Lightweight Probabilistic Broadcast

Alessandro Cacco
mat. 203345
alessandro.cacco@studenti.unitn.it

Andrea Ferigo
mat. 207486
andrea.ferigo@studenti.unitn.it

Enrico Zardini
mat. 207465
enrico.zardini@studenti.unitn.it

1 Introduction

This work aims at illustrating an implementation of the lightweight probabilistic broadcast protocol (*lpbcast*) described in [1]. In particular, both the optimizations presented by the authors, i.e. age-based message purging and frequency-based membership purging, have been implemented, as well as the unsubscription and the events retrieving procedures. This algorithm is based on an epidemiological approach, which ensures scalability and a good degree of reliability. Indeed, although a node transmits an event only to a random subset F of his limited view of the system, the epidemiological approach[1] guarantees that most of the nodes will receive the information after a few number of rounds.
Each node, according to the specifications, performs 4 operations: unsubscription and management of received *unsub* messages; broadcast of an event (i.e. insertion into the queue of the next gossip message); retrieving of missing events; gossip transmission and processing of received gossips. Since the order of these operations (which will be described in detail in Section 2) is not specified in the paper, we have decided to schedule them sequentially at each quarter of a simulation tick. Therefore, each tick corresponds to a round of the gossip protocol (actually, the unsubscription is not always executed).

[1] Basically, every node that receives a message (which is classified as infected) spreads the information in order to *infect* other *susceptible* nodes. This behaviour is based, as its name suggests, on the theory of epidemics [2].

2 Architecture

This section presents the methods and the behaviour of the five classes that have been used to implement the algorithm. In particular, these classes can be subdivided into three groups:

- the control class (`TopologyBuilder`) that instantiates the nodes, initialises the topology and manages the creation of events and the unsubscriptions (Section 2.1);
- the class (`Node`) that defines the behaviour of the agents in the simulation, i.e the nodes (Section 2.2);
- the set of classes (`Event`, `Gossip`, `Element`) that are used to exchange information between nodes (Section 2.3).

2.1 TopologyBuilder

The `TopologyBuilder` class has three main duties: it initialises the simulation; it operates as an application layer for the nodes by making them generate broadcast events; it starts the unsubscription procedure.

2.1.1 Initialisation

The initialisation of the simulation is done by the `build()` method in five steps. Firstly, the `ContinuousSpace` and the `Network` objects needed for the graphic simulation are built. Secondly, the parameters of the simulation are loaded. Thirdly, `nodes_size` nodes are instantiated and added to the context. Fourthly, the view of each node is randomly initialised with the number of nodes specified by the parameter `init_view_size`. Lastly, the events generation and the unsubscription procedures are scheduled for every half tick and a quarter of tick every `unsubcribe_rate` ticks, respectively[2].

2.1.2 Events generation

The events generation is managed by the method `eventGenerator()`, which is called by the scheduler every half tick. Basically, this method selects randomly `broadcast_rate` alive nodes [3] and tells them to generate a new event (`broadcast()` method of `Node` Section 2.2.2), which will be gossiped at the end of the current round.

[2] As described in Section 1, the offsets of 0.5 and 0.25 are used in order to keep the correspondence between simulation ticks and gossip rounds.

[3] A node is considered alive if it is not crashed and is currently subscribed to the group.

2.1.3 Unsubscription

The method `startUnsubProcedure()` is responsible for nodes unsubscriptions. Basically, every `unsubcribe_rate` ticks, at one quarter of the corresponding round, the scheduler executes the method in question, which selects randomly a node and calls its `unsubscribe()` method (Section 2.2.1).

2.2 Node

The `Node` class, which defines the behaviour of the agents in the simulation (i.e. the nodes), consists of four main methods: `unsubscribe()`, `broadcast()`, `processRetrieveBuf()` and `gossip()`. These methods implement the four operations that characterise *lpbcast*. In particular, the first two are managed externally [4], whereas the others are scheduled by the agents themselves at three quarters and at the end of every simulation tick, respectively.

