UNIVERSITY OF TRENTO
MASTER IN COMPUTER SCIENCE

Assignment 1: Lightweight Probabilistic Broadcast

Students: *Giacomo Callegari (M. 207468), Fabio Molignoni (M. 203201)*

Course: *Distributed Systems 2*

1. Architectural choices

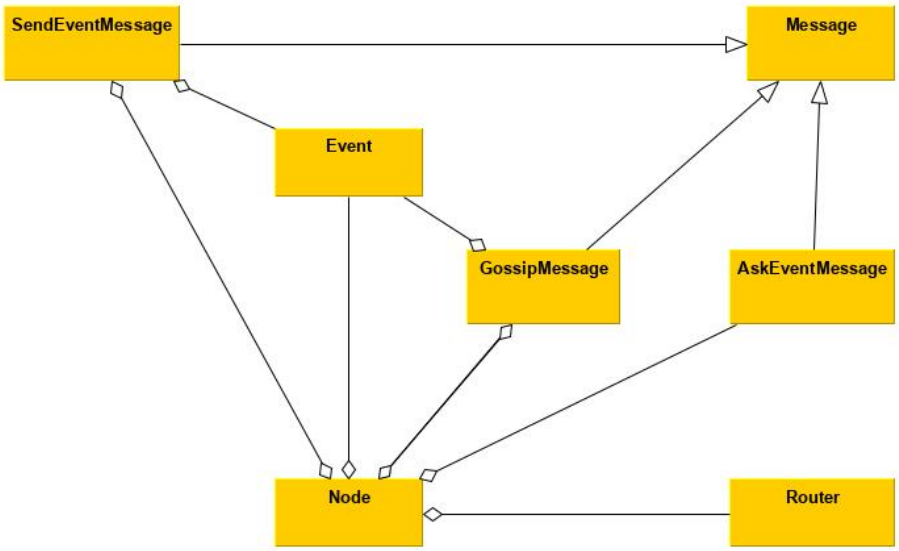


Figure 1: Architecture of the project

Figure 1 shows the main classes used in the implementation of the *Lightweight Probabilistic Broadcast* algorithm. The agent `Node` can be considered as one process which runs the broadcast algorithm and is identified by a unique ID. At each step of the execution, a node can generate messages, send/receive gossips and send/receive specific events to another node. A node can also perform subscription and unsubscription at each step. Both the generation of events and the sub/unsub are handled probabilistically: parameters determine the probability that the node will generate a message and sub/unsub at step t. A node that wants to send some information to other nodes in broadcast generates an `Event`. This class can be considered as a wrapper of the actual content. The algorithm requires the node to communicate with other nodes by sending and receiving messages. In order to implement this feature, it was decided to apply an approach similar to the one used in the framework *Akka*. Nodes know only the ID of other nodes, and they can only communicate with each other using messages. In order to implement message passing, two classes were defined: `Message` and `Router`. Message is a class that defines a generic message that can be sent using the router. The router is the handler of the communication: nodes send messages using the method

`send(message, nodeId)` defined inside this class. The router drops some messages: the actual amount of dropped messages can be set as parameter in the simulator. This feature was inserted in order to allow users to simulate the execution of the algorithm with an unstable connection. There are three types of messages that can be sent by nodes, each type of message is subclass of the `Message` class:

- `GossipMessage` is the gossip message sent at each step by each node.
- `AskEventMessage` is the message used to ask to another node a particular event.
- `SendEventMessage` is the message used to send back the actual event asked by another node using an `AskEventMessage`.

Using this hierarchy of messages, it was possible to define inside a node just a function `receive (Message m)` which is the function called each time a node receives a message. The paper assumes that a node, after a certain amount of time, can directly ask for events. Since the paper does not describe precisely how this should be handled, it was decided to store inside a buffer all the received events, from which the various nodes can get and send events to others. At each step the events that are old enough (i.e. older than 20 rounds) are removed from the buffer, assuming that no one will ever ask for them in the future. In the development of the algorithm it was also decided to implement the "age-based message purging" improvement described in the second part of the paper.

