Lightweight Probabilistic Broadcast

Distributed Systems 2 – 1st assignment

Nicolò Pomini, 203319
nicolo.pomini@studenti.unitn.it

November 19, 2019

1 Introduction

This report describes the implementation and the evaluation of the first assignment of the *Distributed Systems 2* course, which is based on a publication written by Eugster Et al. that describes a protocol called *Lightweight Probabilistic Broadcast* [1]. The assignment is developed using Repast Simphony[1], an open source agent-based modeling and simulation platforms for Java and Groovy languages.

The report describes briefly the protocol in Section 2, for then introducing the architectural choices of the implementation in Section 3. Finally, in Section 4 the protocol is analyzed using several metrics and different parameter configurations.

2 The protocol

The protocol described by Eugster, Guerraoui, Handurukande Et al. [1] is part of the probabilistic epidemic protocol family.

An epidemic protocol is a protocol used in distributed systems to spread around data and messages in the system itself. Unlike broadcast protocols, which send systematically every message to every process that is part of the system, epidemic – or gossip – protocols are designed on the way epidemics spread. A process is considered *infected* when it has received the message that is currently spreading around. In turn, such processes forward the message to some other processes they know, with the aim of infecting as many other nodes as possible. These protocols are both robust and scalable, because they usually support node joins and crashes, and they achieve a good convergence rate – that is the number of protocol iterations needed to spread *enough* around a given message in the system, and they achieve good performance even with a large number of nodes partecipating to the protocol.

[1] https://repast.github.io/

The Lightweight Probabilistic Broadcast [1] reflects such protocols, because it is tolerant to changes in the set of involved nodes, and because every member sends continuously around information about *events*, with the aim of let such information spread around. It is also *probabilistic*, because every node manages its memory randomly, possibly cutting out pieces of information at every iteration. Furthermore, each node's knowledge of the network is shared in the messages that are send around, and a node's neighbourhood can vary at every iteration of the protocol, performing a sort of *node sampling*.

The model of the Lightweight Probabilistic Broadcast is made of a set on nodes $\Pi = \{p_1, p_2, ..., p_n\}$, and each of them has a partial knowledge of Π, called *view*: a process $p \in view$ can be contacted by the view's owner to receive a message. The nodes continuously exchange *gossip messages*, made of several sets of data:

- *subs* contains a list of process ID that are part of the system. The sender of the message puts in this set all the nodes it knows that are part of the network. All the processes in *subs* that are not already in the receiver's view are added to the view itself.
- *unsubs* contains a list of process ID that are no longer part of the network. The sender puts here all the nodes it knows that have left the system, and any process of this set that is belonging to the receiver's view is removed by the latter.
- *events*, containing the events to be spread in the network.
- *eventIds*, containing the identifiers of the events known by the sender of the message.

The exchange of messages allows the network to spread around information, such as events, new joining nodes or nodes that are leaving the system. At every iteration of the protocol, each node prepares a gossip messages with the information it knows – its view of the system, the events and the event IDs it knows about – and sends this message to a randomly selected set of known processes. Upon reception of a gossip message, a node updates its local view and its event set, and then it randomly truncates its knowledge if the latter exceeds in size: in fact, every set of information – the view, the known events and event IDs, the subscription and unsubscription lists – has a maximum size value, and in case it is not respected a random truncation is performed.

A node n can join the network just knowing at least one process p in the system: it is enough that n sends a gossip message to p, putting itself into the message as a subscriber member. In the following iterations, p will spread around the subscription of n. In a similar way, a node that wants to leave the system can send a message to another known process, putting itself into the unsubscription set.

It is possible that a node gets notified about a certain *event ID* without receiving anything about the respective event: in this case, a certain number k of rounds is waited, hoping to receive the event in the following iterations of the

protocol. If this it is not the case, after k rounds, the node asks to a random node in its view for information about the event: if the selected node is not able to respond – because it is crashed or because it does not know the event too – the node that generated the event itself is asked.

3 Architecture and assumptions

In this Section the architectural choices of the implementation are presented. First of all, it is assumed that every object inside the system is identified by an unique string: events and processes have an ID field made of a string. We also assume that at every iteration one event happens, that can be either a new message, a new node that joins the network, or a node that leaves the protocol.