As required by the algorithm, each node keeps six buffers: they represent the local knowledge of the agent and they are used to store the information that is acquired and spread via gossip messages. In particular, the `view` buffer contains the nodes that the agent knows and communicates to. Instead, the `subs` buffer represents a subset of the nodes that are currently subscribed to the group. Since both the optimizations described in the paper [1] have been implemented, these two buffers correspond to key-value maps: the key is a reference to the node, whereas the value is a heuristic metric representing the number of times the agent has heard about that specific node. In this way, when one of these buffers exceeds its maximum size it is randomly purged based on the frequency values.

As regards `events` and `eventIds` buffers, they

[4] They are called by `TopologyBuilder` as described in Section 2.1.

are used to manage the events' propagation. In particular, the first buffer keeps the events that will be gossiped in the next round[5], whereas the second one keeps the ids of the received events (it is a recent history). The purging procedure, which is executed when the buffers exceed their maximum size, is based on the age of the elements in this case. In particular, the events are characterised by an *age* field, which is increased by one at the beginning of every gossip round. Therefore, when the `events` buffer needs to be purged, the age is taken into consideration [6]. Instead, for what concerns the `eventIds` buffer (which is basically a list of ids), the first elements are purged since they turn out to be the oldest ones.

The last buffers are `unsubs` and `retrieveBuf`. The first one keeps the nodes unsubscriptions, which are deleted after a certain time. Basically, each unsubscription has a timestamp that is used to avoid that the notification stays in the system forever (the purging is random for this buffer). Instead, the second one keeps the ids of the events that have not been received yet and need to be retrieved (the node is aware of them thanks to the `eventIds` field contained in the gossip messages).

It is worth to highlight that also failures have been implemented. In particular, a node may crash before performing a gossip with probability `crash_pr`. In this case it needs a number of rounds equal to `recovery_interval` to recover. Moreover, a message may be lost with `message_loss` probability.

Finally, in order to avoid concurrent accesses to the resources of a node, the keyword `synchronized` has been used for the methods that are called by other agents.

[5]These events are the ones that have been generated during the current round plus the ones that have been received for the first time during the previous round.

[6]Firstly, the elements coming from a source that has broadcast more recent events are removed (`long_ago` is used as time threshold). Then the oldest elements overall are deleted.

2.2.1 unsubscribe()

The `unsubscribe()` method[7] manages the unsubscription of the node from the network and schedules its subsequent re-subscription.

First of all, the `subscribed` boolean flag is set to `false` and a subset of `F` nodes taken from the view is notified of the unsubscription[8]. By doing this, the node stops to handle incoming messages and its unsubscription will be gossiped in the following rounds. After that, all the buffers are cleared except for the `view` one, which keeps the `init_view_size` most frequent elements (in this way the node will be able to reenter the network). The last phase is the scheduling of the resubscription after a number of ticks defined by the parameter `unsubscribe_interval`. Basically, at that point the flag `subscribed` will be set to `true` by the `subscribe()` method.

2.2.2 broadcast()

The `broadcast()` procedure, which is called by `TopologyBuilder` as described in Section 2.1.2, performs the creation of a new event. Basically, this method generates the id of the new event by concatenating the node id, the round number and the number of the event in the round. After that, it calls the constructor of the `Event` class and adds the resulting instance to the `events` buffer. Finally, in case the size of the buffer exceeds its capacity (`events_size` parameter), the buffer is purged as described previously.

[7]This method is called by `TopologyBuilder` as described in Section 2.1.3.

[8]This is done by the method `sendUnsub()`, which basically calls the method `processUnsub()` of the target nodes.

2.2.3 processRetrieveBuf()

The `processRetrieveBuf()` method, which is scheduled at three quarters of every simulation tick, performs the retrieval of the events contained in the `retrieveBuf` buffer. In the worst scenario, $k + 2 * r$ rounds are required to retrieve an event.
In particular, the retrieval of an event consists of four steps:

- first of all the node waits `k` rounds. In fact, the event may be received in a subsequent gossip message from another node. In that case there is no need to perform the retrieval;
- if the event has not been received in `k` rounds, the node asks the agent from which it has heard about the existence of that event;
- if the event has not been received from that node in `r` rounds[9], a random node taken from the view is asked for the event;
- if the event has not been received in the subsequent `r` rounds, the node asks the agent that has generated the event, which can be identified from the event id[10].

