Lightweight Probabilistic Broadcast Protocol Implementation

Marian Alexandru Diaconu, Valentino Armani

November 2019

The following report contains a detailed implementation explanation of *lpbcast* algorithm. Moreover, this document does not describe specifications neither contains pseudo-code of the theoretical algorithm, hence for further details of the protocol, please refer to [1].

1 Multi-Agent System

The broadcast protocol is modeled as a Multi-Agent System (MAS). Each participating node corresponds to an autonomous agent which has partial knowledge of the global environment and system state. The system state is composed by the union of all node states. A node interacts with other nodes via messages used to transmit data. In particular, both the generation and the reception of a message may lead to an internal state change. Messages are transmitted through the network using a router, which allows total encapsulation of a node's state with respect to the other nodes. On top of this basic simulation environment, we integrate a virtual application layer using a special agent which does not participate to the protocol. This agent is called AppNode and acts on behalf of all nodes simulating application-dependent behavior such as broadcast message generation.

1.1 Actions Scheduling

Repast Symphony scheduling mechanism allows managing execution of actions according to a simulation clock. The clock, measured in "ticks", is incremented at the completion of the execution of all the actions scheduled at that clock. Specifically, all the protocol-related actions of the agents are scheduled according to Table 1.

At tick 0 the environment is populated with new agents and other data structures used to analyze behavior and data produced by the participants. On the next tick Node and AppNode agents are initialized as well. Consistently with lpbcast protocol, each process periodically (every $T\ ms$) generates a gossip message that is gossiped to F other participants. Our implementation forces the

Action	Starting Tick	Interval
init(environment)	0	0
init(agents)	1	0
send(gossip)	2	2
broadcast(message)	3	2
receive(gossip)	gossip.creationTick+oneOf(1,3,5)	0

Table 1: Schedule times for protocol actions.

nodes to generate a gossip message every 2 ticks, starting from tick 2 as described by the above table. Basically, with this scheduling mechanism, we ensure that a round, as defined in [1] contains all the actions performed between an even tick number and the following even tick. In this vision we schedule receive actions always at odd ticks. Hence the receive tick of a gossip message depends on the creation tick of the message and according to the formula from table 1 corresponds to the creation tick of the gossip message plus a propagation delay between 0 and 2 rounds (1,3 or 5 ticks after the gossip creation tick).

2 Node Agent

A process participating in the broadcast algorithm is modeled as an agent of type *Node*. Every node initially knows only a fixed number of other participants, has limited buffers and has an internal state. Moreover, a node can be in different states:

- **Subscribe State**: This represents the initial state of a node agent that allows him to participate to the protocol. In particular, a node in this state is able to send and receive gossip messages but also contact other nodes and ask for some re-transmission service. Moreover, it can generate broadcast messages. From this state a node can transit in one between **Unsubscribed** or **Crash** states.

- **Unsubscribed State**: In order to enter this state an agent must to insert in its unsubscriptions buffer a new unsubscription containing itself. This allows declaring the intention to leave the topic. Subsequently, the node discards all the information apart from delivered events and some neighbors. The delivered events are kept to avoid delivering of already delivered events. The neighbors instead are kept to avoid a central authority which should manage initialization. From this state a node can transit to **Subscribed** state.

- **Crashed State**: This state simulates a node failure. A node in this state does not participate to the algorithm (ignores received messages) and loses all its local information. It is assumed that also in case of failure knowledge about events delivered and known processes are maintained. From this state a node can transit to **Subscribed** state.

Changes in state are managed by the AppNode that sends particular types of messages in order to signal to the node that some specific behavior is requested for the simulation.

2.1 Messages

The participating nodes communicate between them via messages. These messages are the main element that brings forward the protocol global state. More precisely the messages that are exchanged are of three types:

- **Gossip Message**: This message is used to exchange information about the algorithm status between the nodes such as new events to be delivered or old delivered events. In particular, both the send and the reception of this message change the internal knowledge of the node. This type of message is periodically sent to some random participants.
- **Re-transmission Request and Reply Messages**: When a node discovers, through a gossip message that exists events that have been delivered by other nodes but not by himself, it will schedule a re-transmission request to the sender of the gossip which will be performed only if in the meantime the event is not received. If the sender of the gossip does not have the event anymore, the node waits for another set of rounds and then requests re-transmission from a random node. If not even this request is successful, finally the generator of the event is involved

