Distributed systems 2: Project 1 *(lpbcast)*

Enrico Soprana

1 Introduction

This report talks about a Java implementation of the *lpbcast (Lightway Probabilistic Broadcast)* using the simulator Repast Simphony.

2 Protocol

lpbcast is a probabilistic protocol which is completely decentralized, based on local information and uses fixed-size buffers to improve scalability by maintaining in memory only a limited subset of messages.
This is done to create a compromise between scalability and reliability guarantees.
The paper proposes 2 versions of the algorithm: an unoptimized one and an optimized one. The difference in reliability, redundancy, delivery ratio ,and others are provided for the two different versions and different parameters.

3 Assumptions

Latency and clock drift The simulation implemented assumes that every node has the same latency to every other node which is distributed like a gaussian distribution given the mean and variance provided by the user. To maintain the constraint of consequentiality if the value of the distribution is lower than 0 the value is set to 0 (so we never have a negative value). While this means the distribution is not a normal distribution the probability of a negative value (with the used values) is low enough that we can consider it the same as a normal distribution.
The same thing with the same considerations was done for the drift of the scheduling of rounds. While the drift for the gossip scheduling is implemented it is not implemented the drift of the local clock as it's assumed that the drift never gets big enough to be significant (because of a drift which is too small or another protocol that synchronizes the nodes' clock periodically).

Gossip synchronicity Differently from the paper, the gossips are not synchronized. This meant that a small change was needed to take into account this different assumption. In this change, another buffer (*localEvents*) was added to store the newly cast events generated from the application. This was needed as any event generated between the last gossip and the reception of a gossip could be potentially be deleted to satisfy the space constraint for *events*.

Nodes Another assumption made in the model is that no node is ever created or started (initialization excluded). Each `Node` is identified using a `UUID`.

Message ids Like in the paper the message id is composed by the *source*, identified by a `UUID` and an incremental *id* (local to that node) represented using a `long`.
As the id is incremental we can use this to maintain a cumulative value of the received messages for the source considered.

4 Architecture

4.1 Messages

In this model each `Message<T>` is defined as such:

- Source (always known)
- Destination
- Data contained, which can be of any type

Depending on the type contained messages can be split into two groups:

- "Scheduling" messages, which have as source and destination the same node and are used to represent local events like an expired timer or a local event. These are:
  - `RoundStart` which is just a marker used to schedule the beginning of the new round for that specific node (gossips are not synchronized)
  - `ExternalData` which is used to signal a local event which is in the field *data*
- Normal messages
  - `Gossip` (with the fields as defined in the paper)

- `RetrieveMessage`, which represents the request of an event with as id *eventIdRequested*
- `Event`, which represents the successful response to a `RetrieveMessage` (the node found the requested event and responded with the associated event)

4.2 Initialization

The simulation is initialized immediately with the number of nodes indicated (NODE_COUNT).
Nodes are never created and are gradually eliminated (with a rate depending on the simulation parameters).

4.3 Orchestration

In this model, each round corresponds to a single action of a single agent.

The Oracle class is used to orchestrate the actions and message passing of the agents. For each step the oracle handles a single message from the *message* buffer.
To process works as such:

1. Get and remove the first `Message` from the *message* buffer
2. Update the current time using the associated timestamp with the `Message`
3. Find the source and destination of the message (not necessarily different)
4. Check if the destination is alive, if not set the message as to not deliver
5. If the message is a normal message, with probability DROPPED_RATE set the message as to not deliver
6. If the message is still to deliver call the message handler of the node
7. Update the view
8. [oracle tasks]

Info: The oracle never communicates with the `Node` itself but with a wrapper class called `NodeStat`. This is done so that `NodeStat` can register what is happening and write it on multiple .csv later analyzed using python.
This was done in order to separate as much as possible the logic of each `Node`/process from the logic used for the analysis and the actual implementation of *lpbcast*

The oracle task are organized like this:

1. Every GOSSIP_INTERVAL
   (a) Every 500 events write statistics and clean `NodeStats`' buffers
   (b) Kill every nodes with an uniform rate for which, at EXPECTED_STABLE_TIME, DEATH_RATE nodes are dead
   (c) Generate a list of `ExternalEvent` with as size a normal distribution with mean EVENTS_RATE and variance EVENTS_VAR_RATE. Select the node randomly and the time uniformly in [*currentTime*, *currentTime*+GOSSIP_INTERVAL] [1]
2. If STOP_TIME $\neq 0 \wedge$ *currentTimestamp* $\geq$ STOP_TIME stop the simulation

