DISTRIBUTED SYSTEMS 2

UNIVERSITY OF TRENTO

DEPARTMENT OF INFORMATION ENGINEERING AND COMPUTER SCIENCE

Implementation of a Gossiping Protocol with Append-Only Logs in Repast Simphony

Authors:
Andrea Traldi (ID: 213011)
Constantin Popa (ID: 211986)

Date: December 31, 2020

Abstract:

This document is intended to describe how gossip communication can be achieved using append-only logs and the Repast Simphony framework. We will start by briefly summarizing some of the key concepts of the original paper[2], including the *open gossip* and the *transitive interest gossip* protocols. Then, we will discuss the architecture of our application and how the connections are established between participants since the paper doesn't directly provide a description of the handshake procedure. Finally, we will analyze the performance of the proposed protocols, comparing the two variants, as well as evaluating them individually w.r.t some key parameters, in order to show what are the important aspects that impact the performance and make our suggestions.

1 Protocol summary

The models that are going to be described take inspiration from the broadcast-only communication model based on replicated append-only logs[4] and build on top of it in order to obtain a real-world social application. They make a step forward by reducing the flooding effect and disseminating the information in a more efficient way, although the algorithms don't include the handshake procedure, which is described in a separate document[3]. The general idea is to select many pairs of participants and make them exchange the missing updates, instead of broadcasting them to all the neighbors. Events are the evolution of perturbations, and they can represent both locally generated computations or messages received from other participants. These events are organized in *logs*, each participant having its own append-only log, with a set of operations that can be performed, such as retrieval and update. The append-only property means that only the owner of the log can add new events to the chain, while all the other participants can't modify it. In order to guarantee this property, the digital signature mechanism is used.

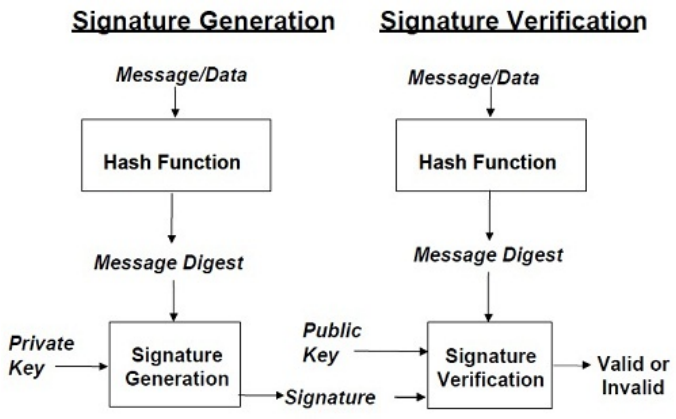

Figure 1: Signature of an event and verification of its validity

Before appending an event to the log, a digest is calculated on the content using a hash function. Then, the private key of the log owner is used for generating the signature and finally, the event gets appended to the log. On the other side, when another participant receives the same event as a copy, it has to decrypt the message using the owner public key and compare the output with the locally computed digest. If the two hashes are equal, the signature verification succeeds and we are sure that the event has been appended by the true log owner and not a malicious participant.
Each participant's *store* (an abstraction of the persistent storage) has a series of associated operations (log retrieval, frontier retrieval, log creation, etc.) and is used to organize the collection of logs. When two participants establish a connection, they exchange the *frontiers* (lists containing the most recent index per log), then figure out what updates the other participant needs, and finally exchange this information so they can update their stores accordingly. The goal is to make the logs replicated at multiple stores become eventually consistent.

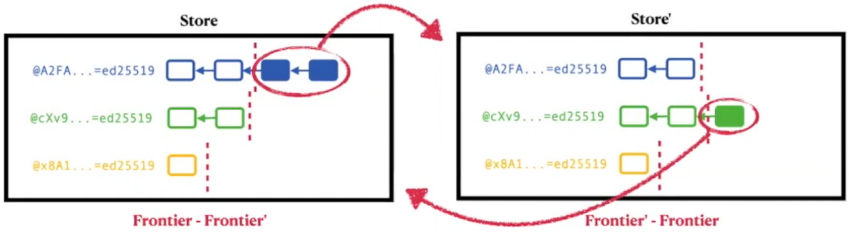

Figure 2: Two participants exchanging missing updates

The paper introduces two protocol variants for implementing how the participants get in contact and exchange these updates:

- **open gossip protocol:** easy to implement and suitable for small and trusted groups. However, participants have to store a lot of data generated by all the other participants and they can't choose what information can be stored on their devices.
- **transitive interest gossip protocol:** more complex but it captures better the social aspect of the application. It is also resistant to spam and sibyl attacks, giving the owner the possibility to choose which logs get replicated. Since the focus is on social environments, there is the need to introduce primitives for showing explicit interest/disinterest for other participants, such as follow and block operations.

Finally, both protocols assume the update exchange happens over a reliable connection, therefore there is no loss of data and there is no need for a retransmission mechanism.