Since the original paper does not specifies how the event should be retrieved, we have decided to generate an `Event` instance with the same id and age equal to the difference between the current round and round in which it was created. Moreover, since in the system there is no central authority with full knowledge of the nodes, in case the last step is reached the node considers the event as retrieved without effectively contacting the original source (it has no way to do it)[11].

[9]This may happen because the node does not have the event anymore or because the request/reply message has been lost.

[10]As described in [1] and in Section 2.3.1, the event id univocally identifies the agent that has created it.

[11]An alternative could be embedding a reference to the original source in the events

2.2.4 gossip()

The `gossip()` function, which is scheduled as the last operation of each round, manages the transmission of the gossip messages but also the node crashes. In fact, if the node is alive, it crashes with probability `crash_pr` and it sends a gossip message with probability $1 - $ `crash_pr`.
In the first case the `crashed` boolean flag is set to `true` and the recovery is scheduled after `recovery_interval` rounds (in the meanwhile the node does not perform any operation).
In the other case, first of all the unsubscriptions older than `unsubs_max_age` are deleted. Each unsubscription is associated to a timestamp that corresponds to the round in which it has been generated. Since this value is never updated, a node unsubscription is removed from the `unsubs` buffer of all the agents in the same round. After this purging, `F`[12] nodes are randomly selected from the view: they represent the targets of the gossip message. Hence, after updating the age of the events contained in the `events` buffer, the gossip message (`Gossip` class, Section 2.3.2) is created and sent to the target nodes through the `send()` method [13] Finally, the `events` buffer is cleared as described in the original paper.

2.2.4.1 receive()

The `receive()` method updates the content of the node's buffers using the information provided by the gossip message[14], which is basically a copy of the sender's buffers (as described in Section 2.3.2).
Firstly, the unsubscription information is processed: the nodes are removed from the local `view` and

[12]The parameter `F` corresponds to the fanout, i.e. the number of nodes a gossip message is sent to.

[13]Basically, the `send()` function calls the `receive()` procedure on each of the target nodes.

[14]This is done only in case the node is alive. Otherwise, the method does nothing

subs (if present) and they are added to the local unsubs (if not present).
Then it's the turn of the subscription information: in this case, the nodes are added to the local view and subs if they are not present; otherwise, their frequency is incremented by one.
After that, the events information is processed: if an event is already present in the local events buffer, its age is set to the maximum value between the two versions; otherwise, in case the id of the event is not present in the local eventIds buffer (which means that it has not been delivered yet), it is added to the local events buffer and its id is inserted into eventIds.
Finally, if the events history (eventIds) of the gossip message contains an event id that is not present in the local history, a new Element is added to the retrieveBuf buffer[15]. This element will be processed by the method described in Section 2.2.3 in the following round.
It is worth to highlight that in case one of the buffer exceeds its maximum size after an update, it is purged applying the corresponding criterion (as described at the beginning of Section 2.2).

2.3 Support classes

This section briefly describes the structure of the three classes that are used by the agents to exchange and retrieve information, i.e.: Event, Gossip and Element.

2.3.1 Event

The Event class defines the structure of the events that are broadcast by lpbcast. Each event is characterised by a unique id, which results from the concatenation of three elements separated by the ':' character: the first one is the id of the node that has generated it, i.e. an integer value ranging from 0 to nodes_size − 1; the second one is the round number at which it has been created, i.e. the simulation tick rounded up; the last one is an integer value ranging from 0 to broadcast_rate − 1, which uniquely identifies the event inside the round. Instead, the other field represents the event's age, which is needed for age-based message purging.

[15]The structure of Element is described in Section 2.3.3.

2.3.2 Gossip

The Gossip class defines the structure of a gossip message, which is basically a copy of the node's local buffers. Hence, a Gossip instance consists of: three lists, which contain events (events), events ids (eventIds) and subscription information (subs); one key-value map (unsubs), which is used for the unsubscription information (the value corresponds to the creation timestamp); a reference to the gossip sender (which is needed for the event retrieval procedure).

