Simulating Lightweight Probabilistic Broadcast in Repast

Enrico Marchi - enrico.marchi-2@studenti.unitn.it (211464)
Giacomo Vitali - giacomo.vitali@studenti.unitn.it (266445)

Introduction

This project consists of the implementation of a version of *Lightweight Probabilistic Broadcast*, *lpbcast* from now on, a scalable, gossip-based broadcast algorithm described and analyzed in [1].
The main feature of *lpbcast* is its scalability in term of memory consumption for membership management and message buffering, this is obtained by restricting the nodes to keep a partial, probabilistic view of the system and by removing the most propagated messages from the buffers.
In the context of *lpbcast*, the messages that are generated, broadcast and delivered by the nodes are called *events*; the nodes, are organized in a network and can communicate by sending (generic) *messages* to any other node present in their view of the system.
We modelled the network of nodes as an agent-based system in Repast Symphony, using the Java programming language. Through multiple simulations of the model, we evaluated the performance of the *lpbcast* and studied how its behaviour depends on a set of parameters.

Implementation

An instance of the `LPBCastNode` class represents an agent of the system, his behaviour is specified by the `step()` method, scheduled to be executed one time for each step/tick of the simulation, and by the `receiveMessage()` method, which is called asynchronously when the node receives a `Message` through the `Channel`.
At each step the node executes the following substep:

1. Randomly generate a new `Event` with a probability specified by a parameter (see the section on model parameters).
2. Try to retrieve events that it has not yet received, using information saved in the `retrieveBuffer`.

One time during each round (the duration of a round in simulation ticks is specified by a parameter) the node executes a third action:

3. Send a Gossip message to a subset of nodes contained in the `view`.

The internal state of each node consists of three sets of node identifiers, `view`, `subs` and `unsubs`, a set of event identifiers, `eventIds`, a map between event identifiers and events, `events`, and the `retrieveBuffer`. With the implementation of the optimization on frequency based membership purging, the state is extended with two more maps (mapping node IDs to integer values), `viewFrequencies` and `subsFrequencies`.

There are other entities that act as "meta-agents" like the `Channel`, which manages the transmission of the messages and delivers them at the beginning of the simulation step, and the `NodeManager`, which dynamically manages the set of active nodes in the system.

The Java classes that constitute the model are organized in packages: `communication`, `lpbcast`, `lpbcast.messages` and `util`.
The communication on the network is implemented in the `communication` package, containing the following classes:

- `Channel`: It manages the transmission of the messages between nodes, in particular, it implements the random latencies and the possibility of network failures (lost messages).
- `Node`: It is an abstract class extended by `LPBCastNode` representing a generic node in a distributed setting.
- `Message`: It is a generic message exchanged on the network.

The `lpbcast` package contains the core of the algorithm:

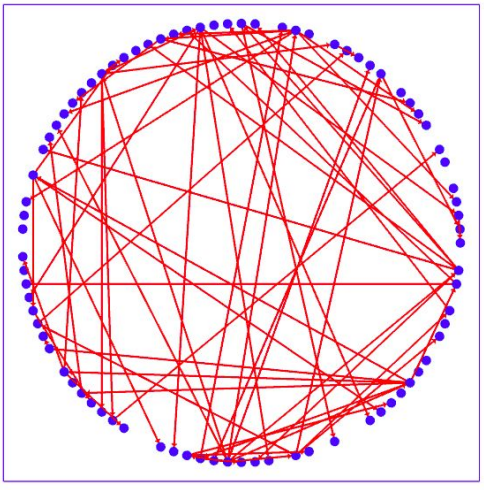
- `LPBCastNode`: It is the concrete node implementing *lpbcast*
- `LPBCastBuilder`: It builds the Repast model for the simulation
- `NodemManager`: It manages the random creation and removal of nodes during the simulation
- `Event`: It represents a message generated by a node and broadcasted on the network
- `EventId`: It contains the proper (integer) identifiers of an event and the identifiers of its original source node
- `Element`: It is an element of the retrieveBuffer, it used by a node to keep track of the information about an event that it has not yet received

The `lpbcast.messages` package contains all the possible `Message` subclasses that the nodes can send and receive:

- `Gossip`
- `EventRequest`
- `EventResponse`
- `Unsubscription`

The `util` package contains various classes used to support the simulation and the analysis, the most important ones are:

- `Visualizer`: displays the current nodes and periodically selects three events and represents their broadcast as graphical networks in different colours (green, red, orange).
- `PersistencyDetector`: Keeps track of how long the individual events are present on the network before expiring.
- `Logger`: Logging of information as the creation and the delivery of the events, used in analysis



The visualization of the event broadcasts through the graphical network is controlled by the parameters `green_event`, `red_event` and

`orange_event`, which specify the ID of the highlighted events.