We have also implemented one of the optimizations proposed in the paper, which is age-based message purging. In the original algorithm, if the `events` buffer exceeds its maximum size, random events are removed until the limit is respected: the optimization suggests to remove the oldest events instead, since they will be more "noisy" (i.e. propagated). We have associated an age to each event, which increments at each gossip and is updated when receiving the same event from someone else. First, events that have a sufficient absolute age are removed; then, if the buffer is still full, the oldest remaining events in the buffer are purged.

2. Presentation of the simulator

The algorithm has been evaluated in Repast Simphony, an agent-based modeling toolkit that supports Java among other languages. `Node` is the only type of agent that we defined, since the *lpbcast* algorithm only describes the interaction between nodes.

We have created several displays to visualize the execution of the algorithm. In particular, they are network displays that represent the nodes of the system and their interaction:

- *Sent Messages* shows the message exchange between nodes
- *Subs* shows the subscriptions of each node
- *Unsubs* shows the unsubscriptions of each node
- *View* shows the partial view of each node

These displays are updated at each tick of the simulation and therefore represent the evolution of the system in real time. In the network, the nodes are normally shown in

black, but become blue when gossiping, green when delivering and red when unsubscribing.

Repast Simphony also allows to create time series charts, where one or more measures are plotted against time (i.e. ticks of the simulation). We have created a chart that represents the number of events and of deliveries in time to better visualize the delivery rate of messages.

Another important feature of Repast Simphony is represented by the parameters of the simulation, which affect the behavior of the algorithm. The parameter values can be changed directly from the interface of the simulator, instead of modifying the code each time. We have defined many different parameters that can be customized in the simulation:

- Number of nodes and fanout
- Maximum size of the various buffers
- Probability of generating/receiving a message
- Probability of subscribing/unsubscribing
- Rounds to wait before fetching events from sender/random node/source
- Flags for age-based message purging/retransmissions/unsubscriptions

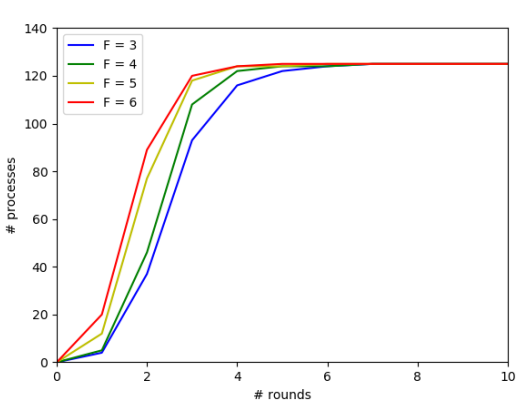
We have selected the `LpbcastBuilder` class as Data Loader for the simulation, in order to define the scenario. We have also specified an aggregate Data Set for events that are exchanged across the system: this allows us to collect data about events at each tick and visualize it in the aforementioned time series chart.

3. Analysis

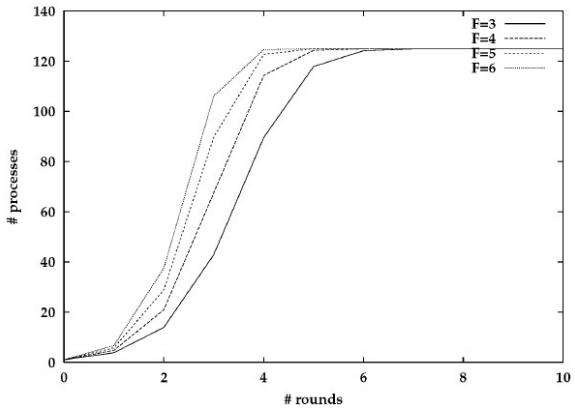
After the development of the algorithm, we performed quantitative analysis in order to verify whether or not the results of our implementation match with the one presented in the original paper. Some of the presented graphs were generated inside Repast, while for others some CSV files were generated and then the final result was plotted using the "Matplotlib" library of Python.