Four main entities are defined: an *event*, a *gossip message*, a *process* and an *action manager*. Furthermore, there is a *builder* class with the role of initializing the system.

3.1 Event

The `Event` class represents an *event*, and can be generated by any process at any time. It has several relevant attributes, such as the *eventID*, the ID of the process that created the event, a *payload* containing any possible data about the event itself, a *startRound* indicating the round when the event was created, and a dictionary called *knownBy* that maps every process that has received the event with the round when such reception happened.

Since this type of data will be stored and sent around inside other data structures – such as sets or dictionaries – it is important that the `Event` class is hashable and comparable with other events.

3.2 Gossip Message

A gossip message reflects what is described in Section 2. It is implemented in the `GossipMessage` class, which is composed by the following attributes:

- a set of subscriptions, containing ID of processes, called *subs*;
- a set of unsubscriptions, containing ID of processes, called *unsubs*;
- the ID of the sender process, called *sender*;
- a set of `Event` objects, called *events*;
- a dictionary that maps an event ID with the process ID that is the generator of that specific event, and this attribute is called *eventIds*.

3.3 Process

The `Process` class is the core of the project, and it is were the protocol described by Eugster Et al. [1] is actually implemented. Let us start describing first how the class is structured, for then explaining the behaviour of the processes.

3.3.1 Attributes and parameters

First of all, this class has a set of parameters that must be set before starting the protocol. They specify the maximum number of items the process' view can store (`MAX_VIEW_SIZE`), the subscribed and unsubscribed set maximum capacities (`MAX_SUBS` and `MAX_UNSUBS` respectively), the number of events and eventIds that the two respective sets can contain (`MAX_EVENTS` and `MAX_EVENT_IDS` respectively). Also, the number of random processes to which send a gossip message must be set (`GOSSIP_SIZE`), together with the number k of rounds to wait before asking information about an event in case the eventID of such event is known but the event itself is not (`K_ROUNDS`).

Every process is identified by a *processId*. A process has a *view*, which is the set of nodes known by the process with which is possible to communicate. This attribute is a dictionary that maps a process ID string to an instance of the `Process` class. Also, a reference to all the processes is needed, in case the process has to contact the creator of an event that must be retrieved and such a creator process is not part of the view. This attribute is called *allEvents*, it is a dictionary structured as the view, and it is used in read-only mode – in fact only the *action manager* and the *builder* modify it.

Also, the process class has several storage sets, used to keep information about events and the members of the network: *subs* and *unsubs*, to keep information about the members of the system; *events* to store event objects. Furthermore, two parallel lists are used to track event identifiers and the creator processes of such identifiers: this two attributes are called *eventIds* and *eventCreators* respectively, and they are sorted by arrival time. In fact, every time a gossip message is received, the information about event IDs and event creators are appended to the two lists, while when removing elements in case the maximum capacity of the two lists is reached, those items with lower indexes in the two lists are removed.

Finally, a buffer called *elementBuffer* is used to store event IDs for whom their respective event object is not known by the process: if this is the case, the event ID, the round in which the ID is received and the sender of the gossip message containing the event ID are saved inside the buffer.

Every node contains also a *lock* object, to ensure that all the internal functions of the class are executed in mutual exclusion. This property is important since every main function involved in the protocol accesses and modifies the attributes described above.

For visualization purposes, each node is a dot inside of a 2D *Continuous space*, and its position does not change over time. Also, a node is part of a visual *Network* that shows the message exchange between nodes: an edge in

this network represents a *gossip message* sent between the two connected nodes.

3.3.2 Process lifecycle

The behaviour of a process is the core of the protocol, and it is explained in the original paper [1]. Basically, at every iteration of the execution, each process prepares a gossip message m, putting inside of it all its knowledge about the network members and the events. Then, it selects n random processes belonging to its view, with $n =$ `GOSSIP_SIZE`, to which the gossip message will be sent. When a message is sent, an edge on the visual network is added between the sender and the receiver of the message.

