Lightweight Probabilistic Broadcast

Zanetti Marco

Pirocca Simone

The LPBroadcast algorithm is a gossip based protocol for broadcasting, it is an improvement of traditional gossip based broadcast algorithms, its focus is to improve the scalability which in traditional algorithms it's a weak point.

The implementation and simulation of the Lightweight Probabilistic Broadcast algorithm has been done using the Repast Symphony framework.

Repast Symphony is an agent-based modelling and simulation platform for workstation and small cluster, it is mainly used for biological modelling and simulation but its application is possible also on distributed systems protocols.

1. MODEL IMPLEMENTATION

The algorithm has been implemented with the use of some Java classes, two of them are the most important Process and SuperAgent.

The Process class represents the structure of a process of the algorithm, one instance of this class is created for each process of the simulation.

The SuperAgent class has been created mainly for the simulation phase, in fact it opearates as an "over the other" agent which is used to introduce chaotic events, in order to simulate the failure of some nodes in the network and understand how it works in stress conditions.

Some utility classes are present in the source code of the model and mainly provide common operation used to manipulate messages or events.

Other important classes are the Event one and the Gossip one, which is the software abstraction of a message send by a process to one other. Given that this two classes does not provide methods or operation but are a sort of containers their description is trivial, we will instead focus on the description of Process and SuperAgent classes and on the explanation of the BroadcastContextBuilder that is the object which takes in charge the duty of instantiating the simulation.

2. SUPERAGENT.java

The SuperAgent is the object delegated to manage the "chaotic" behaviour of a network, simulating some unpredictable failure due to nodes crash. When a node crashes it is not able to process events and send messages, meaning that the robustness of the whole newtork is reduced.

As an supernatural agent in the model the SuperAgent knows any process in the simulation either it is active or crashed, in the source code this condition is implemented with the use of a collection of all processes instantiated. In a real case scenario the whole class won't have any utility but here it is absolutely needed to run the simulation.

The SuperAgent takes some configuration parameters from the ContextBuilder object and complete the setup of processes, giving them an identifier which is just a string builded with the prefix "proc" and a progressive

number. Always because of the simulation needs the SuperAgent also assign the starting view to each process picking a random subset of element from the collection of processes.

The only other thing done by the SuperAgent is the “ChaosThread”, a secondary thread which at specific time interval pick a random number of processes and change their “crashed” field, chainging it from FALSE to TRUE and vice versa to simulate a random failure in a process and it possible recover after some times. Technically this process is started by the ContextBuilder once the environment has been instantiated.

3. PROCESS.java

The Process class is the core of the whole algorithm implementation because here gossips and events are managed also with subscripion and unsubscription from the network.

Many private methods in this class are used to perform state changing ad to manage sub/unsub procedures, others create gossips to be send to other nodes in the network.

It is in this class that the “chaotic” influence of the SuperAgent through a simple control on the CRASHED boolean variable which, if TRUE, prevents the execution of normal function such as gossips send, events retrieving and broadcasting, in this way we have been able to simulate at any time an error in the network without adopting a precise network topology and thus generalizing the simulation as much as possible.

The behaviour of the Process is dependent on Repast’s rounds using the @ScheduledMethod attribute on the “main” method **checkClock()** which checks if we are at the start of a new round and runs some functions, otherwise it creates and sends a gossip on the network. All private methods and functions are called from this scheduled method; at the start of a new round events are retrieved and old ones are removed from the event list.

Some methods such as AddView are just used at startup, in order to initialize the system in a consistent state and give it means to enter and operate in the network, they are usually invoked by the SuperAgent.

4. BROADCASTCONTEXTBUILDER.java

The BroadcastContextBuilder is a class needed by the Repast environment in order to initialize the system and run the simulation for the portion which cannot be set from configuration wizards available once the model is launched.

This builder inherits from the Repast ContextBuilder and in its body firstly retrieve parameters which have to be set in the simulation menu and before the initialization phase which is launched with the reserved button. The list of this parameters can also be set in the parameters.xml file inside project’s source tree but is is way easier to set it in the simulation window; like in many other java project parameters usually need an explicit cast to their primitive type.

