameters appropriately. Unfortunately, Repast Simphony does not provide any customized way to order parameters but automatically sorts them alphabetically, thus we will describe them in this order. With the description of each parameter, we will also provide a default value, which we suggest using in order to model a generic execution of the protocol under normal conditions, which is easier to understand at a glance.

- *Age-Based Message Purging Optimization [enabled]*: allows users to enable and disable the first optimization described in the paper which influences how the `events` buffer is trimmed.
- *Average Frequency Multiplier [0.9]*: this is a float value between 0 and 1, which is multiplied by the average frequency of a subscription as described in [1]. It influences the behaviour only if the frequency-based membership extension is enabled.
- *Buffer ArchivedEvents Max Size [25]*: the maximum length of the buffer `archivedEvents`, which stores messages in order to be able to satisfy retransmission requests.
- *Buffer EventIds Max Size [50]*: the maximum length of the buffer `eventIds`.
- *Buffer Events Max Size [5]*: the maximum length of the buffer `events`.

- *Buffer Subs Max Size [5]*: the maximum length of the buffer `subs`.
- *Buffer Unsubs Max Size [5]*: the maximum length of the buffer `unsubs`.
- *Default Random Seed*: default parameter of Repast. It describes the random behaviours, by using the same seed we can reproduce a single, specific run of the model.
- *Event Generation Probability [0.01]*: the probability (between 0 and 1) of each process to generate an event during every tick of execution.
- *Event Visualization Switch Timeout [100]*: the number of execution ticks after which the visualization is reset and the propagation of another event is shown.
- *Event Generated per Round [0]*: the number of events generated randomly each tick of the simulation. This parameter can be useful in analysis where the network needs to be highly stressed.
- *Fanout [3]*: the size of the fanout of each process. Must be lower than *View Max Size*.
- *Frequency Based Membership Purging Optimization [enabled]*: it allows users to enable and disable the second optimization described in [1], which influences how the `view` and `subs` buffers are trimmed.
- *Initial Events In System [0]*: it initializes the system with a number of events. Useful when performing specific analysis where random event generation is not viable.
- *Max Transmission Delay [10]*: it specifies the maximum transmission delay that can occur between the sending and arrival of a single message. It influences the behaviour of the model only when *Synchronous Mode* is disabled.
- *Node Number [15]*: the number of nodes initially in the system. A low value is advised in order to understand how the protocol works. Higher values can be used when performing analysis.
- *Recovery Eligibility Timeout [20]*: the number of ticks that must pass before a missing `eventId` becomes eligible for the recovery process.
- *Recovery Source Timeout [20]*: the number of ticks after which, a recovery request made to a node, is considered failed. When this happens the request can be re-routed to another node or canceled.
- *Same Origin Event Replacement Timeout [100]*: this parameter influences the behaviour of the age-based event optimization. It is referred as *LONG_AGO* in the original paper where it is described [1].
- *Subscription Probability [0.0001]*: the probability (between 0 and 1) that a new node will join the network in the current tick of execution.
- *Subscription Visualization Timeout [2]*: the number of ticks after which a subscribed node is not considered "new" anymore, its color will get reset back to default.
- *Synchronous Mode [enabled]*: establishes if the model should run in Synchronous or Asynchronous mode. When the synchronous mode is enabled, each message set is received in the following tick, random transmission delay and message loss do not occur.
- *Unsubs Expiration Time [100]*: the number of ticks after which an unsubscription is considered no valid and is removed from the buffer `unsubs`.
- *Unsubscription Probability [0.0001]*: the probability (between 0 and 1) that an existing node will leave the network in the current tick of execution.
- *Unsubscription Visualization Timeout [2]*: the number of ticks after which an unsubscribed node is removed from the view.
- *View Max Size*: the maximum size of the process's view.

During section V, each analysis that uses specific parameters will describe them and explain why this is the case. If the parameters are not shown, the default values can be assumed.

V. Analysis

In order to make our analysis more effective, we employ the ability provided by Repast Simphony to define custom data sources. All the implemented custom data sources can be found inside the `analysis` package of our application.