The behaviour of the Repast model is controlled by the following parameters (can be set on the Repast GUI):

- `round_ticks` controls the duration of a gossip round (in simulation ticks)
- `max_nodes`, `min_nodes`, `initial_nodes`
- `Probability_create_node`, `probability_remove_node` control the rate (from 0 to 100) at which nodes are created and removed at runtime
- `probability_crash` controls the fraction of nodes that, when removed, crash and do not send an explicit unsubscribe message across the network
- `probability_network_fail` controls the fraction of messages that are lost
- `max_channel_latency` controls the latency of the message arrival
- `max_eventIds`, `maxEvent`, `maxSubs`, `maxUnsubs`, `maxView` control the size of the component of internal state of each node (as described in the original paper).
- `max_gossip_target` controls the number of receivers of a single gossip from a node (fanout)
- `optimization1`, `long_ago` enable and control the age based event purging optimization
- `optimization2`, `k_avg` enable and control the frequency based membership purgining optimization

The base scenario of our simulations is defined by the following setting:

- `round_ticks=10`
- `max_nodes=100`, `min_nodes=80`, `initial_nodes=80`
- `probability_create_node=5`, `probability_remove_node=5`
- `probability_crash=50`
- `probability_network_fail=10`
- `max_channel_latency=5`
- `max_eventIds=100`, `maxEvent=60`, `maxSubs=20`, `maxUnsubs=20`, `maxView=20`
- `max_gossip_target=10`
- `optimization1=true`, `long_ago=10`
- `optimization2=true`, `k_avg=0.8`

Analysis of the simulations

The main metrics used to evaluate the simulations are the average delivery rates and the persistency of the events.

Average delivery rate

The delivery rate of an event represents how an event progressively infects the network (by being delivered by the nodes) as a function of the time passed from the generation of the event.
More precisely, for an event generated at time *t*, the delivery rate at time *i* is the fraction of nodes that have delivered that event at time *t+i*, considering only the nodes that were active at time *t* (nodes that joined the network later are not interested in the past history of events).
The average delivery rates is computed by averaging the rates for the set of events that were broadcast in a simulation, restricting the analysis to an interval of 80 simulation ticks / 8 gossip rounds from the generation of the events.
This analysis is based on the logs containing the event creations and event deliveries, the logs are processed with a Python script (log_analysis.py). The graphs are obtained by combining multiple simulations data with another script (deliv_rates_analysis.py).

Persistency

The persistency represents how long the nodes kept re-transmitting a specific event by including it in the gossip messages and keeping it in the event buffer. In particular, for an event generated at time *t*, its persistency is computed as the time that has passed from *t* when it is deleted from the event buffer of the last node that was keeping it in memory (and there are no gossip message in transit that contain it), from this moment onward the event is considered expired and no other node can receive and deliver it.
The analysis on the persistency is based on the information saved by the PersistencyDectecor to a log, which is then analysed with a script (persistency_analysis.py).

Analysis on the size of the gossip targets

The first set of simulations analyzed focuses on the impact of the number of destinations of single gossip (the size of the gossip targets) and the effects of the optimizations.
The graph on *Fig 1* shows the average delivery rates of six simulation; the *x* axis represent the number of ticks after the generation of the messages; the *y* axis represent the average of the fraction of nodes that have delivered the messages until the respective past tick. The six curves share some common parameters (see the base scenario previously described) and they differ in the size of the gossip targets (5, 10, 20) and in the presence of the optimizations (opt or no opt).