3 Router

Messages are transmitted between nodes by simulating a full mesh virtual underlying network. In order to encapsulate the local state of a node and avoid to pass object references between agents, a node is allowed to know only the identifier of other participants. The identifier acts as the node's network address and can be used to route messages. For this reason, a global component called Router is implemented which behaves on behalf of the full mesh virtual network. The router is encharged of the following functionalities:

- **Message Transmission**: When a node sends a message, the router locates the destination node and schedules the actual reception of the message.
- **Message Propagation Delay**: The receive of a gossip message is delayed by a random round number between 0 and 3 as stated in section 1.1. The other messages are received instantly because they are considerably less.
- **Message Loss**: For some simulations the router simulates message loss for every gossip message influenced by the *message loss rate* configuration parameter present in table 5.

4 Application Node Agent

The system has a single coordinator agent called AppNode. This agent is in charge of simulating a unified application layer of all nodes. Basically, it communicates with the node agents imposing them to perform actions such as: event generation, crash, subscribe or unsubscribe. The AppNode manages also some visualization aspects; when a node delivers a message, it does so by signaling to the AppNode that some event has been received. That way, the AppNode is able to keep track of the effective global state of the system. This allows the AppNode to manage more fashion aspects of the simulation such as run-time visualization of node states and transmission graph. Moreover, the global vision of this supernode enables easy to manage and extend modules for computation of statistics or other types of feature extraction from the system.

5 Analysis

The main focus of the analysis was correctness and efficiency checking of the algorithm. To do this we have chosen some measurement metrics from the original paper and implemented them. All the configurations are the same as in the original paper. The results of our analysis are presented starting from subsection 7.1 till the end of the section. However, we reserve the next subsection to briefly present how to interact with the simulator and which are the actual steps needed to configure the environment, run the protocol and output results.

5.1 Simulator configuration

Our simulator is based on Repast Symphony, an agent-based modeling toolkit and cross-platform Java-based modeling system [2]. This technology allows represent graphically a complex system with the usage of efficient visualization features and also plot computed statistics, such as histograms or time series, in order to allow easier analysis.

5.1.1 Semantic Interpretation of Repast Network Projection Visualization

Repast allows to collect data regarding the global state of the system and visualize at different abstraction levels. For example, the gossip transmission graph can be inspected to check if partitions between views have been created or to check how dense becomes the re-transmission graph. More precisely our simulator displays, as yellow arrows, the directed gossip transmission graph. Each edge represents a gossip message which transits from a source node to a destination one and each round the edges between nodes may change due to the transmission of new messages. Each node has a color which describes one of the possible node states as stated in Figure 1.

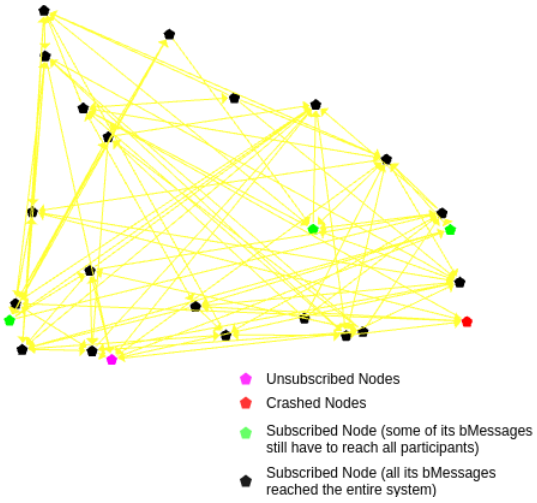

Figure 1: Semantic interpretation of Repast network projection visualization.

5.1.2 Data Collection

The Repast APIs force to create a dataset structure which fetches data from the simulator every tick. This becomes very useful when you want to plot for example the average number of deliveries each round. However, if you want to compute statistics at tick x that are related on some statistics computed at time $x - C$, where $0 < C < totalNumberOfRounds$, you should store statistics for each tick, which may consume a lot of memory but also become difficult to manage. It is easier to collect data for some period of time and then at some point compute and output the results. In our case we have two types of computed statistics:

1. **Expected number of infected Processes for a given round**: Computed in one shot at a specified tick count, outputted using Java *System.out.println()*, collected and used to plot graphics using *gnuplot* [3].

2. **Delivery ratio**: computed using the Repast data collection mechanisms. We create a Repast Data Set and collect at the end of each tick the delivery ratio of each node, average it among all nodes and plot the result in function of the tick count as a time series.