Each analysis describes different aspects of the protocol. Some helped us detect bugs in our implementation, while others allowed us to discover possible ways of improving the protocol itself.

A. View distribution

In order to estimate the expected number of processes that know a given process, we run the model for 100 iterations in a network with 100 nodes and we compute, for each tick, the number of times each process is known by other processes. We then average the number of occurrences of each process over these 100 iterations. As depicted in fig. 2, the expected number of times each process is known by other processes resembles a Gaussian distribution centered in 4.6. The authors of [1] claim that this expected number should be equal to the view's size l. As the size of the view used in this simulation is 5, the results of the analysis seem to confirm this estimate. As a result, we can assume that the views are uniformly distributed as stated in [1].

B. Subscription propagation

In order to evaluate the performance of the frequency-based message purging optimization, we analyze the number of simulation ticks required for a subscriber to be known by the entire network. At the beginning, the network is highly unstable since all the elements contained in the subscription buffers have a low frequency. This is due to the fact that initially all the `subs` buffers are empty and get filled as the network evolves. As a result, a subscription sent during the initial rounds is as likely to be removed from the subscription buffer as all the others, invalidating the improvements provided by the optimization. For this reason, we perform a subscription only after 100 simulation ticks. As the new subscriber is expected to be known by all the other processes in a few ticks, during the simulation we keep the event generation, subscription and unsubscription probability fixed to 0. As depicted in fig. 3,

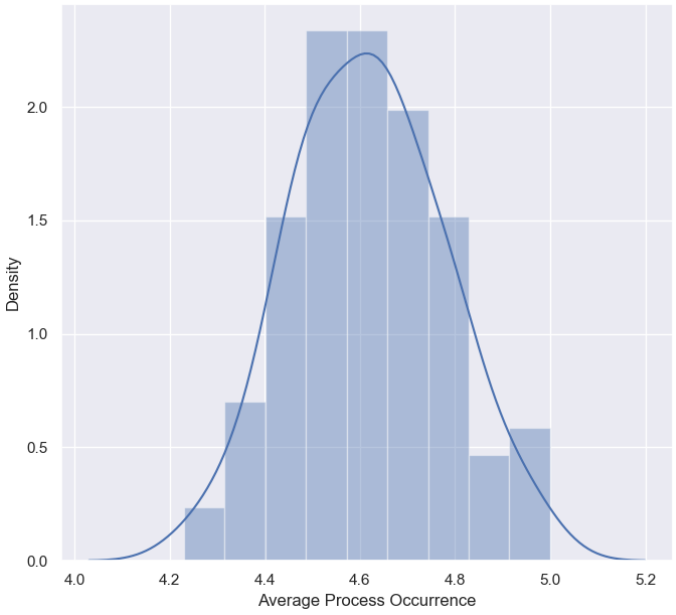

Fig. 2. A representation of the density estimation of the expected number of times each process is known by other processes during a run of 100 iterations in a network with 100 nodes.

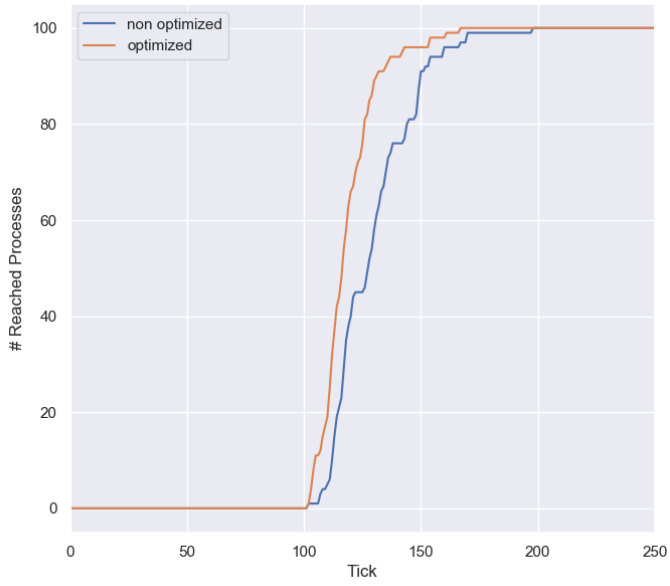

Fig. 3. A representation of the number of ticks required for a new process to be known by all the other process during a run of 1000 iterations in a network with 100 nodes.

the optimized version of the protocol requires fewer ticks to reach the entire network than the non-optimized version. It should be noticed that the optimized version requires half of the ticks required by the non-optimized one to reach 90% of the network.