2 System architecture

Since the suggested protocols are built on top of the previous research on the communication using append-only logs, also the corresponding architecture is an evolution of the previous one. Similarly to our other implementation, we have chosen agent-based modeling as a paradigm for simulating the proposed protocols. The components were designed by keeping in mind best practices such as modularity and reusability, and they were grouped in the various packages based on their logical role. Let's now have a more in-depth look at each of them and their relationships.

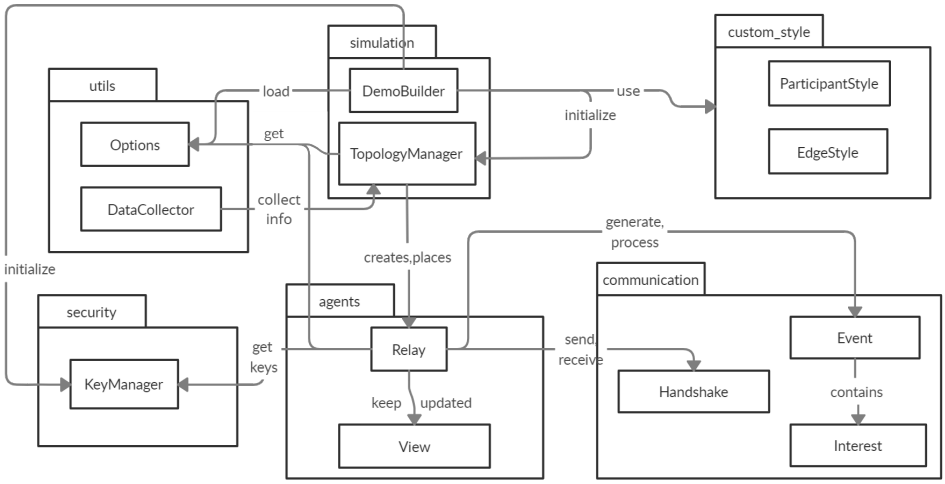

Figure 3: Components and main interactions among them

2.1 Agents and views

The only agent of our model is the *Participant* class and it represents the owner of a *store* data structure. The main purpose of a participant consists of keeping their associated store always updated, by establishing connections with random peers. But since the two protocols operate in a distributed manner, there's a high chance of conflicts when multiple participants want to exchange updates with the same peer. Therefore, there is the need to make sure only one connection is established per participant at a time and the number of global connections is maximized as well over the whole network.
Ideally, we would like to have every participant pairing with another one, and no unpaired participants left. Although this is quite unlikely - unless the protocol becomes centralized - it is still feasible to obtain a good number of connections by adopting a probabilistic approach. In order to start exchanging news, each participant needs either to be contacted by somebody or contact a random peer itself. In the *Evaluation* section

more details will be provided on how and what is the optimal value of this probability in order to maximize the number of paired participants.
The other problem regards the fact that conflicts need to be solved efficiently and participants should agree before they start an update exchange. Since the paper describing the two gossiping protocols omits the description of a handshake procedure, we have taken inspiration from related paperwork[3] published previously for the *Secure Scuttlebutt* implementation to create our own.

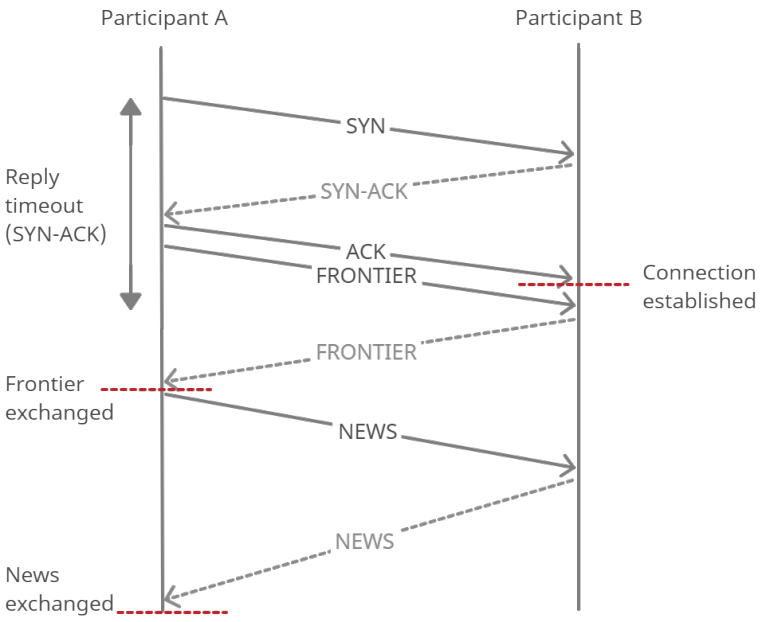

Figure 4: Two participants establishing a connection through a 3-way handshake procedure and exchanging frontiers and news

Each participant samples a random peer from their view when they are willing to start an exchange (in this specific example it is *Participant A*). A *view* is nothing else than a collection of addresses of other participants who can be contacted. Views have a limited size and they are updated with the most active participants in order to minimize the probability of contacting offline/crashed participants. *Participant B* checks whether there are any incoming requests (*SYN messages*). In case there are more than one request available, it chooses one of them randomly, and replies with a *SYN-ACK* (replies are distinguished by dashed lines).
Both participants set a timer after each message that is sent in order to avoid waiting indefinitely. In *Figure 4* there is one example of timeout for the *SYN request* sent by *Participant A*: in case of the *SYN-ACK* not arriving inside the specific window, *Participant A* would have aborted the handshake and contacted a different peer.