With these parameters two utility objects are instantiated, this is mainly for a sake of clarity and source code readability, in fact these objects are used as a common configuration for every process and to pass to the SuperAgent parameters it needs.

At the end of the **bild()** method of the ContextBuilder there is the creation of SuperAgent and of all processes; processes are immediately added to the context of the simulation and then the list passed to the SuperAgent which concludes their setup as we have seen above.

Once also the SuperAgent is added to the context it is possible to start its inner thread and then finish the initialization of the simulation.

Analysis

The purpose of this section is to **replicate** some analytical or experimental **evaluations** of the paper regarding this protocol, after implemented it using Repast framework. To create the ideal scenario for a particular analysis, sometimes we had to adapt the code itself. To obtain our results, we used ad-hoc methods called by particular datasets, used to create graphs and files containing the outputs of the simulation run in certain conditions. All of this is offered directly by Repast.

We decided to focus on the two most important factors of the suggested protocol: the **propagation** of an **event** (clustered by fanout and number of processes) and the **propagation** of a new **subscription**. Our intention was to analyse also the age of events and throughput; unfortunately, errors occurred when an integer method was called by a dataset. Thus, we couldn't dig into also there two results.

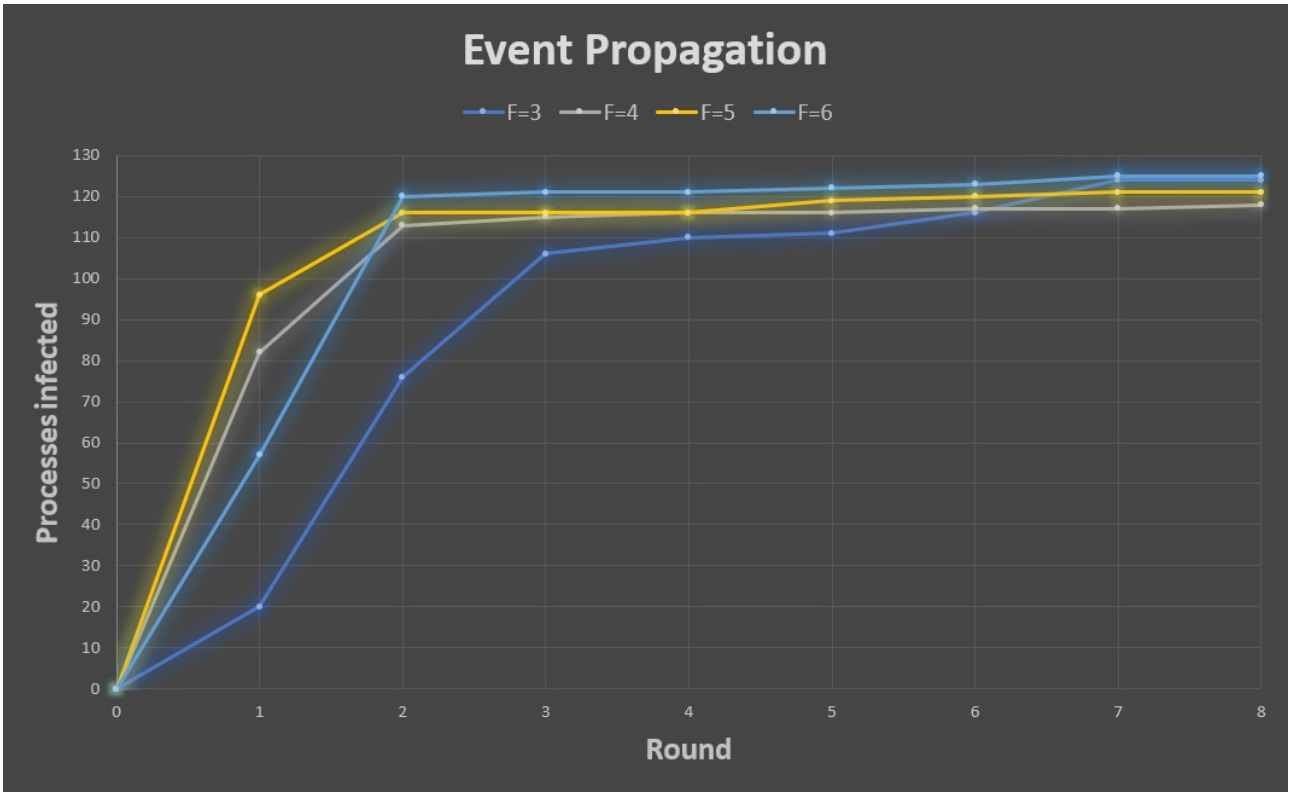
5. EVENT PROPAGATION BY 'F'

This analysis aims to demonstrate the theoretical graph (Fig. 2) of the original paper.