Upon the reception of a gossip message m, the node updates its knowledge of the networks and its event with the content of m, randomly removing items in case the sizes of its storage sets exceed. In case m contains an event ID for whom its respective event object is not contained neither in m nor in the set of events of the process, this ID is saved into the *elementBuffer* attribute, together with the current round and the ID of the sender of m.

Furthermore, in every iteration the *elementBuffer* is scanned. Let b_i be the current element of *elementBuffer*. In case at least `K_ROUNDS` are passed since the reception of b_i and the event related to b_i is not known yet, the process starts asking around for information about the related event. Firstly, a random process in the view is asked. In case the latter is not able to provide the event, the creator of the event itself is asked. Once the event is retrieved, it can be delivered to the application level.

3.4 Action manager

The action manager is a component external from the dynamics of the protocol introduced to control at run time all the processes, and to manage the events that happen during the simulation.

The action manager has the *allEvents* attributes, as described in Section 3.3.1, that is a dictionary mapping every process ID to its respective process instance. Also, the manager has three probability values: the probability p of creating a new process, the probability q of removing one process, and the probability $1 - (p + q)$ of creating a new event. Of course, $p + q <= 1$ must hold, and also according to the original paper, $p+q << 1$ – which means that event generation should be much more probable than event joins or leaves.

At every iteration of the protocol, an action is decided according to the above probabilities: in case a new event is generated, a random process in the network is chosen and it gets appointed to start spreading it; if a new process n is created, it is added to *allEvents*, and at least one node belonging to the network is inserted to the view of n; otherwise, a random node belonging to the network is chosen to be removed. Such a node sends a final gossip putting itself in the unsubscribe set, and then it is removed.

3.5 Builder

The builder deals with initializing the repast environment for the simulation. It collects the system parameters and it generates a *Continuous space* – in which nodes are visually displayed – and a *Network* – where processes are nodes, and gossip messages sent between nodes will be temporary edges (temporary because at the end of every iteration of the protocol the edges are removed, to show only the message exchange of the current iteration).

The builder also assigns to every node their initial view: the latter can be built randomly, giving to each node a random list of peers as view, or the nearest processes on the *Continuous space* are chosen as the initial view. The way views are created is another system parameter.

4 Evaluation

To analyze the protocol, several metrics are used. Let us define a graph $G = \{V, E\}$ where $V = \Pi$ – the set of processes that are part of the system – and $E = \{(p_i, p_j) | p_j \in p_i.view\}$. The graph is directed, since the knowledge of p_j by p_i does not imply that p_j knows p_i. In other word, this is the directed graph defined by the views of the nodes. From now on, only the graph G will be used, and not the visual graph representing the exchange of messages visible during the simulation of the protocol.

Let us now evaluate the behaviour of the protocol using different metrics.

4.1 Event notoriety

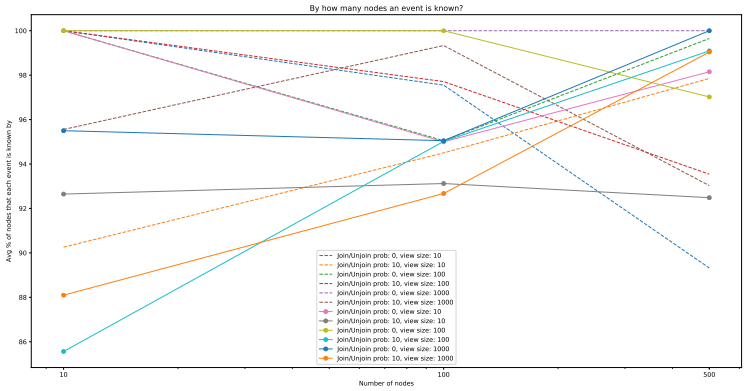

Figure 1: The average percentage of nodes that have received an event.

The first metric under analysis is the *event notoriety*: by how many nodes at the end of the protocol an event is known? The plot in Figure 1 shows the

different behaviours with respect to the parameters of the system: the dashed line represents the case in which the initial views of the nodes are chosen randomly, while the continuous line is the case in which views are made of the nearest nodes.