SYN-ACK is replied with an *ACK* message in order to confirm that the connection is established, then right after it, the frontier is sent as well. The other participant replies with its own frontier. Both peers use these frontiers to calculate the updates the other participant needs and finally send the updates. When the news exchange is done, the connection is closed and the process is repeated with a new peer.

2.2 Events and security aspects

An *event* represents the most basic data structure used for exchanging information in the gossiping protocols. *Logs* are formed by chaining multiple events from a single participant. The *id* of each event corresponds to the public key of the creator and it is generated by the *KeyManager*. The *content* of the event is either the application-specific payload or a signal for showing interest/disinterest. Participants use instances of the *Interest* class in order to record these follow/block operations and also communicate them to the other peers by putting an object of type *Interest* as the content field of a new event.
When participants receive a copy of an event created by another peer, authenticity and integrity need to be enforced. These two properties are guaranteed by the digital signature mechanism described in the previous section. If the digital signature verification fails, the event won't be added to the log. Cryptography lets also participants encrypt their data and hide it from the other participants by encrypting the content of their events. Previously these tasks were delegated to an external entity, but this implementation encapsulates the encrypting/decrypting operations inside the *Participant*'s behaviour.
Finally, the *Handshake* class defines the *SYN, SYN-ACK* and *ACK* messages. These signals are special types of messages that are useful in the initial phase of the connection establishment and before the exchange happens.

2.3 Simulation environment and customization

The *DemoBuilder* is responsible for bootstrapping the simulation and coordinating the other components such as the *KeyManager* and the *TopologyManager*. *TopologyManager* encapsulates all the logic related to the creation of the topology and its maintenance, handling also the crashes and the arrival of new participants. For our implementation, the previous 4 topology types were reused (random, extended star, line and ring) but this time they have a visual effect only. If we constrained the participants to establish a connection with only the nearby peers, the performance of the gossiping protocols would fall drastically since participants have less available options when performing the handshake procedure, and even leading to starvation for some participants in the *transitive-interest gossip* protocol. Although the latency might increase when contacting a peer that is placed far away respect to the participant that initiated the handshake, this is an acceptable trade-off compared to the global number of connections that are

possible without this kind of constraint. For example, sending 100 *MB* worth of news during an exchange makes little to no difference if the other participant is placed in the same city or in a different continent. The latency measured in milliseconds is negligible, while the bandwidth is more important and has a larger impact on the time it takes to exchange the news. If one of the peers has a 10 *MB/s* bandwidth, then it would take 10 seconds to upload the news to the other peer. Therefore, even if the latency was of the order of 100 or 200 milliseconds, it is still much lower than the actual upload time, which becomes the only relevant element when evaluating time complexity.
DataCollector class is responsible for collecting information about the simulation at runtime and printing it to a permanent file, which can later be used for generating graphs. The *Options* class groups together the user-defined parameters for the current simulation by reading the user inputs and making them available to the other components (*Participants, TopologyManager*, etc.).
Finally, the participants and links among them need to be graphically customized so that it becomes easy to identify their state based on the fill color. This task is achieved by the *custom_style* package. More specifically, we have chosen the following semantic for representing the different states a connection goes through:

- **red links**: *Participant A* has just sent a handshake request to the other participant;
- **orange links**: *Participant B* has chosen *Participant A* among the available requests and replied with *SYN-ACK*;
- **green links**: both participants agreed to exchange news and the connection is now established.

Similarly, the participants use the same colors but in reversed order for displaying their status. Their shape is a circular dot and it has the following colors:

- **green dots**: these are the available participants. They are not involved in any communication, so they are soon going to either initiate a handshake procedure or accept an incoming request if there are any available;
- **orange dots**: participants who are involved in a handshake procedure, but the connection has still to be established;
- **red dots**: participants that are busy as the connection has been established and they are exchanging news with another peer.

3 Implementation

Now that we have an overview of the architecture and what is the role of each component, we are going to explain how the key elements achieve these goals using Java and Repast Simphony constructs. The implementation follows closely the pseudo-code of the two gossiping protocols presented in the paper, as well as the set of operations provided on the *store* and *log* data structures. The protocol variant can be chosen using the configuration parameters, which will be described in the next section.