Figure 2 shows the expected number of infected processes using different fanout values. The fanout defines the number of gossip messages sent at each round by each node. In order to produce the graph of figure 2a, we run our algorithm using $n = 125$ processes for 20 times for each possible fanout size. We then compute the average number of infected processes for each round and plot the final result. From the comparison between our results and the one of the paper, it appears that our system is able to replicate well the intentional behaviour of the proposed algorithm. Some differences can be noted, such as the fact that our implementation spreads messages faster in the first phase and slower in the second phase compared to the one of the paper. This may be due to the decision of the initialization of the network. Since no assumptions were made inside the paper, we decided to randomly generate the initial configuration of the system (i.e. the initial view).

Figure 3 shows how the the number of processes inside the system influence the number of rounds needed in order to propagate a message to most the subscribers (99% of them). It appears that our implementation of the paper (Figure 3a) mimics

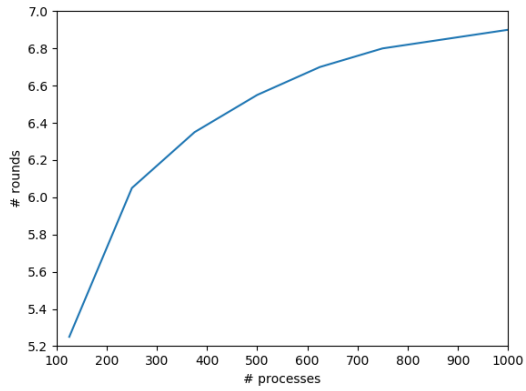


(a) Graph using our implementation

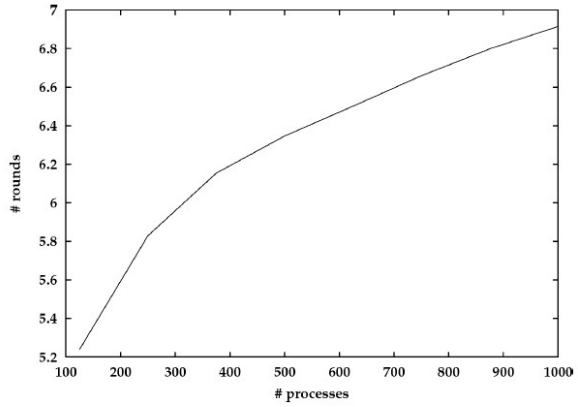


(b) Graph of the paper

Figure 2: Expected number of infected processes for a given round with different fanout values.



(a) Graph using our implementation

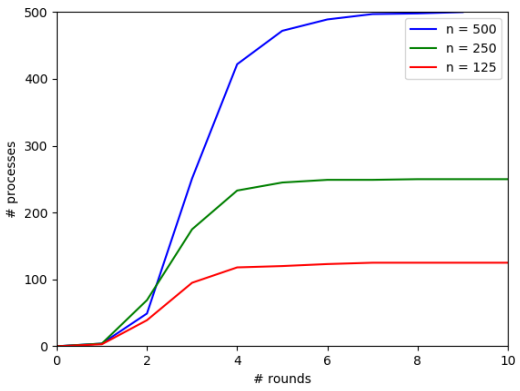


(b) Graph of the paper

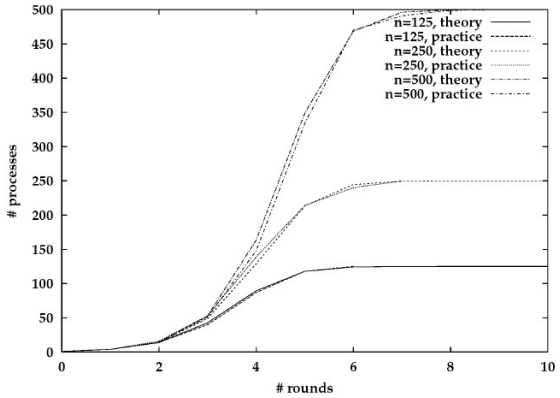
Figure 3: Expected number of rounds necessary to infect 99% of Π, given a system size n.

closely the graph presented in the paper (Figure 3b). In order to produce the graph, we computed the average time of propagation for a single message for a network with n processes, with n that goes from 125 to 1000 adding each time 125 new processes. Our implementation presents some differences, such as the fact for 250 process we have an expected number of rounds of ≈ 6, while in the paper, for the same amount of processes, the expected number of rounds is ≈ 5.8.