5 Implementation decisions

Local events As previously discussed, due to different assumption in our case if an event is generated between sending a gossip and the handling of another one the event could be deleted.
This was solved using this method:

Function `LpbCast(`e`):`
  $localEvents \leftarrow localEvents \cup \{e\}$
EndFunction

Function `EmitGossip():`
  ...
  forall $e \in localEvents$ **do**
    $events \leftarrow events \cup \{e\}$
  end
  $localEvents \leftarrow \emptyset$

  forall $j \in [1..F]$ **do**
    SEND($target_j$,gossip)
  end
  ...
EndFunction

Removal policies As some decisions in removal policy weren't well defined in the paper I developed multiple choices to understand if those policy decisions could change the results. There are two of these choices for which this was done:

- The removal of old events
- The removal of old eventIds

In both cases it was required to remove e' if $\exists e : e.source == e'source \wedge e.id - e'.id >$ LONG_AGO and it wasn't specified the order in which those should be removed when multiple

[1] As gossips rounds are not synchronized we use the duration of the inteval between two gossips

removable pairs are present.

Missing eventIds In the paper the selection of the events to retrieve works like so:

```
forall e.id ∈ gossip.eventIds do
    if e.id ∉ eventIds then
        element.e.id ← e.id;
        element.round ← currentRound;
        element.gossip − sender ←
          gossip.sender;
        retrieveBuf ←
          retrieveBuf ∪ element;
    end
end
```

As eventIds are cumulative, the implemented algorithm tries also to obtain any `EventId` between the cumulative `EventId` received and the one stored by us. As a corresponding `EventId` with the same source could be missing in our *eventIds* it was decided that in that case no further action should be taken as it could have been removed from our *eventIds* solely for size reasons and not because we didn't receive the associated `Events` from that `Node`.

Deaths All the deaths of the nodes happen close to a round "border". This is done to simplify the association of a newly created series of events with their associated node.
Once a node dies, from the next scheduling on the EVENTS_RATE remains the same but are distributed between less nodes (only the alive nodes). This is made as an assumption because events could be created by a client of the service which, once it becomes impossible to contact a node, would just change the node used to access the service.

6 Configuration

6.1 Simulation options

OUTPUT_FOLDER
The prefix of the folder in which the .csv used for analysis is saved

STOP_TIME
After how many seconds the simulation is automatically stopped (0 for never)

NODE_COUNT
Number of nodes created at the start

INITIAL_VIEW_PERC
Percentage of the *view* buffer that should be pre-filled at start (this number is rounded to the biggest integer number)

MEAN_LATENCY
Mean of the gaussian distribution of the latency

VAR_LATENCY
Variance of the gaussian distribution of the latency

EVENTS_RATE
Mean of the gaussian distribution of the events per GOSSIP_INTERVAL

EVENTS_VAR_RATE
Mean of the gaussian distribution of the events per GOSSIP_INTERVAL

DROPPED_RATE
Percentage of dropped messages

DEATH_RATE
Percentage of dropped messages

EXPECTED_STABLE_TIME
Time at which is DEATH_RATE of the nodes are expected to be dead

DRIFT_PER_SECOND
Variance of the gaussian distribution for scheduling the new round

6.2 Node options

GOSSIP_INTERVAL
Every how many seconds a round starts

EVENTS_SIZE
Size of the *event* buffer

EVENT_IDS_SIZE
Size of the *eventIds* buffer

SUBS_SIZE
Size of the *subs* buffer

UN_SUBS_SIZE
Size of the *unSubs* buffer

FANOUT_SIZE
Size of the fanout (number of nodes to which send gossips, or ask events in the modified version of retrieve)

REQUEST_TIMEOUT_ROUNDS
How many rounds to wait before trying again to retrieve an event

OLD_TIME_RETRIEVE
How many round before trying for the first time to retrieve an event

SUBS_OPTIMIZATION
Enable frequency subs optimization

K
Multiplier for the average frequency to select which subs to delete (called like in the paper)

EVENTS_OPTIMIZATION_FIRST
The policy used for the age-based optimization

EVENTS_OPTIMIZATION_SECOND
The method used if the first didn't remove enough elements (random - non optimized, age - optimized, timestamp)

EVENT_IDS_OPTIMIZATION_FIRST
The policy used for the age-based optimization

EVENT_IDS_OPTIMIZATION_SECOND
The method used if the first didn't remove enough elements (random - just testing, timestamp - both optimized and unoptimized)

LONG_AGO
Number of events after which an older event from the same source is deleted

RETRIEVE_METHOD
Method used to retrieve a lost element

7 Visualization

In this section, the visualization of the simulation is discussed. As an example, fig. 1 is provided.
As no particular relationship between nodes exists, the position of each node (shown as a dot) is randomly chosen. Each node can be in two states:

- Dead
- Alive

Messages are shown as an arrow that points from the source of the message to the destination. Colors were used to identify the type of the message and if the message was received or lost during the transmission.
The legend works as such:

- Gossip received
- Gossip lost
- Ask retrieval received
- Ask retrieval lost
- Single `Event` received (response to retrieval)
- Single `Event` lost

Together with the message, the view of the node is shown as arrows with color . The visualization of the view is created after the handling of the message so it represents the state of the node at the end of the repast round. If SUBS_OPTIMIZATION is enabled the width of the arrow is used to show the frequency[2].