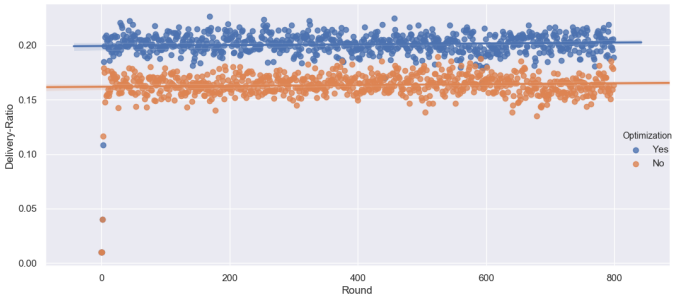

Fig. 4. Graph representing the improvement of delivery ratio due to the age based message purging optimization

C. Age based message purging optimization

The original lpbcast approach [1] proposes an optimized purging policy of the events buffer: when the buffer is saturated, instead of removing a random element, the events with higher age are removed. Giving priority to the newest events, the age-based purging optimization should lead to an increase in the delivery-ratio of the protocol.

In order to prove the effectiveness of this optimization, we have performed some simulations analyzing the deliveries performed at each round by the processes. To allow the execution of the purging policy, the size of the events buffer has to be small and the number of events generated per round has to be high. We have so used an events buffer with a capacity of 5 elements generating constantly at each round 30 new events by different random processes. Because of the large amount of data to be processed due to the high frequency level of events generated, for this kind of simulations, the network is initialized with 60 nodes. Notice that the values of the parameters used for this simulation are very similar to the ones used in the original paper so that a comparison can be done. As depicted in fig. 4, the optimization improves the delivery ratio of the protocol. The blue points represent the delivery ratio level obtained during the simulation rounds of the optimized version. On the other hand, the orange points represent the delivery ratio level in the non-optimized version. From the resulting graph, it can be inferred that the optimized message purging policy improves in a significant way the event's delivery ratio. A comparison between the results obtained in our simulations and the ones presented in [1], shows how the average delivery ratio obtained in our simulation is lower. This is probably due to the small size of our events buffer causing each process to deliver fewer events at each round.

Given that the older an event is, the more chances it has to have been propagated through the entire network, adopting the classic purging strategy removing a random element, it is possible that a very old event is included within the next gossip message increasing in this way the probability that the recipient has already deliver it to the application. Adopting the optimization, purging the oldest events instead, only the newer are included inside the next gossip, promoting in this way the

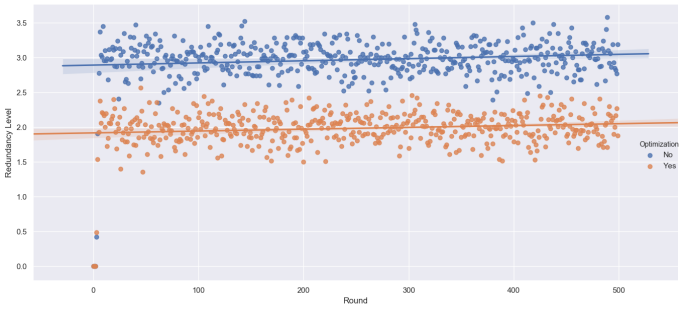

Fig. 5. The graph shows how the age based message purging optimization is effective in terms of redundancy level. The plotted points represent the average amount of redundant events received by a process at every round

propagation of events which has not already been delivered by the other processes reducing so the redundancy level of the network.