Figure 4 shows a graph strictly related to the previous one. In particular, it compares the results obtained in the analysis with the actual simulation of the algorithm. In order to produce this graph, as we did before, we run several times the algorithm with different values for n, computing the number of rounds needed to propagate a single message to the nodes. Our implementation is able to replicate the results obtained by the analysis of the authors. The main differences between the two graphs is at round 2 of the run with $n = 500$. As it is possible to see, in that round our implementation performs worse than the one with $n = 250$. The system is anyway able to recover and obtain, in the end, the expected final result.

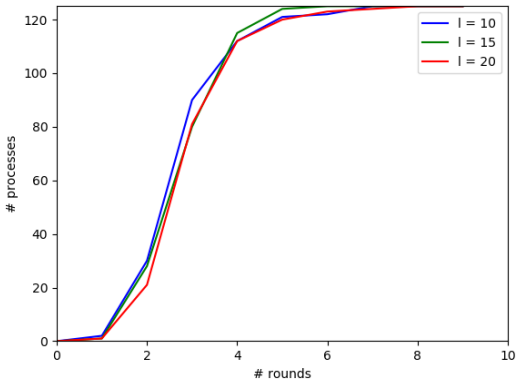


(a) Graph using our implementation

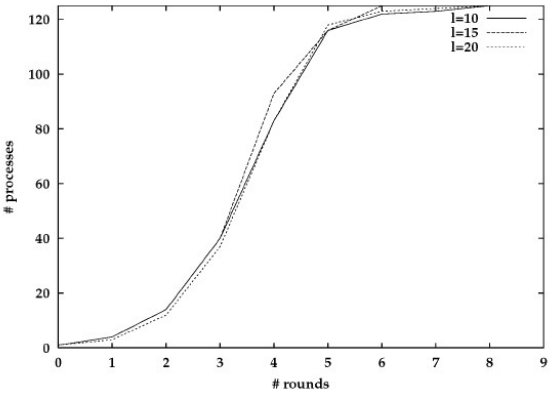


(b) Graph of the paper

Figure 4: Analysis vs simulation



(a) Graph using our implementation

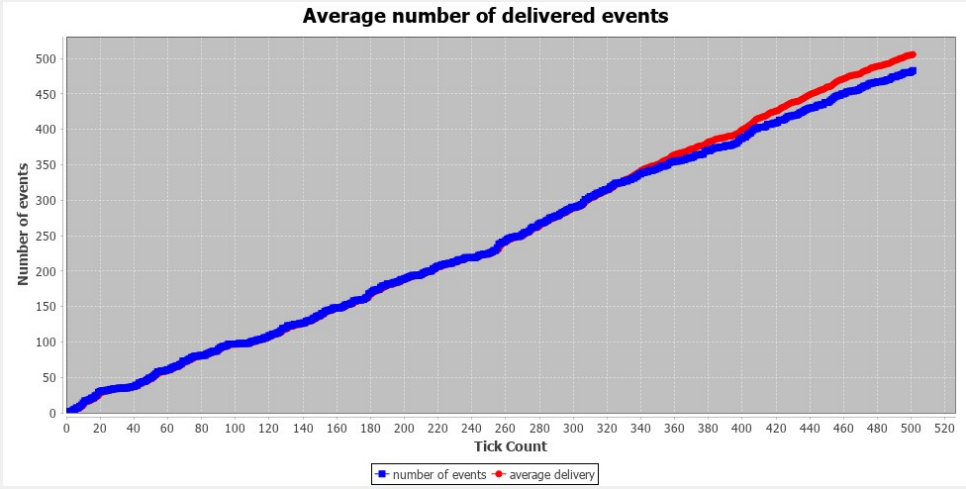


(b) Graph of the paper

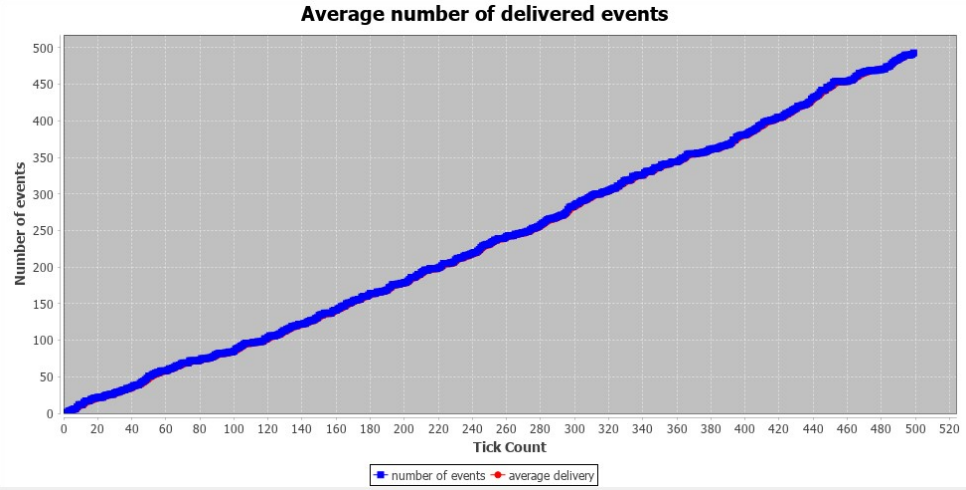
Figure 5: Expected number of rounds necessary to infect 99% of Π, given a system with view l

Figure 5 is used by the authors of the paper to analyze how the size of the view impacts the number of gossip rounds it takes to infect every process in the system. As in their analysis, our implementation shows a small dependency between l and the number of gossip rounds required for the dissemination of an event.