2.3.3 Element

The last support class is used by the processRetrieveBuf() method to perform the events retrieval. Basically, an Element contains information like the id of the missing event, the round at which the node has heard about that event for the first time and a reference to the sender of the corresponding gossip message.

3 Experimental Results Analysis

The implementation of *lpbcast* has been tested with several runs, in order to have an insight on its performance with different configurations. The runs have been carried out in the Repast framework, using as default parameters the ones in Table 1 with different interesting variations that are specified in the re-

lated sections. These analyses are designed to evaluate the performance for systems with different quantity of nodes, view size (l) and fanout configurations. The first one is particularly important, as it highlights the scalability of the protocol (Section 3.1). The next two show the impact of two of the main parameters, i.e. fanout (Section 3.2) and view size (Section 3.3). Finally, some analyses on the propagation of membership information (Section 3.4) and on the event delivery probability (Section 3.5) are presented.
It is worth to remember that all the analyses have been performed on a version of *lpbcast* with both optimizations, unsubscription and event retrieval procedures implemented.

Table 1: Default protocol parameters for experimental runs.

Parameter	**Value**
Average frequency scaling factor (k)	0.5
Broadcast Rate	30
Fanout (F)	4
Initial view size	2
Max age for unsub element	10
Message loss probability	0,05
Node crash probability	0,01
Number of nodes	125
Ticks before another unsubscribe	20
Ticks needed for recovery (after a crash)	4
Ticks the node stays unsubscribed	15
Out-of-date event interval	7
Size of eventIds	180
Size of events buffer	60
Size of subs buffer	60
Size of unsubs buffer	50
Timeout for message retrieving (r)	3
View size (l)	20
Waiting rounds for message retrieving (k)	3
Simulation ticks	120

3.1 Scalability Analysis

The scalability performance analysis runs (`SC`) have been executed according to Table 2. Figure 1 shows that the event propagation throughout rounds on different network sizes is very similar, which verifies the protocol scalability capabilities. Furthermore, Figure 2 evinces how fast events spread throughout the nodes: all the runs take a reasonable number of rounds w.r.t. the node quantity.

Table 2: Variations of `SC` runs on the default parameters.

Run	**Number of nodes**
`SC00`	125
`SC01`	250
`SC02`	500

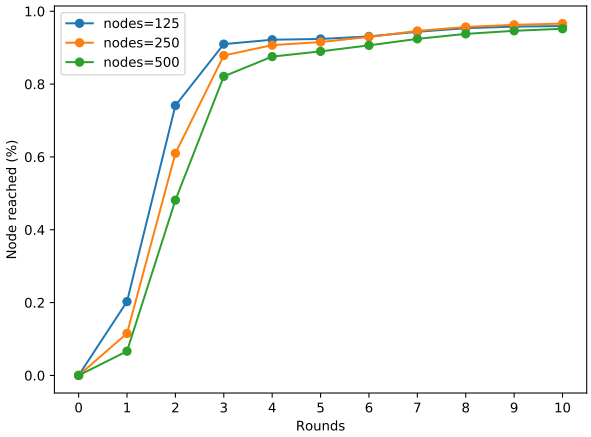

Figure 1: Propagation of events throughout rounds, considering `SC` runs.

3.2 Fanout Analysis

As explained in Section 2.2.4, the protocol has a fanout parameter F specifying the number of nodes

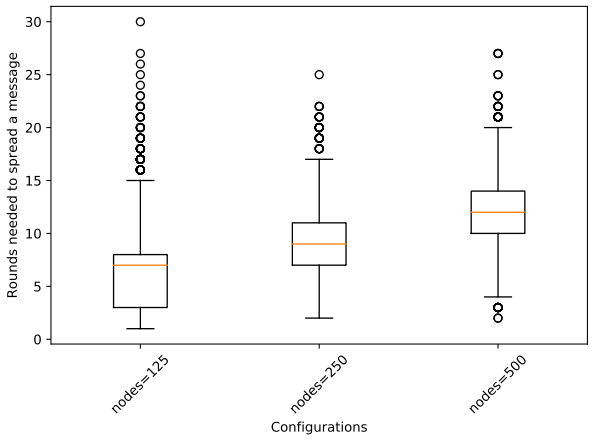

Figure 2: Number of rounds needed for an event to spread to 90% of the nodes in SC runs.