3.1 Topology management

When the simulation starts, the *TopologyManager* creates the number of participants specified by the user, assigns them a globally unique *Public Key* (which serves also as an identifier), and places them according to the chosen topology in the continuous 2D space. The logic used to position the participants in order to form a specific topology was described in detail in the previous report[1]. Since the public key has too many characters to be displayed as a label under each participant during the simulation, an auto-incrementally integer is used instead. Upon participant creation, *TopologyManager* prepares also the view of each participant, by selecting a partial subset of the whole population and adding them to the view. After this initialization, it will be the participants' task to keep the views updated and decide which peers are removed or added to it. Finally, this component asks the *KeyManager* to generate new pairs of keys for all the participants, including the ones that will join later during the simulation.
Simulations start with a number of participants equal to the capacity chosen by the user, therefore at the beginning, there are no available locations for new joining participants. However, a location is freed up when a participant crashes. The *TopologyManager* has the task to remove it from the context and mark its location as available. As soon as new locations are made available, the *TopologyManager* simulates the arrival of new participants by generating them based on the probability parameter (more on this in the section dedicated to the parameters) and applying the same bootstrapping operations which were listed for the initial participants.

3.2 Store and logs

The order of the logs inside the store is irrelevant, but the retrieval and removal operations need to be optimized. For this reason, the store of each participant was implemented using a *HashMap* instance, which guarantees *O(1)* time complexity for these operations. The key of the *Map* is represented by the public key of the log owner, while the value is the chain of events itself (the *log*). Unlike logs inside a store, the order of the events contained has to be preserved, hence an *ArrayList* is more appropriate for the *log* data structure. Finally, frontiers are *HashMap* objects that map the same key

identifiers found in the *store* map, to a simple integer value which represents the index of the last event per log.
Java offers built-in tools for handling hashing operations and digital signatures. The *previous* field of each event is calculated by calling the *hashCode()* method on the object itself, already provided by the Java language. The computation of the signature follows a more complex process: firstly all the fields of the *Event* class (*id, previous, index, content*) are transformed into a *byte array*. Then, the *MessageDigest* class computes a digest using the *SHA-256* technique. Finally, the signature is obtained by passing the digest and the private key of the owner to the *Signature* class (also provided by Java) and specifying the *RSA* algorithm. Other participants who want to verify the signature will have to use the same class, but passing the public key instead.

3.3 Participant life-cycle

Most of the actions that a participant chooses to perform during a tick, such as event generation, handshake initialization and follow or block (in case of the *transitive interest gossip* protocol), depend on some probability values. At each tick, a random value is generated, and if the outcome is lower than a certain threshold, then the action is executed. For example, during a simulation with 100 participants, each having 0.1 probability to generate an event at each tick, and an execution that lasts 5000 ticks, we are expecting to have the following number of events by the end of the simulation:

$$n = probability * n\_peers * n\_ticks = 0.1 * 100 * 5000 = 50000 \quad (1)$$

If we wanted to map this example to a real-world case, we could assume that each tick is equivalent to one second and every second there is 10% of participants generating a new event. Furthermore, crashes follow a similar logic: they are rare but not absent, however, values like 3 or 4 *nines* reliability (0.999 and 0.9999), or in other words, 0.001 and 0.0001 probability to crash shouldn't be hard to achieve. By applying a similar formula as the previous one, and considering the case of 99.99% reliability, we are expecting approximately 50 participants to crash and 50 participants to join in a population of 100 peers over a period of 5000 seconds. We are safe to say that this kind of situation can be considered as a worst-case scenario since the outcome will most likely be more optimistic than that in a real setting. Therefore, a probability to crash of 0.0001 can be used as a lower bound.
As anticipated before, bandwidth has a major impact on the time it takes to exchange updates between peers, rather than latency, which becomes negligible. The values used for the bandwidth and the events should be assumed to be measured in *MB* and *MB/s*. For example, having a bandwidth of 10 *MB/s* (which nowadays is common enough), and an amount of 350 news (usually for the setup described earlier this is the upper bound for an **individual** exchange), the expected upload time is the following:

$$upload\_time = \frac{n\_events * event\_size}{peer\_bandwidth} = \frac{0.1MB * 350}{10MB/s} = 3.5seconds \quad (2)$$

If we were to add the overall latency time, this would add a few hundred milliseconds to the above results, however, the final number would still be under 4 seconds.
Now that the mechanism used to perform the main participant actions are clear, we can go through the states that together define the participant life-cycle.

- **Available:** the participant checks whether some other peer has sent a handshake request, otherwise it sends one itself, with a certain probability, and by sampling a random peer from its view. Please note that it's also possible that the participant skips the current tick and decides to initiate a handshake procedure at a later time.
- **SYN sent:** this is the immediately following state after the *Available* state, in case the participant chose to initialize a handshake procedure. The participant will wait for a *SYN-ACK* message until the timeout expires.
- **SYN received:** this is the immediately following state of the *Available* state, in case the participant chose to accept one of the incoming requests, rather than initializing one itself. The participant will wait for an *ACK* message in order to have the confirmation that the other node is still willing to establish a connection until the timeout expires.
- **Connection established:** Participants go into this state when they receive either the *SYN-ACK* or *ACK* messages, depending on their role (initiator or acceptor). Peers will soon exchange their frontiers.
- **Frontiers exchanged:** Based on the other peer's frontier, each participant calculates the missing updates and starts uploading them.
- **News exchanged:** When the exchange is done, participants start applying the updates to their store. In the case of the *transitive interest gossip* protocol, peers update also their knowledge as regards the follow and block operations performed by the other participants. During this phase, participants update their view as well, based on the activity of the other peers, which can be deduced from the *news* data.
- **Timeout:** While waiting for a reply in any of the previous states, participants can timeout and abort the exchange. This is necessary in order to detect failures from the peers and avoid waiting indefinitely, so the participants can establish new connections if something goes wrong.
- **Finished:** Participants close the connection either at the end of a successful outcome of the news exchange, or after a timeout of any of the previous states. The next step after this phase is repeating the cycle again, starting from the *Available* state.