Finally, the plots in *Fig 2*, *Fig 3* and *Fig 4* show how the persistency of the events is distributed as the size of the gossip targets changes. The three cases can be approximated with three normal distributions with different means and variances:

- Size 5: $\mu = 56$, $\sigma = 73$
- Size 10: $\mu = 43$, $\sigma = 58$
- Size 20: $\mu = 32$, $\sigma = 28$

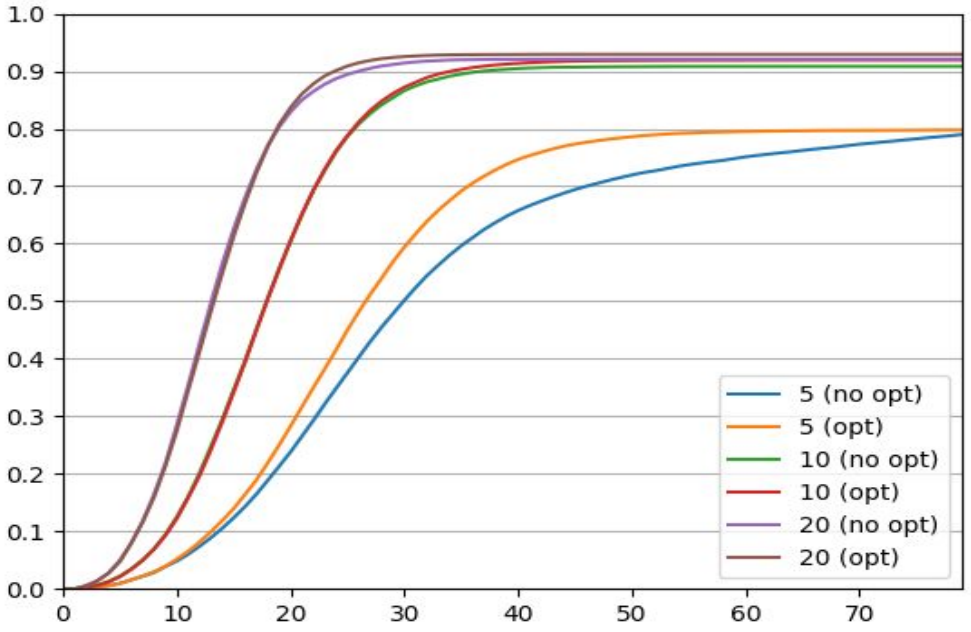


Fig 1: Average delivery rates with different gossip target sizes and different optimizations