When an event is created only the view of the interested node is shown.

[2]You can click on the tip of the arrow to see the weight of the weight of the arrow. The weight is calculated as such $weight = frequency + 1$

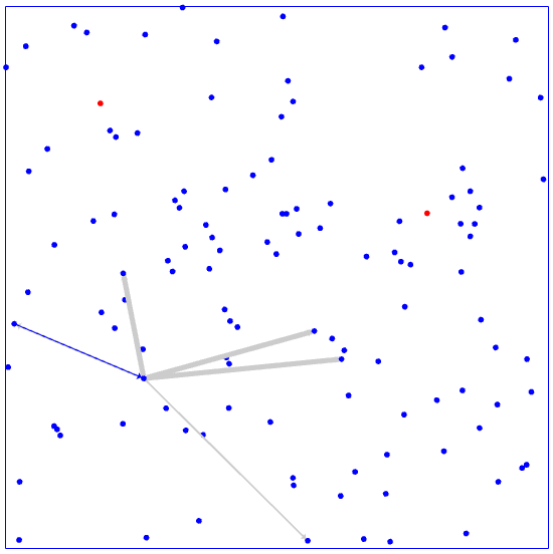


Figure 1: Visualization of the simulation

8 Analysis method

The simulation was analyzed using python (with jupyter, pandas, numpy and matplotlib). The jupyter notebook, used is provided in the project folder.
In all the cases analyzed, all the events created in the last 15 rounds were ignored. This was done to make sure that lack of knowledge of a node regarding an `Event` wasn't due the time of propagation of the `Event` in the system but due to a behavior of the algorithm itself.
When appropriate, the same method was used to ignore the reception of an `Event`, or lack thereof, for all the nodes which died within 15 rounds of the creation of the `Event`.

8.1 How the data is organized

The data is saved in a folder with the path OUTPUT_FOLDER + timestamp. Inside this folder the data is organized as such:

- **dead.csv** A table with columns: id (of the node), timestamp. The id is in UUID format which the timestamp is the number of seconds passed between the beginning of the simulation and the death of the node. If a node doesn't die during the simulation the timestamp is set to -1.0.

(In these folders there is a file for each node node which contains everything that happend to that node)

- **createEvent** In this folder each file stores the list of `Events` which are created in the inter-

ested node as $(id, timestamp, round)$[3].

- **eventHandled** In this folder each file stores the list of `Events` which are handled (delivered to the application layer) as $(id, timestamp, round)$.
- **eventReceived** In this folder each file stores the list of `Events` received by the node (not necessarily delivered to the application layer) as $(id, timestamp, round)$.
- **retrieveFail** In this folder each file stores the `EventId`, timestamp, and round of a failed run (couldn't recover the `Event`) of the retrieve procedure. The columns are: $(source, id, timestamp, round)$.
- **retrieveFound** In this folder each file stores the `EventId`, timestamp, and round of a successful run of the retrieve procedure. The columns are: $(source, id, timestamp, round)$.
- **retrieveTries** In this folder each file stores the `EventId`, timestamp, and round of an insertion of an element in *retrieveBuf* with columns $(source, id, timestamp, round)$. This is different from the union retrieveFail and retrieveFound as they do not take into consideration whether an element is obtained through a gossip and not the retrieve procedure. When this happens the corresponding element in *retrieveBuf* is deleted, and the retrieve procedure canceled (so it doesn't fail or succeed).