4 Performance analysis and parameter tuning

Before diving into this section, please notice that the charts have been aggregated inside the Charts folder where it is possible to view them at a higher resolution.

In order to properly evaluate this implementation, the results will be extracted from many different simulations. For each of these, there will be constant parameters that will define the specific baseline of the evaluation trying to represent a *realistic* scenario for the reasons shown in the previous section. The actual values taken into account for the simulations are the following:

1. Fundamental parameters:
    - **Ticks of execution:** 1000 ticks/seconds
    - **Random Seed:** 1.810.052.790
    - **Dimension of the environment:** 65 (measured in Repast Simphony space units)
    - **Number of participants:** 41
    - **Event size:** 0.1 (measured in MB)
    - **Bandwidth:** 10 (measured in MB/s)
    - **Probability of new participant joining:** 0,01 (per tick/second)
    - **Probability of participant crash:** 0,01 (per tick/second)
    - **Probability of random delays:** 0.0
2. *Open* **and** *transitive interest gossip* parameter:
    - **Probability of generating a new event:** 0,01 (per tick/second for each participant)
3. *Transitive interest gossip* (only) parameters:
    - **Probability of following a random participant:** 0,005 (per tick/second for each participant)
    - **Probability of blocking a random participant:** 0,005 (per tick/second for each participant)
4. Handshake parameters:
    - **ACK timeout:** 2 (ticks/seconds)
    - **Frontier exchange timeout:** 2 (in ticks/seconds)
    - **News exchange timeout:** 5 (in ticks/seconds)

However, we are still missing one important parameter: the probability of starting a new communication with another participant. In fact, during the implementation, we discovered an unexpected correlation between this value and the overall performance; hence we decided to run a number of tests in order to discover the optimal value.

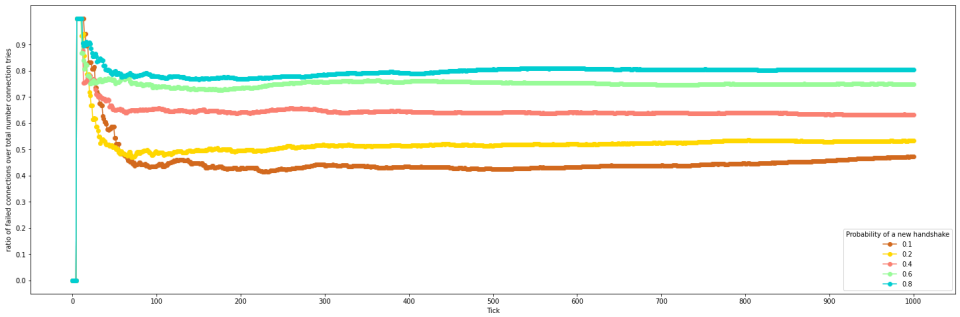

Figure 5: Ratio between the number of failed connections over the total number of connection attempts performed by the participants

In order to find this value and use it for the rest of our evaluations, we have computed the amount of failed connections over the total number of connection attempts performed by the participants w.r.t. the probability of starting a new communication. As we can see in *Figure 5*, the best solution is counter-intuitively to take a low value of probability (such as 0.2 or 0.1) in order to reduce the ratio of failed connections. The reason why more connection attempts don't necessarily lead to more successful connections is due to the fact that most of the participants will be waiting for a response from another participant who is waiting for a response as well and so on. On the other hand, if there are fewer peers that initiate the requests and many peers that are available to accept them, there is a high chance these participants won't be busy and they will reply to the handshake.

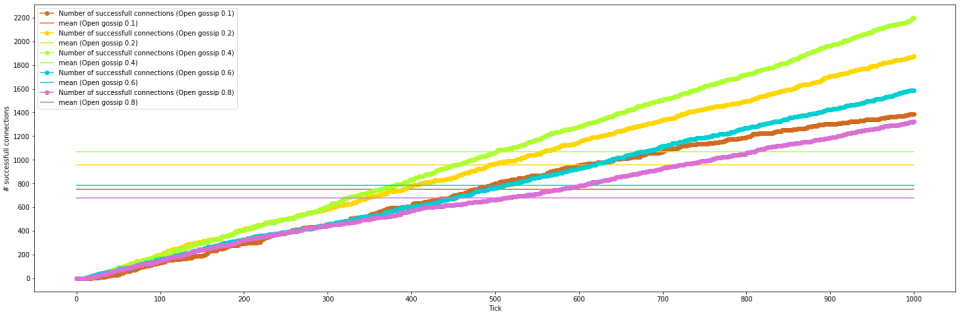

Figure 6: Comparison on the absolute number of successful connections with different probability values