The plotted lines are the average of the fractions of nodes that have received an event over the total number of nodes, that is described by the following formula:

$$\frac{1}{|events|} \sum_{e \in events} |p \in \Pi : e \in p.view|$$

As can be seen in the plot (Figure 1), the stochastic nature of the algorithm gives very different results to similar situations – e.g. same number of nodes, or same size of the node's views. When the number of nodes is quite small – 10 or 100 processes – using an initial random view tends to help events to be known by more processes.

For the rest, is very difficult to find some regular pattern: usually the larger the node's views with respect to the overall number of nodes, the better, but there are some exception. In general, the most important result is that on average, all the events are known by at least the 85% of the nodes.

4.2 Rounds needed for an event to be spread

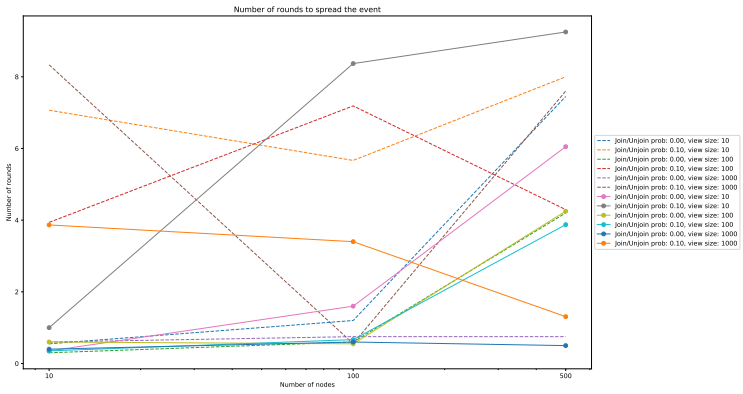

Figure 2: The average number of rounds taken by events to be spread around the network.

The next metric measures the average number of rounds needed for an event to reach all the processes that the event has been delivered to. Let $knownBy$ be the set of processes that have received an event e during the execution of the protocol. The number of rounds needed to build the set $knownBy$ is given by the round number when the last process added to $knownBy$ received e minus the round number when the event is born.

The plot in Figure 2 represents this metric. Dashed lines are the cases in which the initial views are chosen randomly, while the continuous one are the cases where the nearest nodes compose a process' view. As can be seen, on average knowing the nearest nodes from the beginning helps to spread event faster than the other case. Two different behaviours can be inferred by the results:

- The cases in which the system is stable, so no nodes join or leave the system – where Join/Unjoin prob is 0.00 – the number of rounds needed to spread an event is more limited. This is because once an event has reached all the nodes it does not have to be known by anybody else, since no new nodes will join the system. Of course, when the size of node's views is much smaller that the overall number of nodes, the average number of round increases.
- On the other hand, when nodes can join the system, the average number of rounds needed to spread around event is higher, because every time a new node joins, all the event are still unknown to the latter node, and so potentially at every iteration there is a new node that needs to receive some events, and so the overall number of rounds needed increases.

4.3 Clustering coefficient

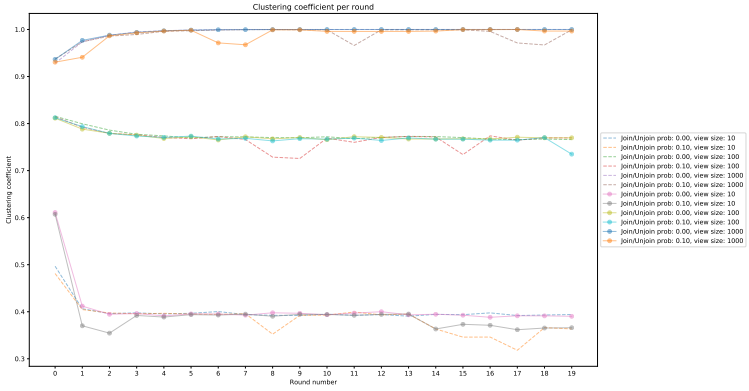

Figure 3: The clustering coefficient of the graph per round.

The clustering coefficient of a graph G is the average over the local clustering coefficient of all nodes in the graph. The local clustering coefficient of a node $n \in G$ measures how the neighbours of n are closed to be a complete graph.