To set up this scenario, we chose a specific event id (that was "**proc1-1**" as you can see from the code). First of all, every process has a method called `hasRandomEventId()` that the simulator calls every round, to know how many of them has already received this particular event. Then, these information are aggregate within the dataset "**Event Propagation**" and displayed on the graphs, as well as stored in a file.

The basic parameters don't change: **125** total **processes**, **view size** of **50** and **40** new **events** generated **every round**. Instead, the parameter "**F**" that represent the fanout changes, since we are interested to see the difference of event propagation with different number of processes that a single process can infect. Once collected all the information about different values of **F**, **from 3 to 6**, we created a graph containing the results. We define the **goal achievement** where the event is able to **infect 95% of the processes** (that is 120). This because we discovered that in our simulations it takes few rounds to infect about 120 processes, while many rounds pass before reaching the last ones.

You can observe all the raw outputs in the "*Analysis/Event Propagation (by F)/*" directory.



As you can see, with **F=3** on average it takes **6** rounds to achieve the 95%, as demonstrated in the paper. Instead, with **F=4** the total rounds decrease to **4**. Going forward, we notice that with **F=5** has more or less the same behaviour of the previous one. Finally, we're not surprised that **F=6** achieved the goal in only **2** rounds.

6. EVENT PROPAGATION BY 'n'

This paragraph, instead, plays the role of emulate the simulation results of the paper (Fig. 5.a).

The setup of the environment, as well as datasets and graphs used, are the same of the above one: so, we still need "**proc1-1**" as random event to check. Also, the parameters of the simulation are the same, except for the fanout F, that is set to 3 by default. Then, this time, the number of processes **n** changes, that is set to **125**, **250** and **500** (as done in the paper). The goal achievement again comes with 95% of processes infected.

You can observe all the raw outputs in the "*Analysis/Event Propagation (by n)/*" directory.