9 Analysis

In all the analysis some common parameters are used (otherwise stated):

NODE_COUNT	125
INITIAL_VIEW_PERC	0.5
MEAN_LATENCY	0.013s
VAR_LATENCY	0.003s
DRIFT_PER_SECOND	0.00001s
GOSSIP_INTERVAL	30
SUBS_SIZE	10
VIEWS_SIZE	15
FANOUT_SIZE	4
REQUEST_TIMEOUT_ROUNDS	1
OLD_TIME_RETRIEVE	1
SUBS_OPTIMIZATION	False
K	1
EVENTS_OPTIMIZATION_FIRST	None
EVENTS_OPTIMIZATION_SECOND	Random
EVENT_IDS_OPTIMIZATION_FIRST	None
EVENT_IDS_OPTIMIZATION_SECOND	Timestamp
LONG_AGO	3
RETRIEVE_METHOD	Paper

[3]When a round is mentioned the local round is used. This means that *rounds* from different nodes could refer to different time ranges

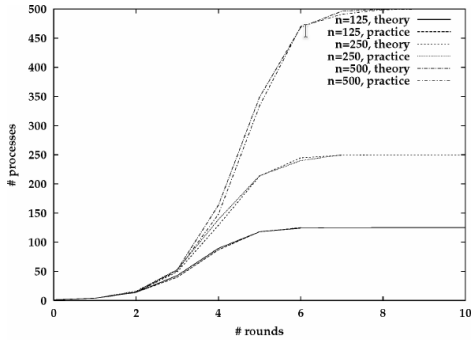
9.1 Infection time (unoptimized)

The figure in this section are created from the data about the dissimination of all events created before $210s$ (~ 7 rounds).

Infection time given the number of processes

INITIAL_VIEW_PERC	1.00
DROPPED_RATE	0
DEATH_RATE	0
EVENTS_RATE	10, 40
EVENTS_VAR_RATE	0
EVENTS_SIZE	60
EVENT_IDS_SIZE	120
VIEWS_SIZE	3
FANOUT_SIZE	3

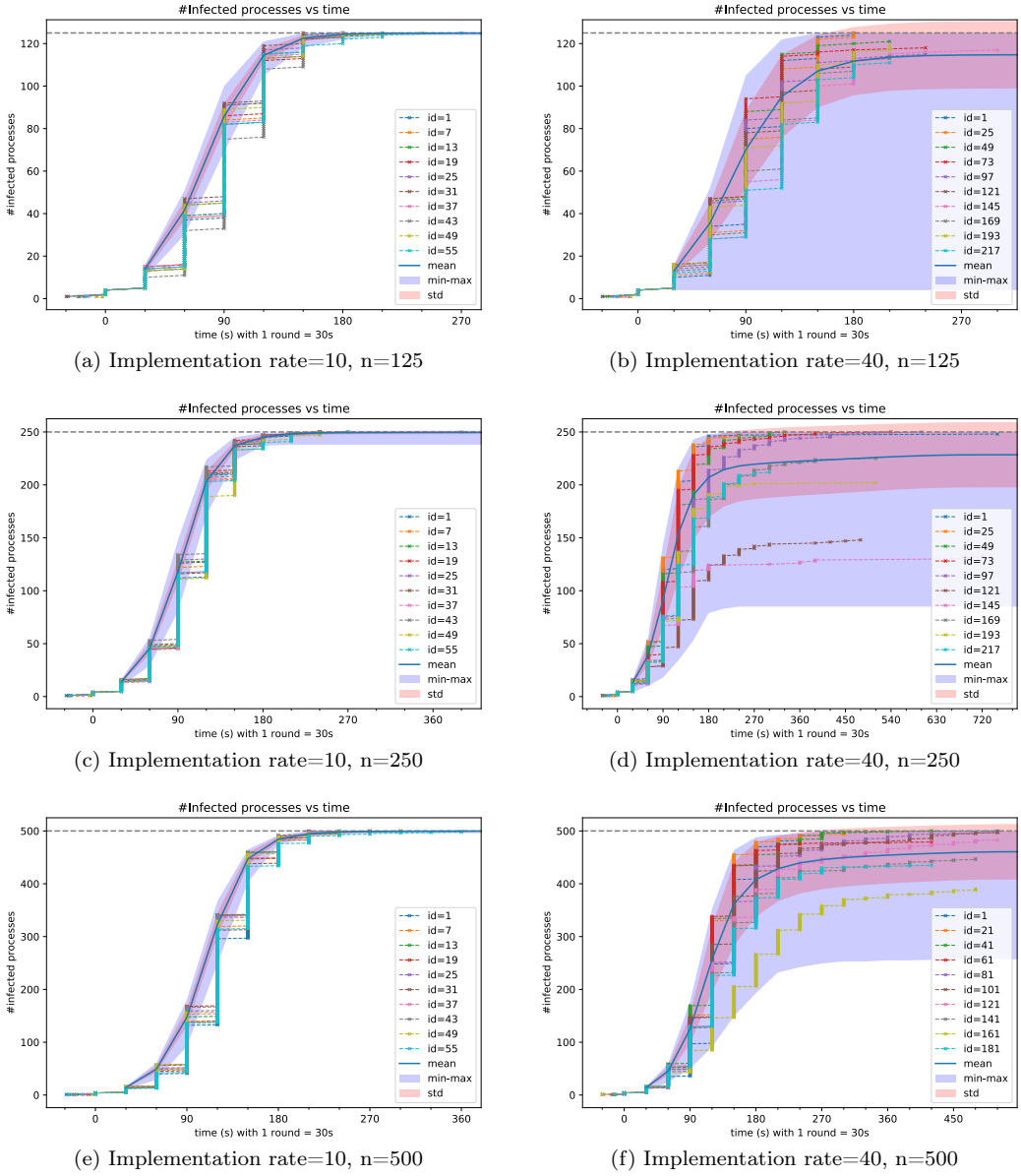
In this test (fig. 3) we want to see how many rounds are necessary to completely disseminate the message in the network.
As an `Event` can happen at any moment (and at any distance from the next gossip), to make the figure clearer the lines were aligned at their first gossip to better show how the behavior of the system can change with time.
As this change depends on the load of the system different configurations of the simulation are proposed for the implementation.