The clustering coefficient is described by the following formula:

$$CC(G) = \frac{1}{|V|} \sum_{n \in V} \frac{|e_{nj} : j \in V, e_{nj} \in E|}{n_k(n_k - 1)}$$

where n_k is the number of neighbours of the node n. The higher the clustering coefficient, the higher the probability to receive redundant information, but also the chances to spread around information in a small number of rounds are higher. A clustering coefficient equals to 1 means that the nodes are a complete graph, while a coefficient equals to 0 means no connections between nodes.

As can be seen by the plot in Figure 3, also here the size of the views is the key parameter that makes the clustering coefficient value change. In case the view size is high – 1000 nodes – the coefficient tends to 1; with a medium size – 100 nodes – the coefficient is stable between 0.7 and 0.8; in case the views are small – only 10 nodes – the clustering coefficient is around 0.4, except for the very first rounds, where the initial views composed by the nearest nodes make such a metric to be around 0.6.

4.4 Average in-degree

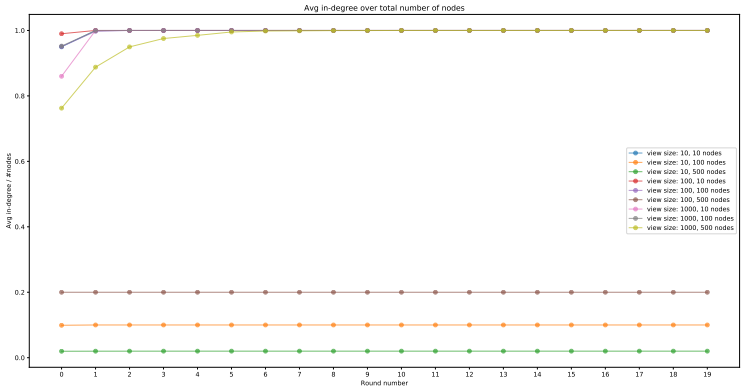

Figure 4: The average *in-degree* over the total number of nodes, round per round.

The average *in-degree* of G is the average over all nodes of their *in-degree*. The *in-degree* of a node n is the number of nodes that have n in their view. The formula to describe the average in-degree is the following:

$$AID(G) = \frac{1}{|V|} \sum_{n \in V} |\{p \in V - \{n\} : n \in p.view\}|$$

The higher the average in-degree, the higher the probability that nodes receives new information in a few rounds.

The plot in Figure 4 shows the ratio of the average in-degree with the total number of nodes (minus 1, since a node does not point to itself): a value of 1 means that a node is pointed by all the other nodes. It is visible that when the view size is small with respect to the total number of node, the average in-degree suffers, since every node can have a small fraction of all the nodes in its view. These are the cases with 100 nodes and view size = 10, 500 nodes and view size = 10, 500 nodes and view size = 100

When the views are larger, approaching the total number of nodes, the average in-degree always converges to 1 after a very few iterations of the protocol.

4.5 Average path length

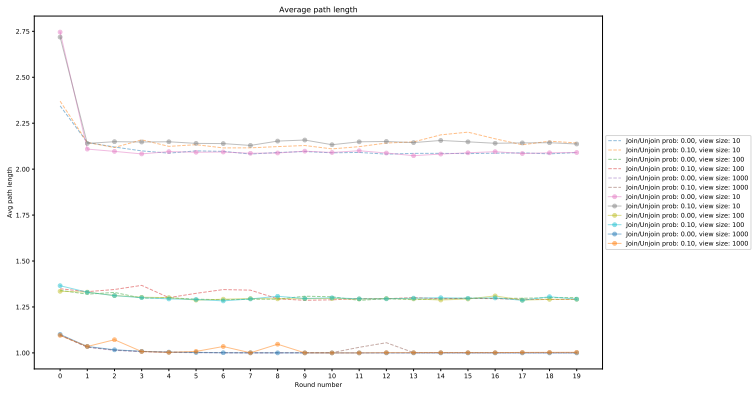

Figure 5: The average path length between nodes.