However, even though values below 0.2 might seem to be better since they offer even fewer failures, another trade-off must be made. For example, the value 0.4 offers the

best outcome when it comes to the absolute number of global connections, as shown in *Figure 6* above, even though its success ratio was not as good as the one obtained for the 0.2 and 0.1 values earlier. In the end, the value 0.2 was chosen since it offers a very similar total number of connection compared to the value 0.4 and much higher than 0.1. At the same time, it has the advantage of saving on network resources since there will be fewer failed connections, unlike the 0.4 value, which doesn't bring a worthy improvement. Finally, we can add the last parameter:

- **Probability to start a new handshake:** 0.2 (per tick/second for each participant)

4.1 Dynamic Network

We have decided to monitor the number of participants present in the network at each tick to get an overview of how it evolves.

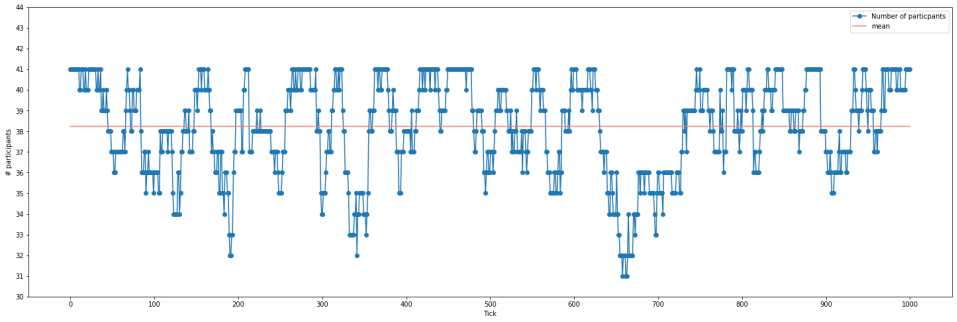

Figure 7: Number of participants

As shown in *Figure 7* the network is dynamic enough with the selected parameters, covering a good variety of situations (low vs high churn). In both cases, the correctness of the protocol was preserved and the simulations converged (convergence proof will follow soon).

4.2 Successful connections

Previously we have discussed what is the optimal parameter for establishing new connections in the general case in order to maximize the ratio of successful connections. Now it's time to make an evaluation on the absolute number of the successful connections instead. Furthermore, we are trying to understand how the *transitive interest gossip* protocol is affected by the follow and block operations.
In *Figure 8*, we can see that by blocking the communication with some of the peers, the whole network is affected as there is a higher competition between peers and more request conflicts. This results in an overall decrease of approximately 350 less successful

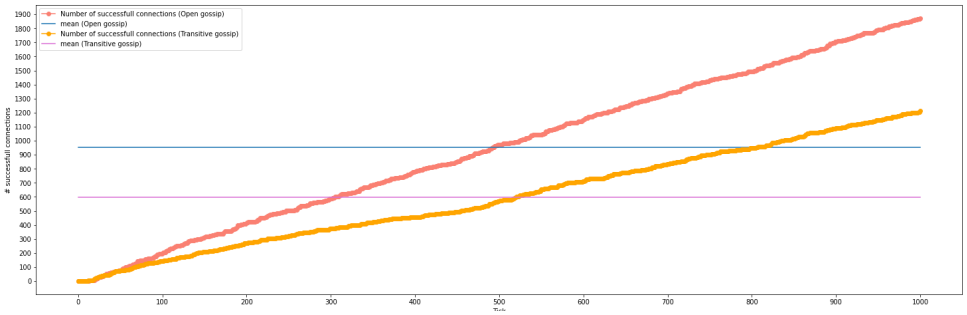

Figure 8: Comparison on the absolute number of successful connections between *open* and *transitive interest gossip* protocols

connections over 1000 ticks of execution, w.r.t the *open gossip* variant, or in other words, approximately 35% less successful attempts.

4.3 News

The following results are extracted from simulations performed on the *open gossip* protocol only since there are no significant differences between the two variants when considering the amount of news aspect.

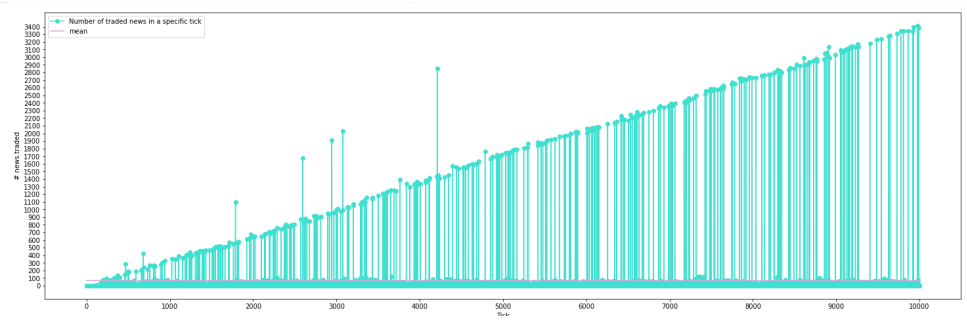

Figure 9: Number of news exchanged per tick

First of all, we have measured the number of news that are exchanged during each tick (*Figure 9*). It is possible to see that the values linearly increase, due to the fact that as the execution goes on, many new nodes join and need to be updated on the whole history of the participants. This leads to a progressive increase in the number of events that have to be exchanged. For this reason, we consider that the two protocol variants should take this aspect into consideration, in order to avoid network congestion in the presence of multiple participants joining at the same time. The current protocol would cause huge network-intensive exchanges between a new participant and an old one, preventing the latter to make progress or even leading to timeouts. One of the solutions might be the segmentation of updates in smaller parts and involving multiple sources