(a) Paper

Figure 2: Infected process vs time (paper)

The figures were organized differently from the paper (fig. 2). A figure for each case was used to show more information. 10 equally-distant (in id) events are selected and their dissemination shown. The mean (blue curve) and the standard deviation (red area) are shown together with the worst and best message (blue area).
We consider the first `Event` cast as unaffected from the other `Events` as $\frac{propagationTime}{\text{GOSSIP\_INTERVAL}} < |events|\text{EVENTS\_RATE}$. This is because the buffer *events* will not be full until $\frac{\text{GOSSIP\_INTERVAL}|events|}{\text{EVENTS\_RATE}}$ so before that, no interference between the dissemination of



(a) Implementation rate=10, n=125

(b) Implementation rate=40, n=125

(c) Implementation rate=10, n=250

(d) Implementation rate=40, n=250

(e) Implementation rate=10, n=500

(f) Implementation rate=40, n=500

Figure 3: Infected processes vs time

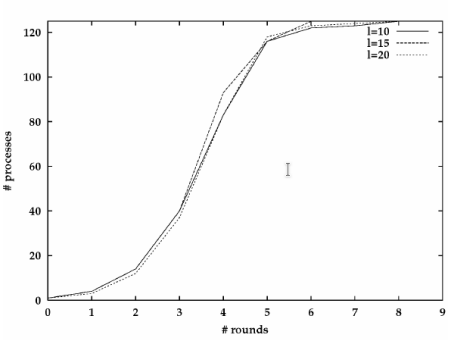
different `Events` can happen. For this reason, we can compare the first message with the message disseminated in the paper. The obtained results are comparable with the ones in the paper.
We can see from the figure that with the increase of EVENTS_RATE the number of infected nodes (at a specific time) decreases, the standard deviation widens and more importantly the worst-case message is received by fewer nodes.
This happens because when the buffer *events* becomes full, `Events` are randomly removed, possibly stifling the dissemination of new events. This is especially noticeable with a high number of nodes and a high number of events.

Infection time given the size of view In this test (fig. 4) we want to see what influence the view size has on how many round are needed to disseminate the whole network.

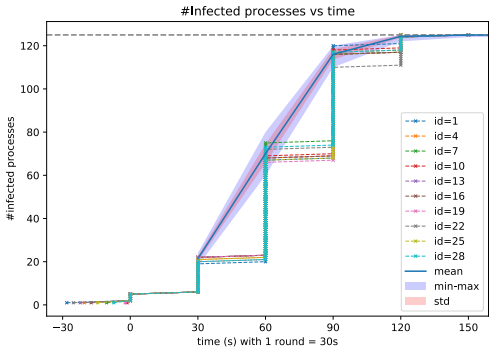
As we can see from the figure the number of infected processes for each round are comparable with the ones of the paper and that (like in the paper) the view size doesn't contribute significantly to the result.