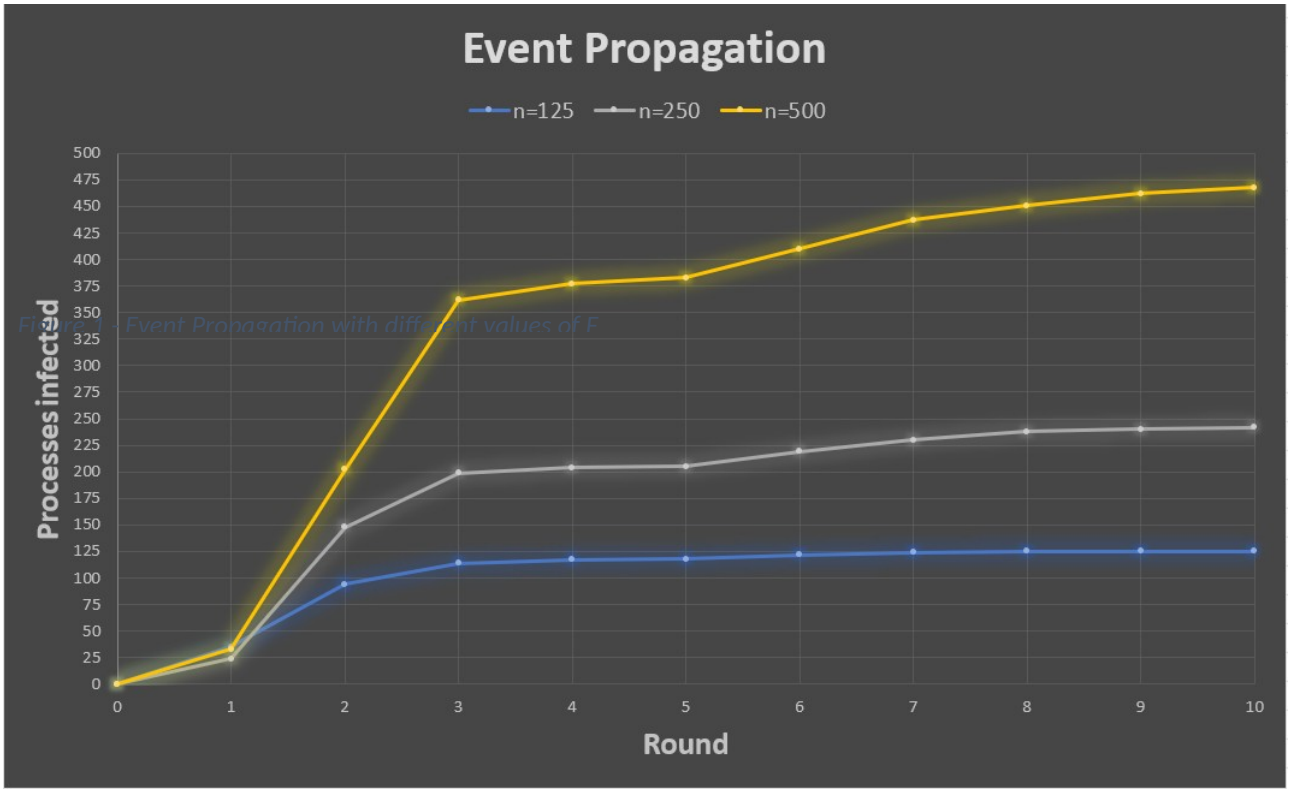


Figure 2 - Event Propagation with different values of n

This graph described well that for **n=125** (more or less as for the first case of the previous graph), the goal is reached after **5** rounds, and the number of rounds increase linearly with the number of processes. In fact, with **n=250** the 95% processes are infected after **8** rounds, while with **n=500** we have to wait **10** rounds.

These results are quite different from the paper one, but still the behaviour is the same, remembering that our simulation is not done in a physical environment but with software, that cannot emulate too much networks and process delays.

7. SUB PROPAGATION

The last evaluation wants to demonstrate, as in section 7 of the paper (Fig. 15), that the optimized purging of subscriptions works better than a simple random purging.

The setup for this scenario was not so simple. First of all, since a **frequency** integer for every subscription received was **required**, we added to the already existing Hash table another one, containing the **id** of the **sub** as **key** and an integer (**frequency**) as **value**. Then, each time a process receives a sub

during the event reception phase, it increases the value of the respected process within the new Hash table called `subFrequencies`. This way will be easier to check the frequency of a particular element, as well as the average calculation during the algorithm. Finally, when we need to purge the list, a new method called `purgeOldSubs()` is now used, that implements exactly the steps described in the paper, also calling the `selectProcess()` function. You can find the details about the implementation in *Process.java* and *Util.java* classes. This is all done for the optimization of subscription purging only.

For this purpose another dataset was created, "**Random Process in Subs**", that calls the method `hasRandomProcessInSubs()` of every process in order to know how many of them knows the chosen one.

The most difficult part, then, was to simulate an adequate subscription for both scenarios, since for what we decided as **default** all processes were already **subscribed**. So, we decided to take a chosen process, that was "**proc1**" as you can see from the code, and change the behaviour of that process, as well as actions made by other processes with respect to it.

For example, during the initialization of the experiment (when the super-agent add a random view to each process), if a process see the **chosen id** within the passed list, if will **discard** it; so it will not be inserted neither in the **view** nor in the **subs** lists. This way, when "**proc1**" will subscribe, no processes would know him, as it should be.

Then, let's see the behaviour of the **chosen** process. First, his status initially is **UNSUBSCRIBED**. He waits for about **100 rounds** (so 5000 ticks) doing nothing. This is done to let **other processes unsubscribed** or **subscribed** in the meanwhile, to shuffle the lists. Then, at tick **5000** he sends a subscription message (choosing a random process in view as any other process does) and also the **dataset** starts **saving** data.