The chart in fig. 5 demonstrates the effectiveness of the age-based purging optimization also in terms of redundancy level. Using the same parameters of the previous experiment, two different simulation runs have been performed. The blue points represent the redundancy level using the optimized events purging strategy, while the orange ones represent the redundancy level when the random purging strategy is adopted.

In order to obtain the results shown in fig. 5 and highlight the effectiveness of this optimization, the network is stressed by generating a very high number of events. Without such conditions, our simulations revealed that the age-based purging strategy does not particularly improve the performance of the protocol.

D. Message Propagation

To estimate the performance of our protocol, we measure the number of rounds necessary for a message to reach every actor in a network composed of 1000 nodes. This measurement was repeated for different sizes of the fanout to establish the impact of this parameter.

During this analysis, the generation of random events was disabled, as well as random subscriptions and unsubscriptions. Furthermore, the measurements were averaged over a sequence of 20 runs in order to account for randomness. The buffers of `archivedEvents` and `eventsIds` were kept long enough to avoid the failure of the recovery process and ensure that it was performed for each missed event id. The result obtained are shown in fig. 6.

As we can see, initially the message is propagated through the network very quickly during the initial gossip phase. The first decrease in the slope corresponds to the end of the gossip when every node that has received the message has already gossiped about it. Inevitably, the nodes that were not selected by any process will be left out, but they will be notified about the existence of that event thanks to the continuous gossiping of event ids. The plateau corresponds to the recovery timeout, which was set to 20 tick as the graph correctly represents. As the timeout expires, the nodes that were left out begin the

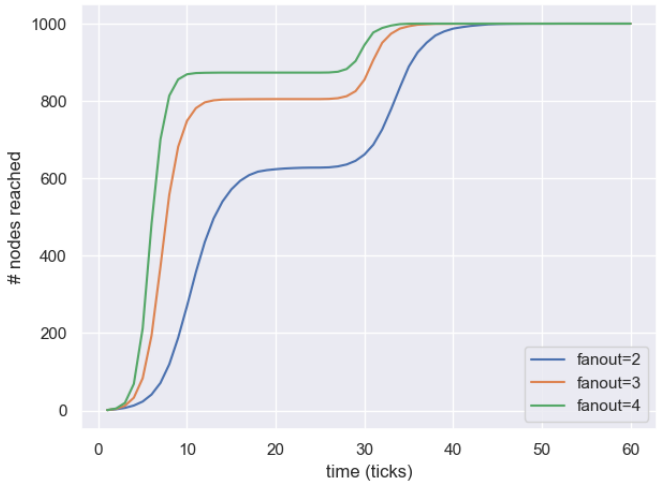

Fig. 6. Message propagation with different fanout sizes.

recovery process and eventually, the message is received by all nodes.

Increasing the size of the fanout, the number of nodes that are reached by the initial gossip increases (thus do not have to perform the recovery process), but the improvements are less significant at every step-up.

E. Recovery Requests vs Fanout

In the graph shown in fig. 7 we can see the average number of recovery requests for the propagation of one event in a network of 1000 nodes.

As expected, increasing the fanout decreases the number of recovery requests since the event reaches more nodes during the gossip phase. Regardless, a recovery protocol is needed: even when setting the fanout size to 5 (equal to the view size), some processes are not reached by the gossip. This behaviour is to be expected, given that we are considering a probabilistic broadcast algorithm.

As in the previous analysis, the measurements were averaged over 20 runs, the random generation of events, subscriptions and unsubscriptions were disabled and the buffers were long enough in order to correctly perform the recovery of each lost event.

F. Recovery Requests vs View Size

In fig. 8, we can see how the same measurement of the previous analysis changes as we increase the view size of the processes. The average number of retransmission requests barely decreases, which confirms the claim of the authors of [1], who state that the number of nodes infected by an event is slightly correlated with the size of the view.

The same parameters used for the previous analysis were applied to this one. The only difference is that the fanout was kept constant (to its default value 3), while the view size was changing.