The paper does not describe fully how the parameters should be initialized. Clearly, some of these parameters are really important and should be treated carefully. For example, during our analysis, we saw that by setting bad initialization parameters there is the risk of having the same message delivered more than once by the same process. In order to show that, we used the Repast visualization tool to plot the time charts shown in figure 6. In each graph, the blue line at round t represents the total number of messages generated up until t. The red line represents the average number of events delivered by each process up until t. Assuming that no subscriptions/unsubscriptions are performed during the evaluation time span, we expect that the two lines will overlap most of the time, since every process should deliver exactly the same amount of messages that has been created up until that point. We decided to test this hypothesis with different maximum sizes for the `EventIds` buffer, which contains a subset of ids



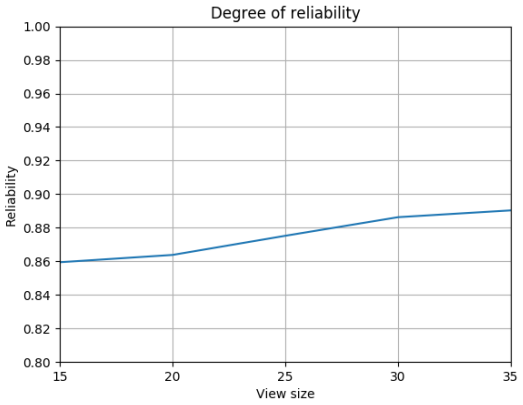
(a) Standard implementation



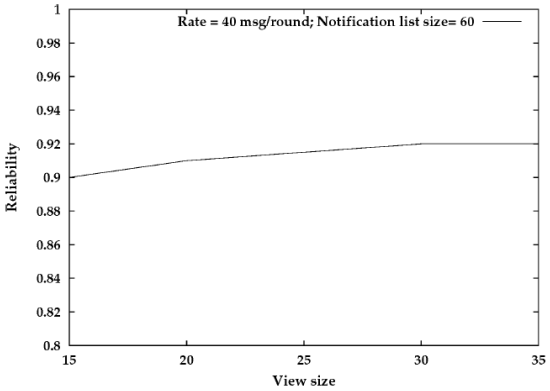
(b) Result using aging on the "EventsForRetransmission" buffer