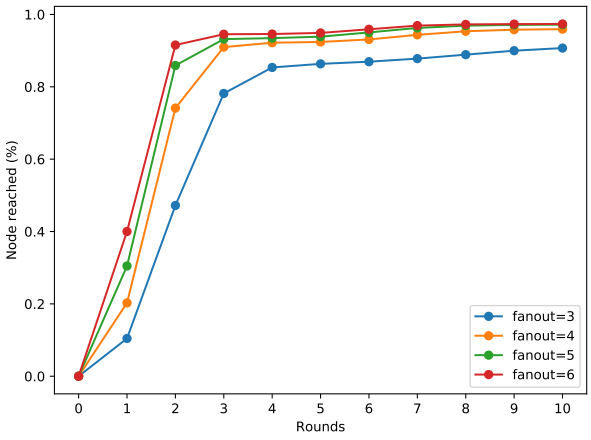

Figure 3: Propagation of events throughout rounds, considering FO runs.

in the view to send a gossip message to. Different fanout values have been tested, as listed in Table 3. As expected, this parameter has a visible impact on the protocol performance, as event information is propagated at different rates. In fact, as it can be observed in Figure 3 and Figure 4, the number of rounds needed for an event notification to spread around the system is higher with lower fanout values, consequently impacting on the coverage (e.g. FO00 plateau in Figure 3 is lower).

Table 3: Variations of FO runs on the default parameters.

Run	Fanout(F)
FO00	3
FO01	4
FO02	5
FO03	6

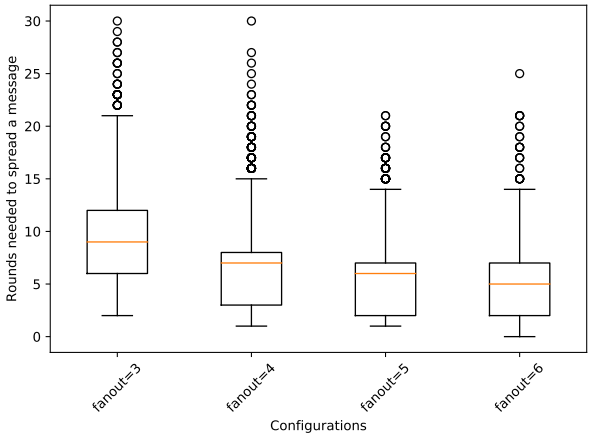

Figure 4: Number of rounds needed for an event to spread to 90% of the nodes in FO runs.

3.3 View Size Analysis

Contrary to what one might think, a higher view size (l) does not imply better protocol performance. In fact, as shown by the VS runs listed in Table 4 in the Figure 5 and Figure 6 plots, all the considered view sizes have similar performances, with

almost identical event propagation trend. These results evince the independence between view size and event propagation, accordingly to what stated in the original paper [1].

Table 4: Variations of VS runs on the default parameters.

Run	View Size (l)
VS00	10
VS01	15
VS02	20
VS03	25
VS04	30

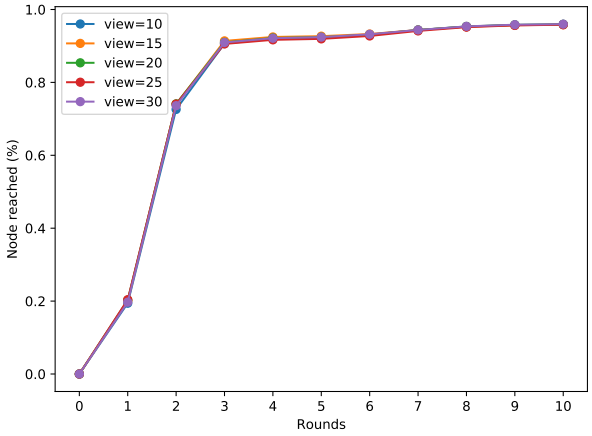

Figure 5: Propagation of events throughout rounds, considering VS runs.