The average path length measures how many hops are needed to connect any couple of nodes in G on average. Of course, the longer the average path length, the longer the expected number of rounds needed to spread around information. The average path length is defined as:

$$l(G) = \frac{1}{|\Pi|(|\Pi| - 1)} \sum_{i,j \in \Pi, i \neq j} d(i, j)$$

In the graph defined by process' views, the distance between two nodes is not symmetric, so $d(i, j)$ may be different from $d(j, i)$.

As can be seen from the plot in Figure 5, the size of node's views is the parameter that makes the average path length change. Dashed lines are the cases in which the initial views are chosen randomly, while the continuous one are the cases where the nearest nodes compose a process' view.

It is clearly visible that three clusters of average path length are created: the one containing all the cases where the maximum view size is 10 has the highest average, while the case in which the maximum size is 1000 has the lowest average.

4.6 Robustness

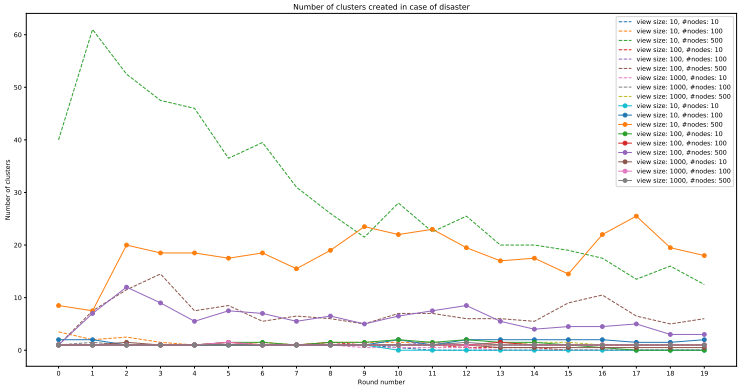

Figure 6: Number of clusters of nodes per round, in case of a catastrophic series of events.

To measure the robustness of the protocol, some simulation of a catastrophic series of events are performed, increasing the probability that nodes leave the network. In these cases, such a probability is equal to 0.8, which means that there are more nodes that leave than events generated.

What is measured here is the number of clusters of nodes that are generated by the continuous removal of members of the nodes. As can be seen in the plot in Figure 6, also in this case the worst performances are obtained when there is a great difference between the maximum size of node's views and the total number of nodes inside the system.

It is also visible that in the latter cases, using random initial views – the dashed lines – tends to cause a larger number of clusters during the execution of the protocol.

When the maximum size of the views is close to the overall number of peers, the maximum number of clusters that are generated by the continuous leaving of processes is limited 3-4 clusters.

5 How to run the simulator

The project must be imported into *Eclipse IDE*[2], assuming that the version of Eclipse in use has the plugin for Repast Simphony. Once imported, run `ProbabilisticBroadcast model` with the run icon, and a Java application window should pops up. Such a window is the graphical user interface of Repast that allows to set the parameters and to run the simulator.

By default, all the parameters are already set, but in the *parameter* bar it is possible to set them manually. Also the data loader is alreay set, and the `Builder` class is used as loader – see Section 3.5 for details about this class. Two types of datasets exist: one called *DS* that collects the metrics about events – event notoriety and number of rounds to spread an event – while the other one, called *TickStats*, collects information at every round of the simulation. These two datasets are written into files by two different file sinks.

To initialize the simulation, press the *Initialize run* button, while to start it, press *Start run*.

6 Conclusions

The implementation and the evaluation of the *Lightweight Probabilistic Broadcast* [1] have shown that under some conditions – given by the parameters of the system – the protocol is able to spread around information to a very significant portion of the network, using a limited number of rounds to do so.

The most important choice is trading off properly the number of nodes involved with the size of each process' view: the evaluation demonstrated that if these two parameter are very different – with the total number of nodes much larger than the maximum view size – the protocol may take a long number of iteration before delivering information to all the system.

References

[1] P Th Eugster, Rachid Guerraoui, Sidath B Handurukande, Petr Kouznetsov, and A-M Kermarrec. Lightweight probabilistic broadcast. *ACM Transactions on Computer Systems (TOCS)*, 21(4):341–374, 2003.

[2] https://www.eclipse.org/