9.2 Reliability vs view size (optimized vs unoptimized)

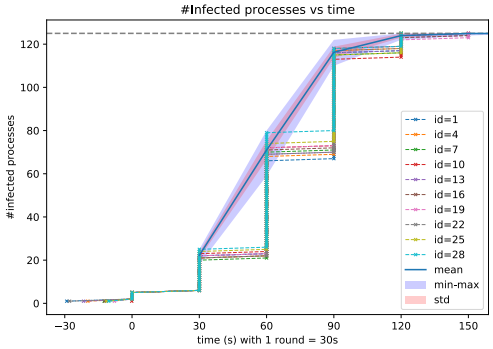
In fig. 5 we measure the number of processes that, on average, receive a message. We can see that, differently from the paper, the curve is flatter and



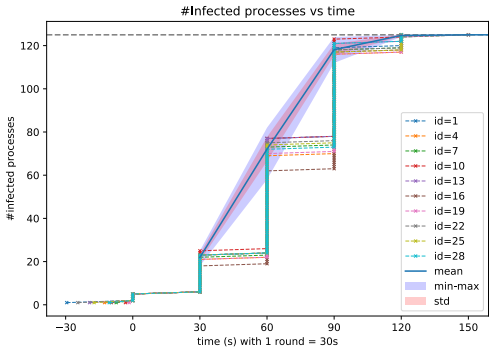
(a) Paper



(b) Implementation rate=10, view=10

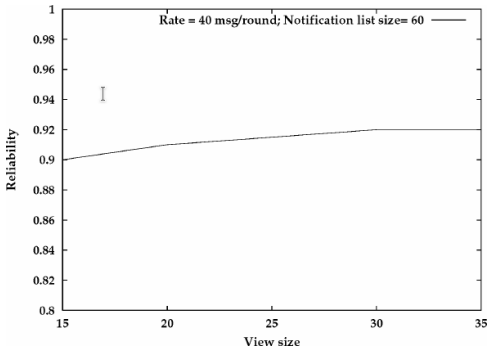


(c) Implementation rate=10, view=15

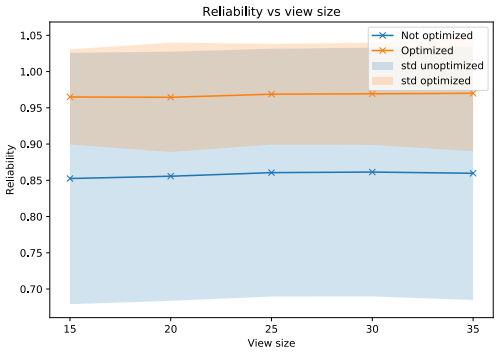


(d) Implementation rate=10, view=20

Figure 4: Infected processes vs time (given the fanout 3 with 125 processes)



(a) Paper



(b) Implementation with rate 40

Figure 5: Reliability vs view size

the reliability is slightly lower. This could be due to different assumptions.

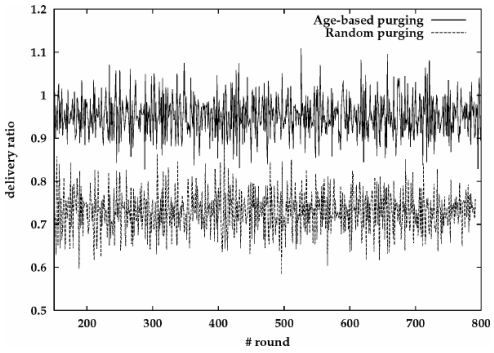
I also decided to compare how the reliability changes with the optimized version. As shown in the image the optimized version leads not only to better reliability but also a less spread one (the reliability of each message is closer to the system-wide one).

9.3 Delivery ratio and redundancy (unoptimized vs optimized)

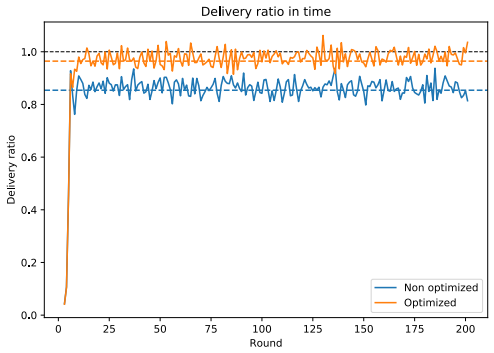
NODE_COUNT	60
EVENTS_RATE	30
DROPPED_RATE	0
DEATH_RATE	0
EVENTS_SIZE	30
EVENT_IDS_SIZE	120
SUBS_SIZE	10
UN_SUBS_SIZE	5
VIEWS_SIZE	15
FANOUT_SIZE	4

Delivery ratio This test (in fig. 6) measures the ratio between the number of delivered events to the application layer and the messages broadcast in each round (on average for each node).