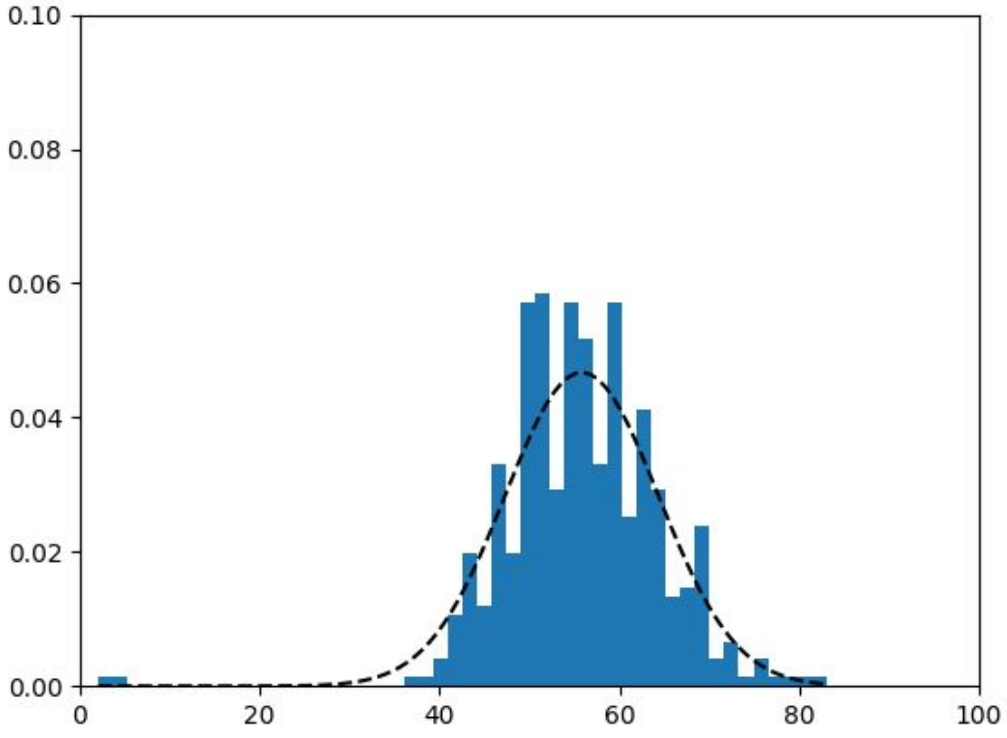


Fig 2: Persistency of the events with gossip target size set to 5

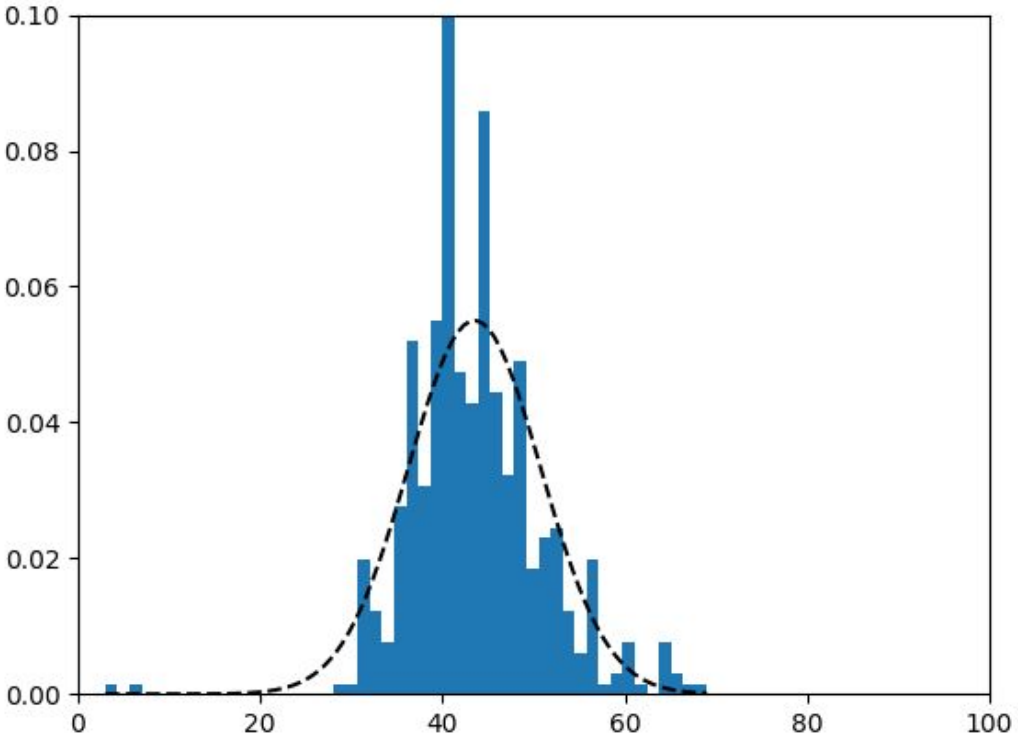


Fig 3: Persistency of the events with gossip target size set to 10

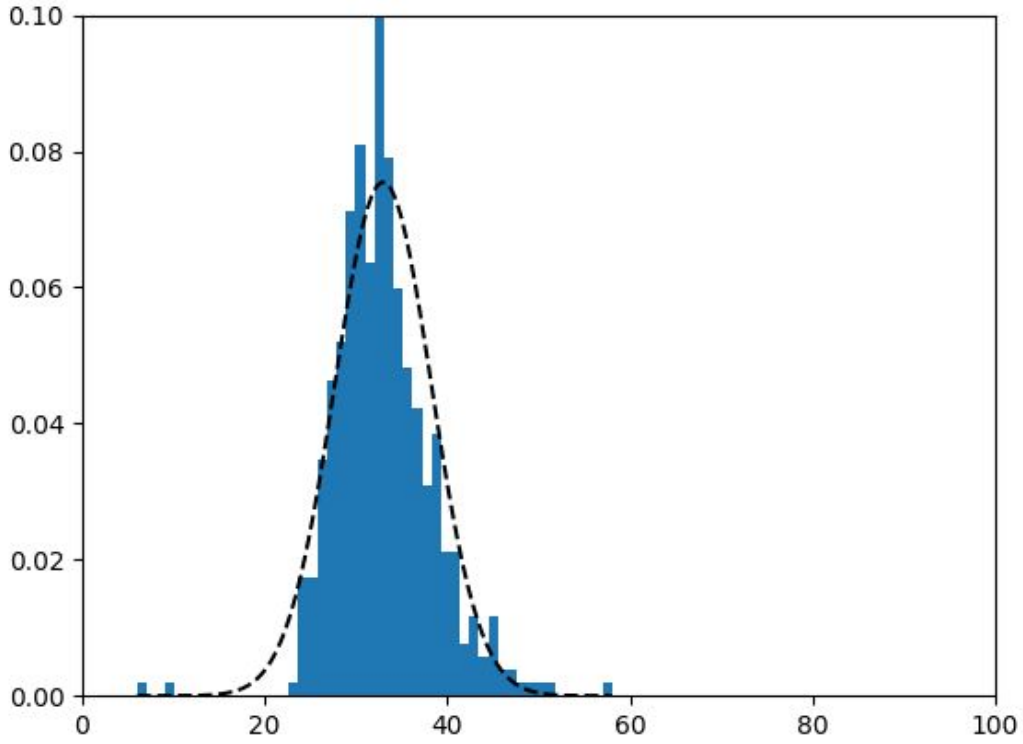


Fig 4: Persistency of the events with gossip target size set to 20

Analysis on the robustness

In the graph *Fig 5* the parameters are the same of the base scenario, except for the probaility of network failurs causing