Figure 6: Comparison between generated and delivered events.

of events delivered by the node, assuming that at each round ~ 1 event is injected inside the system. In figure 6a we tested the system with `MAX_SIZE_EVENTIDS=35`. As it is possible to see, using this value we do not obtain the expected result. In particular, after a certain round (≈ 340) the average number of delivered events grows faster than the number of generated events. Since that, on average, a process delivers more events than the number of generated ones, this mean that some old messages are delivered more than once. The main problem is that some messages remain in the system even if they are old. Since a node cannot know a-priori that it has already delivered that message (since it is no more in the eventIds buffer), it will request,receive and deliver again the same message many times. We decided to use the `eventsForRetransmission` buffer to avoid this incorrect behaviour. This buffer contains the recent events that the node has delivered. In particular, the buffer contains only the events that were generated in the last t rounds (the actual value of t can be decided in the simulator). This buffer allows us to implement two important mechanics. First of all, this buffer is used by the nodes to perform retransmission: if a node receives a request for a particular event, it can send it back using this buffer. The second important purpose is to avoid multiple deliveries: before performing the delivery, the node checks that the event is



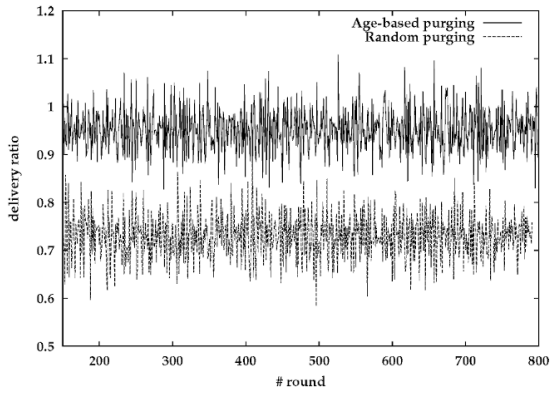
(a) Graph using our implementation



(b) Graph of the paper

Figure 7: Analysis vs simulation

(a) Graph using our implementation (WIP)



(b) Graph of the paper

Figure 8: Analysis vs simulation

not already present inside the `eventsForRetransmission` buffer. Using this approach, we are able to purge from the system messages that are old, and that therefore we assume have arrived to all the nodes. Using this implementation, while keeping the maximum size of eventIds to 35, we obtain the expected result (shown in figure 6b), therefore confirming our hypothesis.

Figure 7 shows the degree of reliability of the system for different view sizes. Here, reliability is intended as the probability of a process delivering any event notification: it can be computed as the ratio of the number of deliveries over the number of generated events. The simulation of the authors shows that reliability slightly decreases for smaller view sizes, and a similar result has been obtained in our simulation. In both cases, there is a small correlation between the view size and the degree of reliability, while the analysis suggested that these measures are independent. Our reliability values are slightly smaller than the ones obtained by the authors: we have performed only 25 rounds of execution due to computational constraints, but a longer run would probably reach a higher reliability, matching the results of the paper.

Figure 8 shows the delivery ratios of the algorithm with and without the optimization of age-based message purging. The results provided by the authors show a re-

markable difference, as the optimized *lpbcast* achieves a delivery ratio of about 0.95 and the original algorithm has an average of 0.7. Our simulation also shows an advantage in the use of age-based purging, but the difference is very small. We can observe, however, that the optimization produces a more stable delivery ratio from about 250 rounds onwards.

4. Installation

We used Repast Simphony to create a self-contained JAR installer of the simulator, which can be found at the following link `https://drive.google.com/open?id=1RGtfkaJWzE7oYaGtMU8KV9Mnpc1ZvJZ2`. The installation is very simple: after opening the installer, a step-by-step guide will be shown to the user. Then, the simulator can be started by running the batch file in the installation folder, which also contains the Java source code.