Redundancy level In this test (in fig. 7) we try to understand how many of the received messages

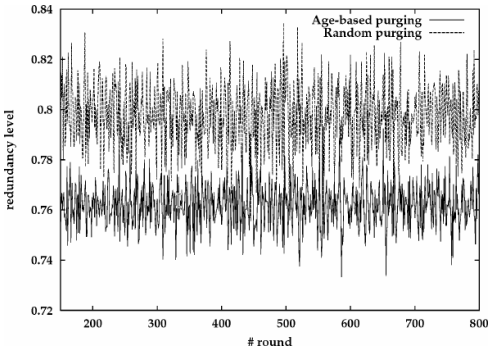


(a) Paper

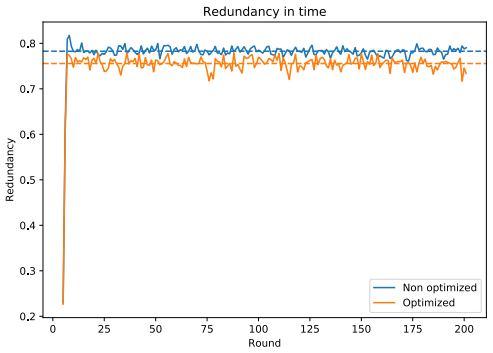


(b) Implementation

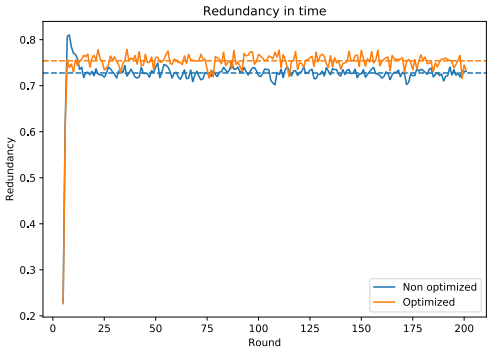
Figure 6: Delivery ratio vs #round



(a) Paper



(b) Implementation with first received



(c) Implementation with delivered

Figure 7: Redundancy vs #round

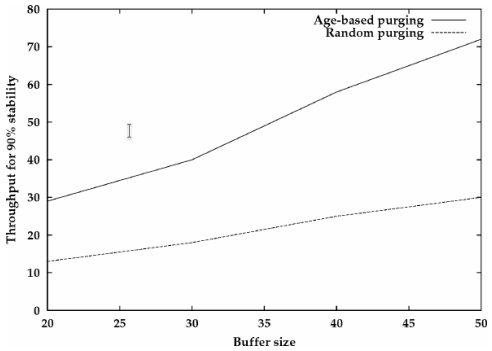
was already received in the past wrt to the total of the messages received.
In this case, we consider the messages received event if they are not handled by the application layer. This means that we consider as redundant messages all the repetitions messages with the same id after the first reception even if the message is not delivered to the application layer.
This can lead to different results as, due to the bounded nature of *eventIds*, an event could be considered delivered to the application layer, and for that reason not delivered "again". This event could be later delivered to the application if all the event ids of that source are deleted. If this happens once the message is received there will be no corresponding event id and for that reason, it will be delivered to the application layer with the result that the first delivery of the event to the application could be something other than the first occurrence of the message. This can be alleviated with a very big *eventIds* ($\gg$ NODE_COUNT but that would undermine the "scalable" property of the algorithm).[4]
To better illustrate the difference both conceptions of "redundant" are shown. We can see that the difference of definition of redundant can cause the opposite result in the run shown.

[4]For the same reason if all the event ids from a source are deleted multiple times an event could be delivered to the application layer multiple times

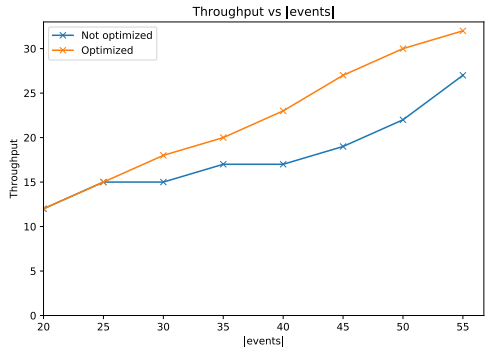
9.4 Throughput

NODE_COUNT	60
DROPPED_RATE	0
DEATH_RATE	0
EVENT_IDS_SIZE	120
SUBS_SIZE	10
UN_SUBS_SIZE	5
VIEWS_SIZE	15
FANOUT_SIZE	4

In the throughput (in fig. 8), while the optimization on the buffer was better than the non optimized version the difference is smaller than in the paper. I also run the test with both optimizations (*events* and *subs* buffer) and noticed that they are in conflict to the point that the behaviour is the same of the non optimized version.