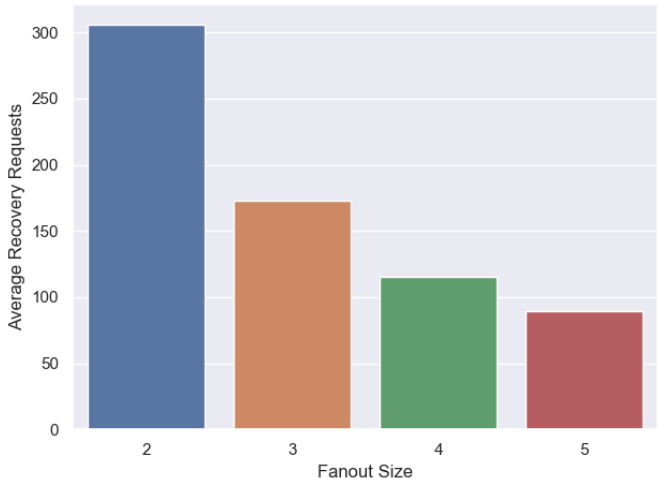

Fig. 7. The average number of recovery requests for the propagation of one event, using different fanout sizes

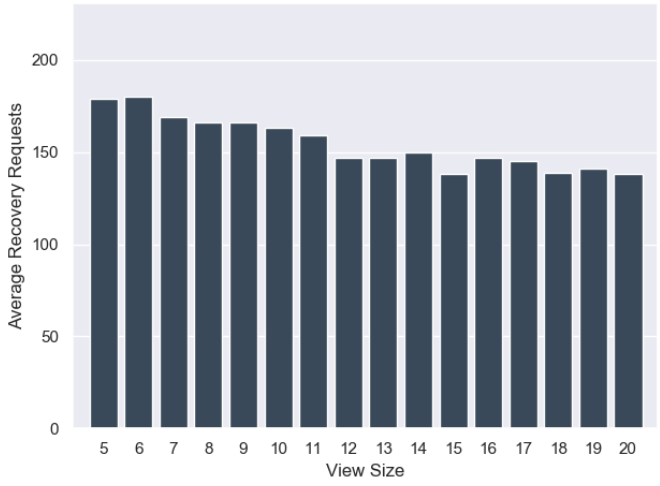

Fig. 8. The average number of recovery requests for the propagation of one event, using different view sizes

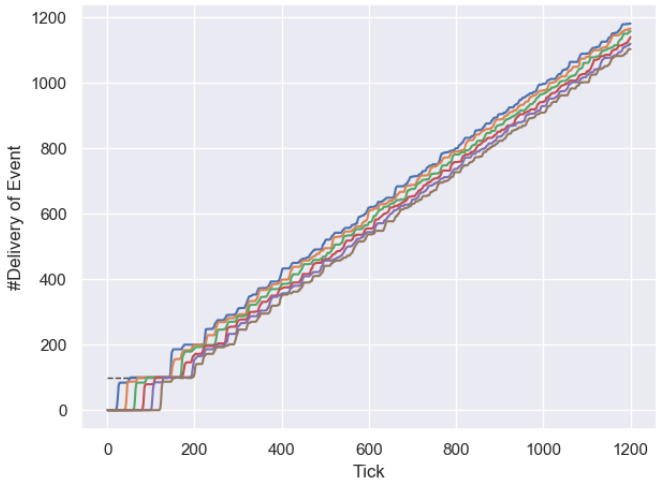

Fig. 9. The number of deliveries with relation to time elapsed. Each color represents different events. The dashed line represents the number of processes in the system.

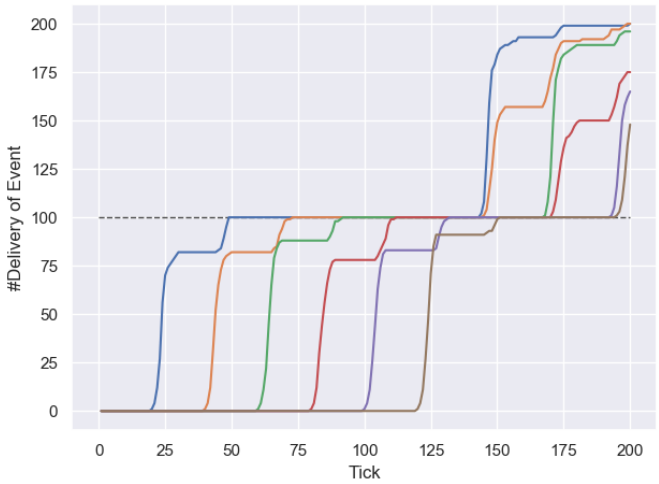

Fig. 10. The number of deliveries with relation to time elapsed. Each color represents different events. The dashed line represents the number of process in the system.