a fraction of the messages to be lost (0%, 10%, 25%,50%,75%). The graph shows how the the system is able to cope with noisy channels.

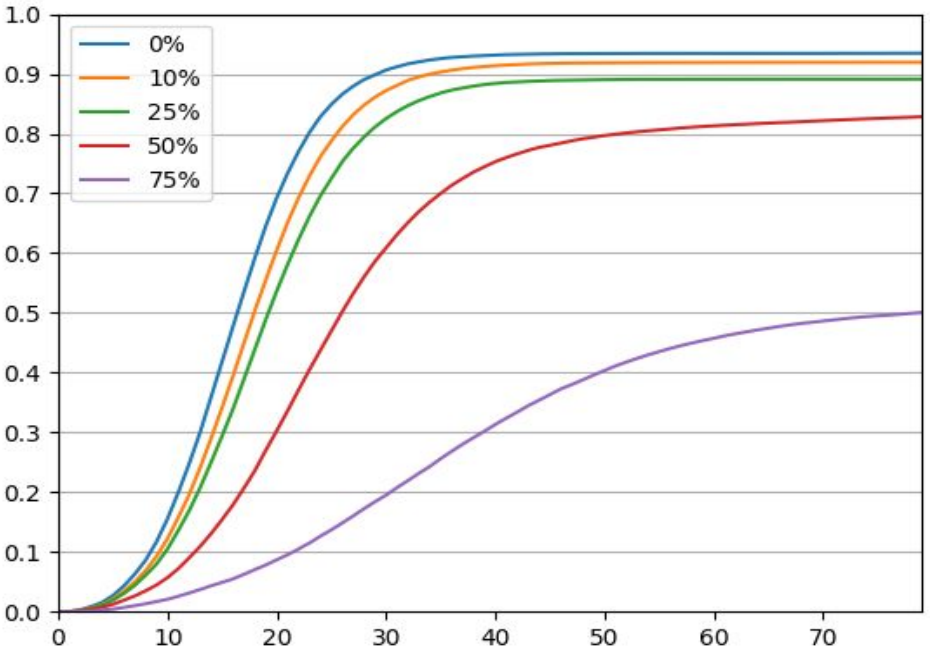


Fig 5: Average delivery rates with different probabilities of network failures

In the graph *Fig 6* it's the same as before, except the parameter "probability crash": the percentage refers to the number of nodes that have crashed without sending an explicit unsubscribe message across the network.