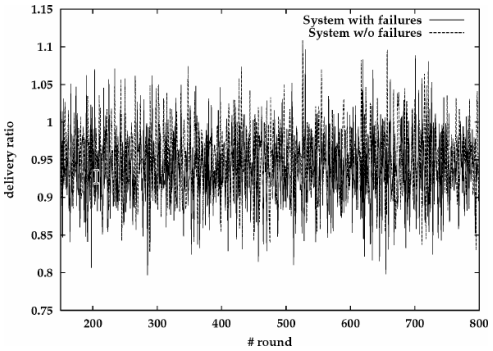


(a) Paper

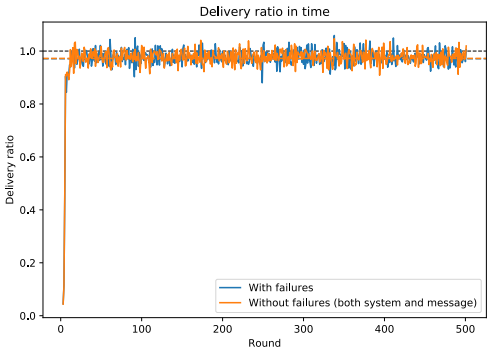


(b) Implementation

Figure 8: Throughput



(a) Paper



(b) Implementation

Figure 9: Delivery ratio vs #round (with and without failures)

9.5 Delivery ratio and redundancy (optimized, with and without failures)

NODE_COUNT	60
EVENTS_RATE	30
DROPPED_RATE	0
DEATH_RATE	0
EVENTS_SIZE	30
EVENT_IDS_SIZE	120
SUBS_SIZE	10
UN_SUBS_SIZE	5
VIEWS_SIZE	15
FANOUT_SIZE	4

Delivery ratio Like in the paper we can see that the delivery ratio doesn't change significantly by adding failures.

Info: Both dead nodes and lost packets are considered failures

Redundancy I decided to add this figure (fig. 10) to see if adding more events in the buffer during the retrieval process (as we lose packets and node die we need to retrieve more events) made any difference for the redundancy measure.
This didn't make any significant change in the measure.

Info: Like in the previous figure, both dead nodes and lost packets are considered failures

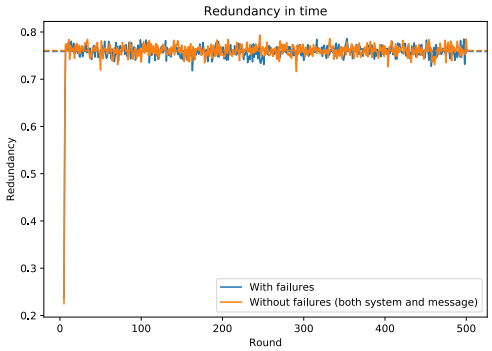


Figure 10: Redundancy vs #round (with and without failures)

9.6 Different method to retrieve lost `Events`

NODE_COUNT	60
DROPPED_RATE	0
DEATH_RATE	0
EVENT_IDS_SIZE	120
SUBS_SIZE	10
UN_SUBS_SIZE	5
VIEWS_SIZE	15
FANOUT_SIZE	4

During the development, I had some performance difficulties with the retrieve procedure. I tried to create an alternative one and tested the throughput on the new method.
This method works like this:

forall $e.id \in gossip.eventIds$ **do**
  ...;
  chose $F-1$ *nodes from view*;
  forall *i=1..F-1* $\in$ *target* **do**
    ask e.id to $target_i$
  end
  if $\neg$ *receive* **then**
    ask e.id to e.id.source;
  end
end

The use of this new method gave this results:

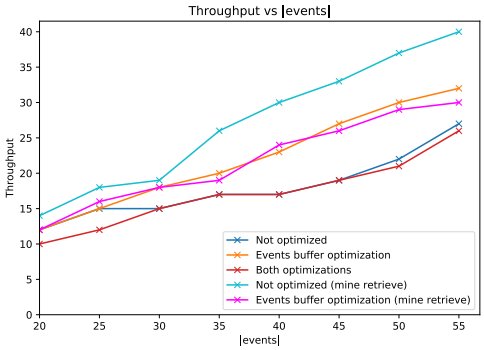


Figure 11: Throughput vs |events|

The method gives better results, especially in the unoptimized case. The general improved could be attributed to the greater "aggressiveness" of the algorithm (asking to more node in a parallel fashion and asking to the source before then the paper implementation).
While I wasn't able to prove it, I suspect that the reason for which the unoptimized version gave a better result is due to a more random content inside the *events* buffer.

9.7 Difference between the different methods to purge elements from *events*

During the implementation, I decided to test for different policies regarding the delation of too old elements (see removal policies in sec. 5). I made different tests but I wasn't able to see any difference in any of them. I suspect that this is due to how often such decisions can happen, at least with the parameters that I used.