G. Event Looping

During the analysis of the protocol, we noticed some strange behaviour: after executing the simulation for a while the number of deliveries for a single event was linearly increasing, surpassing even the number of processes in the network, as shown in fig. 9. This means that each process was delivering the same events over and over again.

With some debugging, we discovered that this is due to a combination of the way the recovery of missing events is handled and how the buffer `eventIds` is trimmed. In fact, the simplistic approach described by [1] is not enough to prevent events from looping indefinitely inside the system.

To better understand this behaviour we ran a simplistic simulation with 100 nodes. All the buffers size were set to 5, except for the buffer of archived events, which was large enough in order to be able to satisfy all retransmission requests. We instructed the model to generate an event every 40 ticks of execution and stop generating events after the 120th tick (therefore 6 events generated in total) in order to cause an overflow in the buffer `eventIds`. The results of this simulation are shown in fig. 10.

As we can see, the protocol initially behaves correctly: events are generated and are gossiped around the network, the nodes that do not receive the events ask for their recovery after the 20 tick timeout has expired. This happens until the `Event6` is generated (in fig. 10, the brown line), at this point the buffer `eventIds` is full and the oldest event id contained is removed, which is the `EventId1` (in fig. 10, the blue line). In the meantime, all the nodes that have not received

`Event6` are still sending out gossip messages containing `EventId1` (because their buffer has not overflown yet). When this happens, all processes that have already received `Event6` will receive `EventId1`. Since this is not in their `eventId` buffer, they will consider it as a lost event. The recovery will be performed, the event will be delivered once more and the `EventId1` will be added again to the buffer, therefore renewing its "age" and creating an infinite cycle.

It is important to note, that this is an issue of the protocol as described by the authors of the original paper itself and is not related to our implementation. To solve it, the introduction of an additional timestamp might be required in order to avoid recovering events that are too old. An easier solution to implement, is to avoid replying to a recovery request when the `age` parameter of the event requested is higher than a specific threshold. The requester will timeout multiple times, but after trying again with different processes, it will eventually surrender and the event will be removed from the system.

VI. Conclusion

Having local knowledge of a subset of the processes inside the network allows each process to disseminate the information in a completely decentralized way. This approach seems to be particularly useful for P2P systems, allowing the algorithm to achieve robustness and scalability, still maintaining a good degree of reliability. This project was especially interesting because it allowed us to obtain some hands-on experience with this kind of algorithms. The analysis process was useful to debug and test the correctness of our implementation and also revealed various insights on how the parameters can drastically change the behaviour of the protocol. In the future, we aim at implementing additional optimization to our simulator in order to see how much we can increase the performance and efficiency of our implementation.

VII. How to Install

In order to run the simulator:

- Run the installer `lpbcast_model.jar` by either double-clicking on it or by opening a terminal and typing *java -jar lpbcast_model.jar*.
- The Repast Simphony Model Installer will be displayed, follow the guided installation procedure.
- Execute the generated shortcut or run the file `start_model.bat` contained in the installation folder.
- Customize the parameters of the model in the parameters panel, initialize the run and start the run.

If you are not able to start the simulation, this may be due to an outdated version of your Java Runtime Enviroment: Java 11 or newer is required to correctly execute the software. Please consider to update your Java version.

References

[1] Eugster, P. T., Guerraoui, R., Handurukande, S. B., Kouznetsov, P., & Kermarrec, A. M. (2003). Lightweight probabilistic broadcast. ACM Transactions on Computer Systems (TOCS), 21(4), 341-374.