3.4 Membership Information Propagation

Another interesting analysis is the one related to membership information propagation, that is, how well nodes get to know subscribing ones (re-subscribing in our case). Some samples for this type of occurrence have been collected with node size

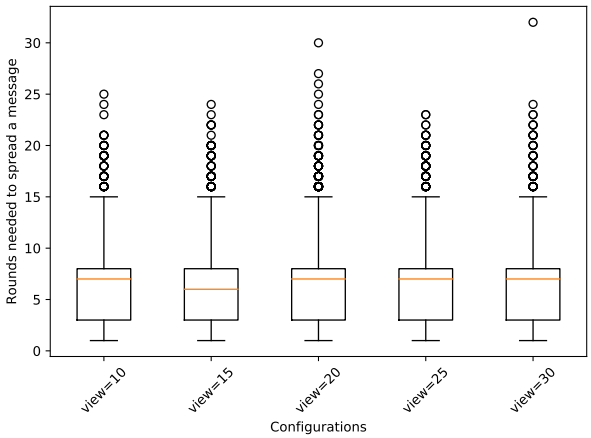

Figure 6: Number of rounds needed for an event to spread to 90% of the nodes in VS runs.

equal to 125 and 500. The results are shown in Figure 7 and Figure 8: basically, the spread of membership information turns out to be very fast. It is important to highlight that this is due to the implementation of frequency-based membership purging (one of the optimizations described in the paper).

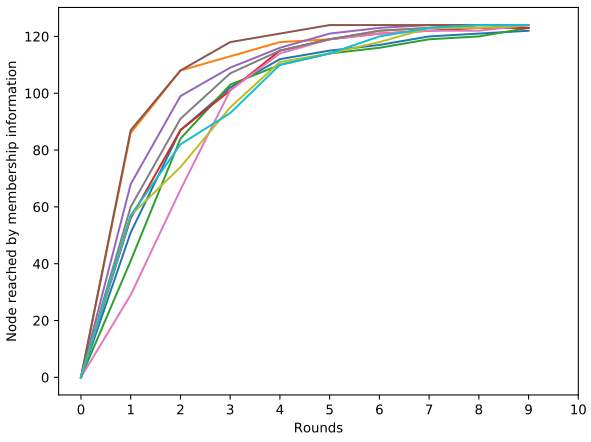

Figure 7: Samples of re-subscribing information propagation with default parameters (125 nodes).

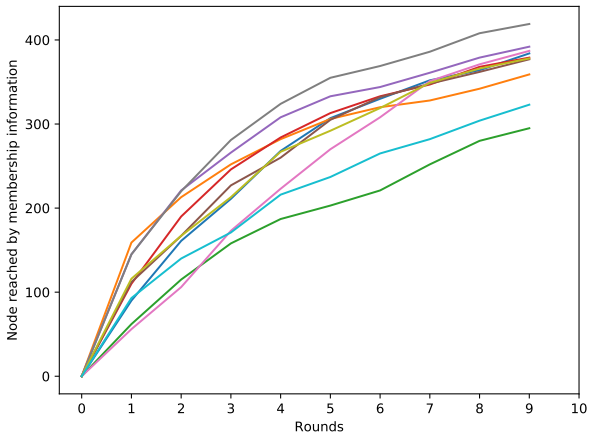

Figure 8: Samples of re-subscribing information propagation for 500 nodes.

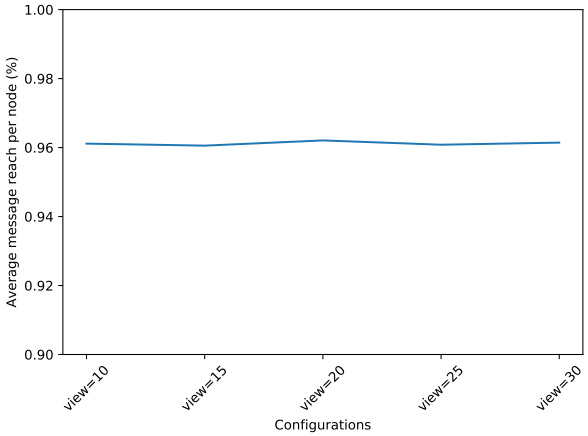

Figure 9: Event delivery probability for `VS` runs.