in order to distribute the workload of the peers. Finally, in the same figure, it is worth noticing that the mean is very low due to the fact that there are no exchanges of news in most of the ticks since the peers are attempting to create new connections through the handshake procedure instead.
We have also measured the total amount of exchanged events up to a specific tick, as shown in *Figure 10*. The difference between the previous graphic and this one is that the former proofs that the **individual** exchanges consist of more and more news as time advances, while the latter gives an idea about how many news were exchanged **globally** before the simulation converged.

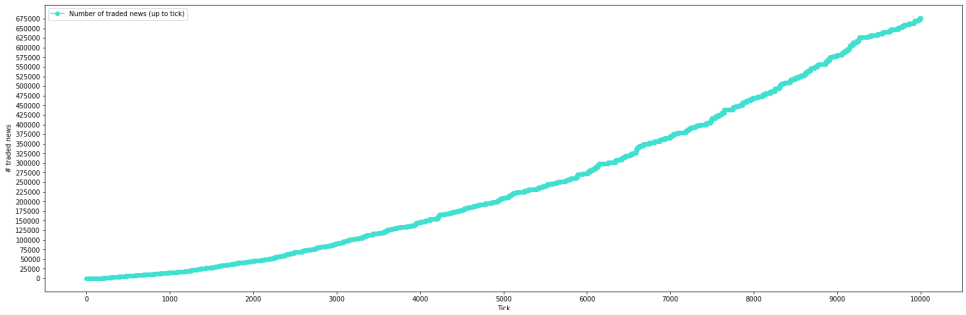

Figure 10: Number of news traded up to a specific tick

This sort of evaluation helped us to better understand the order of the magnitude for the exchanged events, visualize the slope of the line and come up with coherent examples in the *Implementation* section.

4.4 Bandwidth

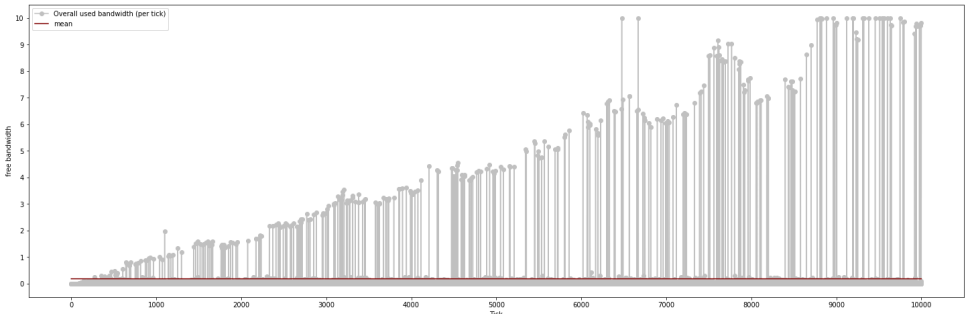

Figure 11: Used bandwidth per tick

Figure 11 shows how the available bandwidth decreases over time since it is directly dependent on the number of events exchanged in the network (Figure 9). It is clear then that the protocols need to either handle the old logs from participants that have not been active for a long time in a special way (e.g: avoid their dissemination unless

log owners come back online), or increase the participants' bandwidth; otherwise, the execution might face a performance bottleneck (like the one around tick 9000 and afterward where the entire bandwidth is consumed).

4.5 Convergence

Finally, we have measured the time it takes for the simulation to converge. This was achieved by starting the execution and letting the participants generate events up until a certain point (200 ticks). Afterward, participants stopped creating new events but continued exchanging the already existing ones. The goal was to observe the time that is needed in order to have the information disseminated to all the peers (100% of the events in the store).
The first experiment consisted of measuring the dissemination time of a single event, hence we paused the generation of new events right after the generation of the very first event. We then proceeded to count the number of participants that had that specific event in their store.