Hence, in order to output the results of the experiments related to the expected number of infected processes for a given round, you have to check the standard output of the java program and analyze the numbers or use *gnuplot* or similar tools to plot charts. If you want instead to check the average delivery ratio you can simply inspect the graphic produced by Repast simulator in the GUI.

5.1.3 Configuration Parameters

The simulator exposes parameters that can be set to recreate specific configurations. These configurable variables map the parameters introduced in the original paper [1] and influence the algorithm's behavior. The most relevant parameters are the ones cited in the configuration tables of this section.

number of nodes	125
fanout	3
view size	20, 25, 30
messages/round	1
analized messages	50
buffer size	30
optimizations	False

Table 2: Configuration of view size irrelevance experiments.

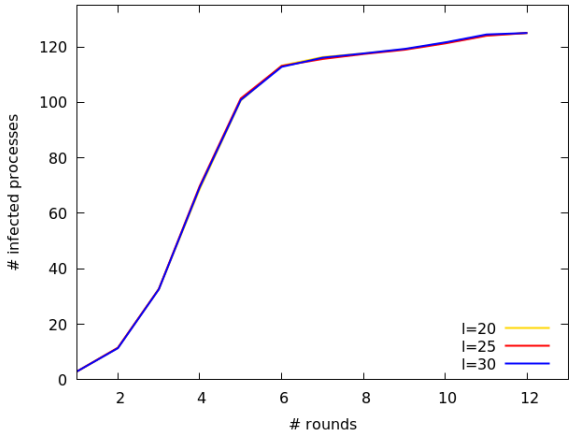

Figure 2: Number of rounds necessary to infect the system with different view buffer sizes (they are almost the same).

5.2 View Size Irrelevance

We want to understand what is the expected number of rounds that a broadcast message takes to reach all the nodes. The analysis is done using the configuration from Table 2 and follows the following pattern:

1. Each round a random process generates a broadcast message
2. The system is left running solo to stabilize for 100 rounds
3. Then 50 messages are collected and statistics are calculated

Basically, we store for each message all the receivers along with their reception round and compute for each round the average number of receptionists. Figure 2 reports the simulation results obtained for different values of the view size in a system of 125 processes. We can conclude that the described implementation does not suffer in terms of infected processes. Moreover, in 10 rounds, almost every node is infected.

number of nodes	125
fanout	3, 4, 5
view size	30
messages/round	1
analized messages	50
buffer size	30
optimizations	False

Table 3: Configuration of fanout relevance experiments.

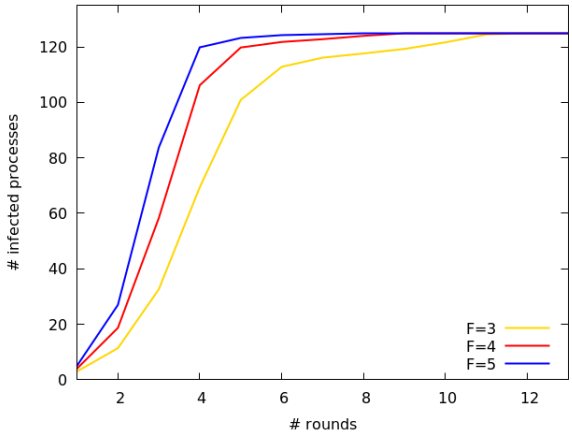

Figure 3: Expected number of infected processes for a given round with different fanout values.

5.3 Fanout Relevance

The *fanout* represents the number of processes that are randomly selected by a node to gossip with. There is a relation between it and the number of rounds needed by a message. The analysis was computed using the same pattern of previous experiment but with another configuration (see table 2). It can be observed in figure 3 how increasing the fanout the number of rounds necessary to infect all processes decreases, hence the implementation behaves as expected.

5.4 System size

The size of the system is another aspect that has been analyzed. Moreover, a positive aspect of our implementation is that it resists to activity of 1000 nodes which exchange messages continuously. The experiment is done performing ten times one of the previous experiments, and checking when the computed number of infected processes reaches 0.99 of the system size. It can be noticed in figure 5 how the number of messages needed to reach 99% of the processes has a logarithmic trend in function of system size and is equal to 16 when there are 1000 nodes participating to the algorithm.

number of nodes	100, 200, ..., 1000
fanout	3, 4, 5
view size	30
messages/round	1
analized messages	50
buffers size	30
optimizations	False

Table 4: Configuration of experiment on system size.