3.5 Event Delivery Probability

The percentage of events delivered to the application layer by a node, which corresponds to the probability of a node delivering an event, is intuitively related to the view size and fanout parameters. The effect of these parameters can be observed in Figure 9 and Figure 10: it turns out that the view size does not influence the delivery probability, while the fanout has a significant impact. In fact, a higher fanout setting leads to a drastic improvement, but it is important to consider also that higher values will bring more redundant messages.

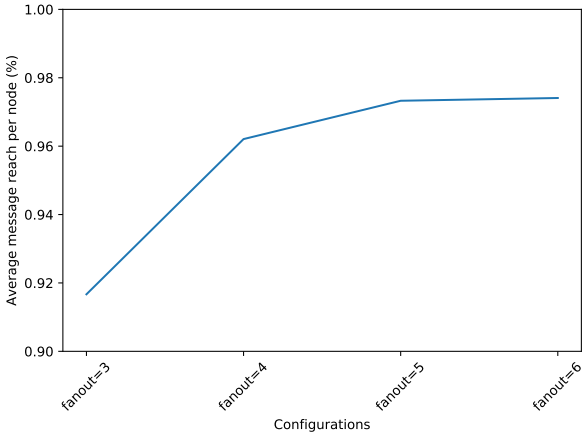

Figure 10: Event delivery probability for `FO` runs.

4 Guide to the simulator

In order to run the simulator, the first step consists in importing the project. After that, the `Lpbcast` model has to be run: the default graphic interface of Repast Symphony will be shown. At this point, the simulation parameters can be set in the *Parameters* panel[16]. Instead, the *Scenario Tree* panel allows to select the data to be saved (`Data Sets` item) and the location (`Text Sinks` item). As regards the former, it is possible to choose the methods to be called on the agents and the time to do that. As for the latter, the data set fields to be stored in the file can be selected. Once all this has been done, it is

[16]The complete list of parameters can be found in Section 3.

possible to start the simulation. Actually, the tick at which to pause or stop it can be set in the `Run Options` panel.
For what concerns the data analysis part, the project is already set to log raw simulation information in the *output_data* folder, following a certain format. These files can be loaded with the *lpbcast_analyzer.py* python3 script, specifying some comma-separated plot requests as the first parameter, with a sequence of log files to load as the rest of the parameters. Plots are visualized and saved in the *plots* folder. Valid plot requests are:

- *propagation_100*: boxplot of the number of rounds needed for an event to reach all the nodes (only considers events that reached all the nodes);
- *propagation_90*: boxplot of the number of rounds needed for an event to reach 90% of the nodes (only considers events that reached at least 90% of the nodes);
- *round_propagation*: average number of nodes reached by an event in the rounds after the generation;
- *coverage*: bar plot showing the percentage of events reaching 100%, 80%, 60%, 40%, 20% of the nodes;
- *coverage_top50*: upper 50% portion of the *coverage* bar plot;
- *coverage_top10*: upper 10% portion of the *coverage* bar plot;
- *event_delivery*: average percentage of events delivered by a node;
- *membership_propagation*: samples of resubscribing information propagation over the first 10 rounds;
- *all*: generate all the plots.

References

[1] P. T. Eugster, R. Guerraoui, S. B. Handurukande, P. Kouznetsov and A.-M. Kermarrec, "Lightweight probabilistic broadcast", *ACM Trans. Comput. Syst.*, vol. 21, no. 4, pp. 341–374, Nov. 2003, ISSN: 0734-2071. DOI: 10.1145/945506.945507. [Online]. Available: http://doi.acm.org/10.1145/945506.945507.

[2] N. T. Bailey *et al.*, *The mathematical theory of infectious diseases and its applications*. Charles Griffin & Company Ltd, 5a Crendon Street, High Wycombe, Bucks HP13 6LE., 1975.