For better understand the numbers of the following graph, we have to say than, from tick **5000** to **6000** the number subscription was about **80 processes**, because of the random function for sub/unsub. So, this time we can define the goal achievement when **95%** of 80 processes are infected, so **75**.

You can observe all the raw outputs in the "*Analysis/Sub Propagation)/*" directory.

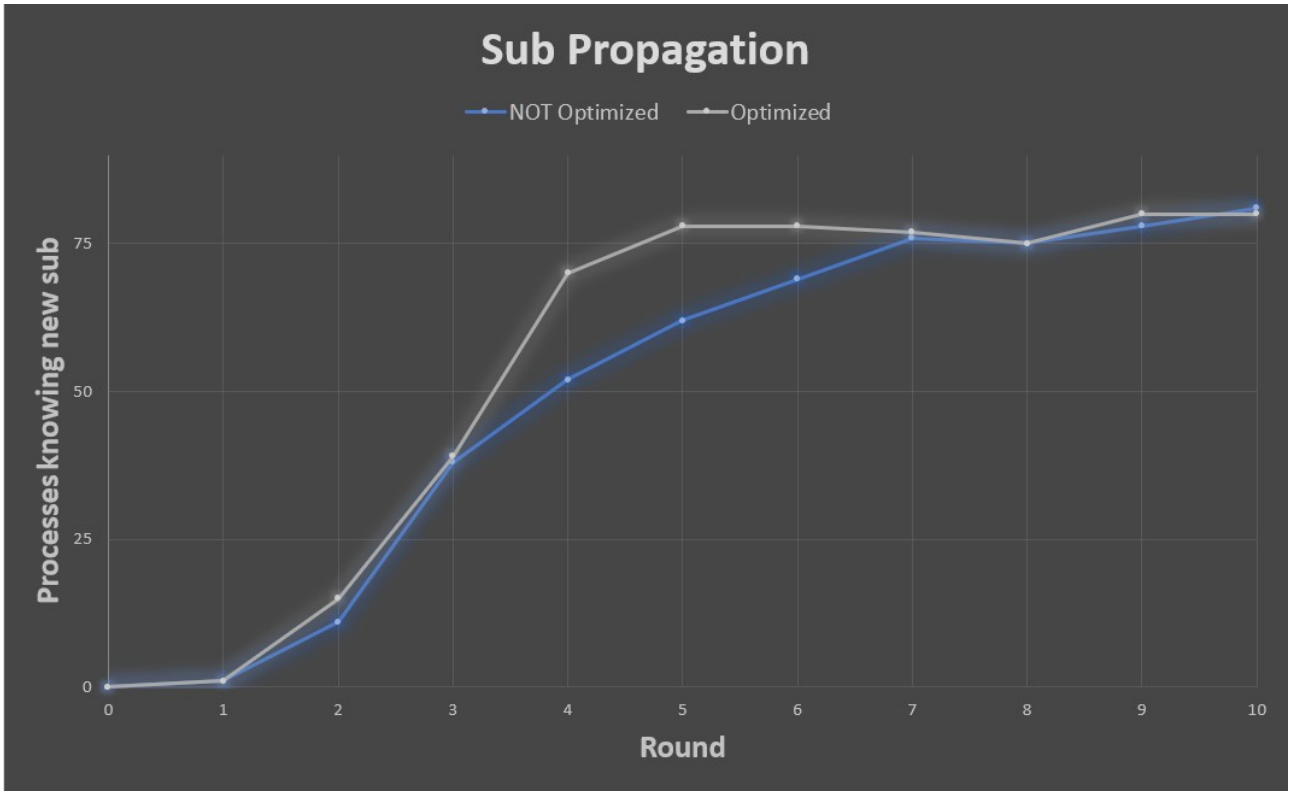


Figure 3 - Sub Propagation with Optimized and Not Optimized purging algorithm

We notice that for a normal **random** purging, in order to infect **75** process out of **80** you have to wait for **7** rounds, while with the **optimized** one that threshold is achieved in only **5** rounds. This demonstrates the efficiency of saving a frequency number for each subscription received, so new subs are keeping for longer time in the list rather than the older ones.