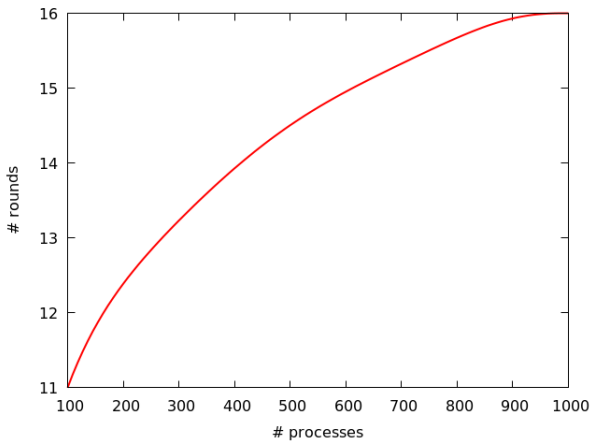

Figure 4: Expected number of rounds expected to infect 99% of the system, given a system of size n.

5.5 Delivery Ratio

In the following experiment we measure the delivery ratio, that is the ratio between the average number of messages delivered by a process per round and the number of messages broadcast per round. We have performed the experiment computing the delivery ratio for each process. At the end of each round the

average is calculated and plotted. The mechanisms used are the ones stated in subsection 7.1.

number of nodes	60
fanout	4
view size	30
buffers size	30
messages/round	30
analized messages	50
failure rate	0.05
message loss rate	0.1
optimizations	False, True

Table 5: Configuration of experiment on delivery ratio.

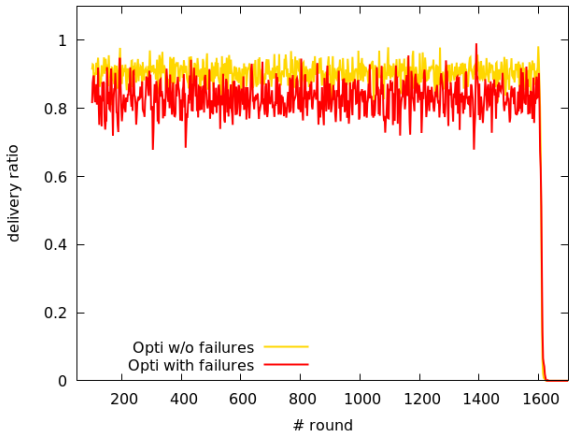

Figure 5: Impact of failures on overall delivery ratio. Process crash ratio = 5%, message loss ratio = 10%.

As it can be noticed from figure 5, the failure of the participants has some impact on the delivery ratio. If a node crashes, the number of times that an event is gossiped decreases for two reasons: (1) because the crashed node does not gossip anymore the received events during the last round, and maybe they are lost forever and (2) because the message-based communication model implies that gossip messages sent to crash nodes are lost forever (there is no acknowledgment). On the other hand, figure 6 shows that in our implementation, applying events buffer age purging optimization does not change the delivery rate. This is explained by the way our buffer implemented. We dedicate the entire next session to explain how this is done and why age purging does not affect the delivery rate.

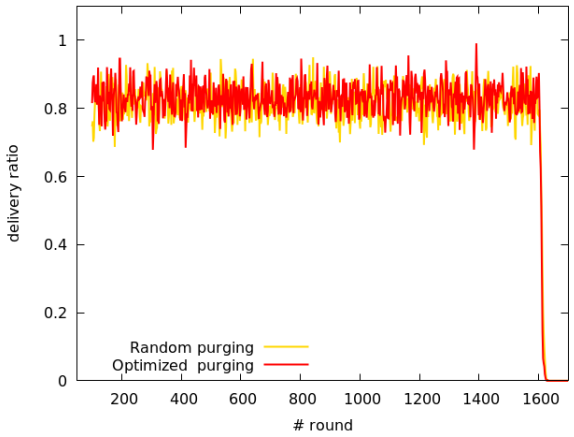

Figure 6: Delivery ratio of initial (random purging) and optimized (age-based purging) versions of lpbcast.

6 Delivery Optimization

We introduce this section because after implementing our version of lpbcast, we noticed that the delivery ratio was very high, around 0.85 as can be visualized in figure 6 where 30 new messages are generated each round.

6.1 Continuous Delivery and Continuous Gossip of Delivered Events Problems

The original paper does not take care of describing how to ensure a node to not deliver or gossip twice a message. More precisely the article contains the statements of table 6.