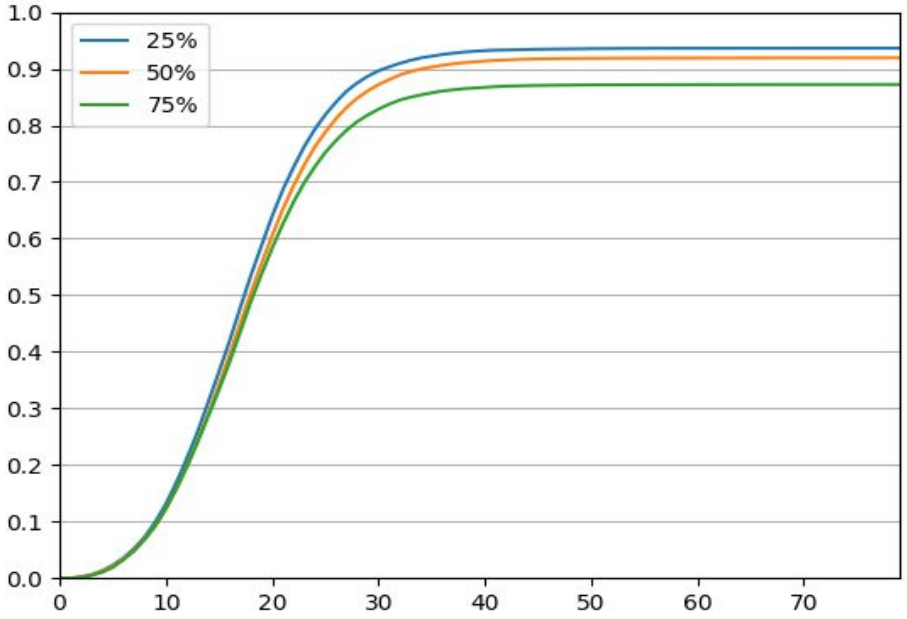


Fig 6: Average delivery rates with different (node) crash probailities

Analysis on the scalability

The graph *Fig 7* combines the data from multiple simulations with different amouns of nodes: the *x* axis stands for the average number of active nodes in the simulation, while the *y* axis represents the average number of gossips required to infect the 90% of the network for the respective simulation.

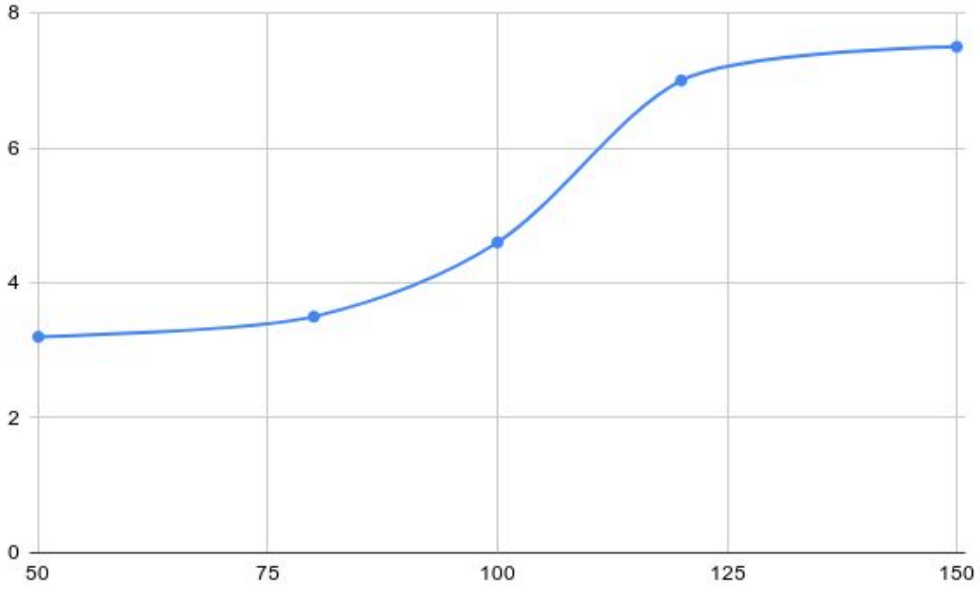


Fig 7: Average number of rounds to infect 90% of the network

References

[1] Eugster, P. T. *et al.* (2003) 'Lightweight probabilistic broadcast', *ACM Transactions on Computer Systems*, pp. 341–374.