(1) Every such event notification is gossiped at most once. Older event notifications are stored in a different buffer, which is only required to satisfy re-transmission requests.
(2) ... every process stores identifiers of event notifications it has already delivered in a buffer named eventIds. We suppose that these identifiers are unique, and include the identifier of the originator process. That way, the buffer can be optimized by only retaining for each sender the identifiers of event notifications delivered since the last identifier delivered in sequence
(3) Multiple deliveries are avoided by storing all identifiers of delivered event notifications in eventIds...

Table 6: Original paper suggestions about how the protocol should behave.

Given that a node in order to re-transmit events should have stored such events, it is infeasible to a node to store all events that have been generated during all rounds and provide re-transmission service for every broadcast message that it has delivered. It is reasonable to think that a node is en-charged to re-transmit only events generated by him. This allows a node to store a limited amount of events and also delete them after some time. Moreover, *eventIds* buffer cannot by itself avoid multiple deliveries. Consider the case where a node has deleted an event that has already been delivered from the *eventIds* buffer. That node may receive the same event a second time, and it is very probable that this will happen given the random nature of the algorithm. When this happens, a node delivers the same event another time. Moreover, nothing stops him to re-broadcast it as well, triggering deliveries also on the other participants.

6.2 Optimized lpbcast

In order to avoid multiple deliveries and gossips, we focus on the citation number (2) of the above subsection. The original articles mentions some kind of optimization that can be done storing for each sender the identifier of the last event delivered in sequence. Our implementation in fact contains an optimized data structure named *EventIdsLog* which may match such optimization and is basically based on an extended *HashMap* defined as follows:

$$EventIdsLog : NodeIds \rightarrow OrderedList(EventIds) \tag{1}$$

In order to have globally unique event ids, such identifiers are composed by the concatenation of the generator node id and an internal counter local to the node which starts from 0. This allows the *EventIdsLog* data structure to organize the event ids in a hash table where the key is represented by the node id and the value is an ordered list of event ids. The first element of the list (the lowest) represents the last event delivered in sequence from that node and the remaining represents the events delivered out of sequence. When a node wants to check if an event has been already delivered, it has to check if the id is less than the last event delivered in sequence or it is equal to some of the events delivered out of sequence. Moreover, whenever it delivers a new event it has to add it to the *EventIdLog* and check if the last in sequence element has to be updated. The update is simply done by removing the old last in sequence and keeping the new one. Each node has internally an instance of this hash map and uses it to keep track of all messages that have been broadcast so far. Moreover, the continuous exchange of gossip messages even when no new events have been generated, guarantees a high probability that a node sees each generated message at least once and therefore it will schedule a retrieve. Once the retrieve is scheduled the future reception of the event is ensured because in the worst case the source node is contacted and asked for re-transmission. This mechanism ensures that even in presence of failures the probability of receiving all messages is very high. In relation with figure 6, we can now interpret the graphic better. The introduction of the *EventIdLog*-based optimization ensures

with a high probability that all the messages are delivered which are highly related to the numbers of events generated each round. In fact, in the average case, a node delivers many of the messages plus some re-transmission, therefore a delivery ratio around 0.85 seems reasonable for this architectural choice and explains why the implemented optimizations do not have effect on the system.

7 Conclusions

This report presents our implementation of lpbcast protocol. We explained the main architectural choices of the simulation environment which has been modeled as a multi-agent system. Further, the implementation has been tested and analyzed to first check protocol correctness and then measure its efficiency and performance. Moreover, we give our interpretation for the analysis results and explain the problem of continuous delivery and continuous gossip of delivered messages problem. In conclusion, we explain how these problems are solved with the introduction of *EventIdLog* data structure and how it is used by the participants.

References

[1] Patrick Th. Eugster, Rachid Guerraoui, Sidath B. Handurukande, Petr Kouznetsov, and Anne-Marie Kermarrec. Lightweight probabilistic broadcast. *ACM Transactions on Computer Systems*, pages 341–374, 2003.

[2] Michael North, Nicholson Collier, J. Ozik, Eric Tatara, C. Macal, M. Bragen, and Pamela Sydelko. Complex adaptive systems modeling with repast simphony. *Complex Adaptive Systems Modeling*, 1, 10 2013.

[3] Thomas Williams, Colin Kelley, and many others. Gnuplot 4.6: an interactive plotting program http://gnuplot.sourceforge.net/ (accessed: 21.11.2019).