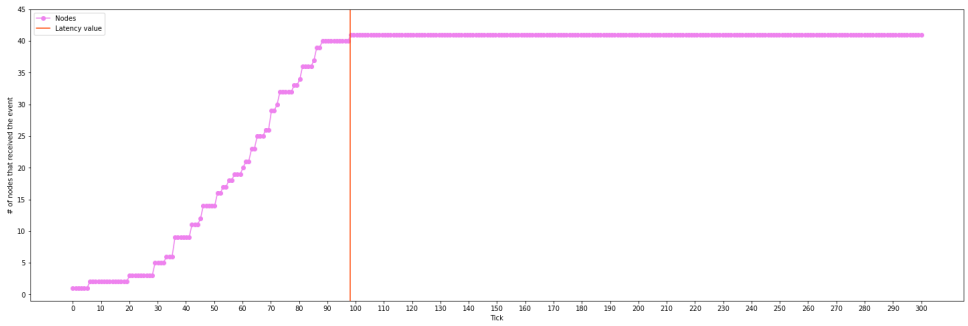

Figure 12: Dissemination time

Looking at *Figure 12* it's possible to notice that the dissemination time has a value of around 100 ticks for a single event, before it reaches every single participant. Starting from this result, we measured the convergence time in presence of multiple events. In practice, at each tick, we have kept track of the number of events present in the store compared to the total number of events that were generated. The ratio between the two gives the percentage of the missing updates and an estimate of how near the convergence is. In the case of the *transitive interest gossip* protocol, we didn't consider the events generated by blocked nodes when calculating the percentage, otherwise it would've been impossible to reach 100% status at all the nodes (*Figure* 13).
Figure 14 shows a vertical line on tick *200* which indicates the moment where we decided to pause the generation of new events. After this tick, participants could only exchange updates, without creating new ones. The outcome was that around 200 ticks later (i.e. at tick 400) all the participants have reached the 100% update ratio and the

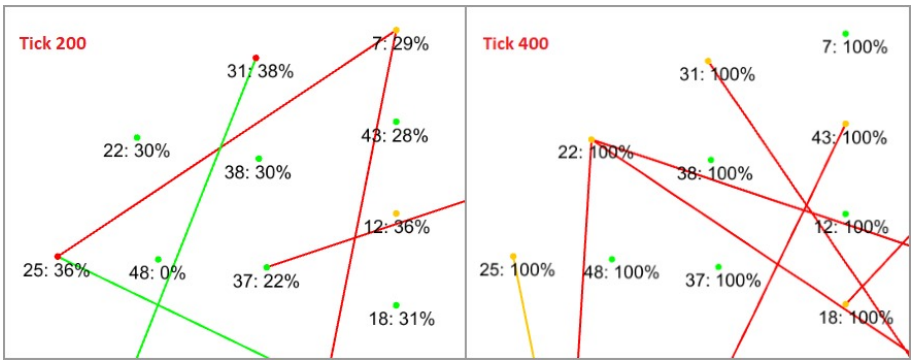

Figure 13: Example captured during a simulation that shows how the completion ratio progresses

simulation converged. From the very same chart, we can notice two different versions of *transitive interest gossip* protocols: the *soft* one and the *hard* one. These two protocols differ due to the absence (soft) or presence(hard) of *line 4* in *Algorithm 3* that defines the *transitive interest gossip* protocol in the original paper[2].

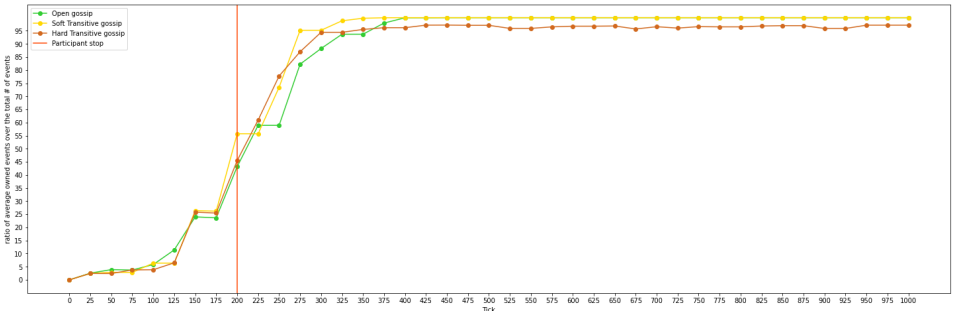

Figure 14: Convergence time comparison for different protocols

Informally speaking, this specific line of code imposes that each participant should remove a log from its stores when it gets in contact with a peer that had blocked that participant. The problem is that eventually, another participant will ask for the log that has just been removed, therefore it will be added back to the store. This leads to a vicious circle where one participant asks for the log to be removed, and another one for it to be added back. This way, convergence is impossible to reach in a normal execution and typically there will be continuous jumps from approximately 90% update ratio to 100% and back. We suggest that this algorithm could implement a more complex mechanism in order to decide when a log is removed from the store, in order to avoid unnecessary waste of network resources on retransmitting the logs whose removal could be avoided. In order to have a better understanding of the performance of our implementation, we have decided to dig deeper into this topic and create a few variations of the previous evaluation, by acting upon the number of participants and the execution time. Note that, since the three protocols behave similarly, the following evaluations will be per-

formed on the *open gossip* protocol. Moreover, we decided to keep the network static for this scenario only, by removing the possibility of participants to join or crash. The main reason that leads us to change the simulation parameters is that, otherwise, participants could join, create events and then crash immediately after, without having any kind of communication in the meantime with other participants. Clearly, this would've deprived the other nodes to collect every event generated, causing many store with lacking updates.

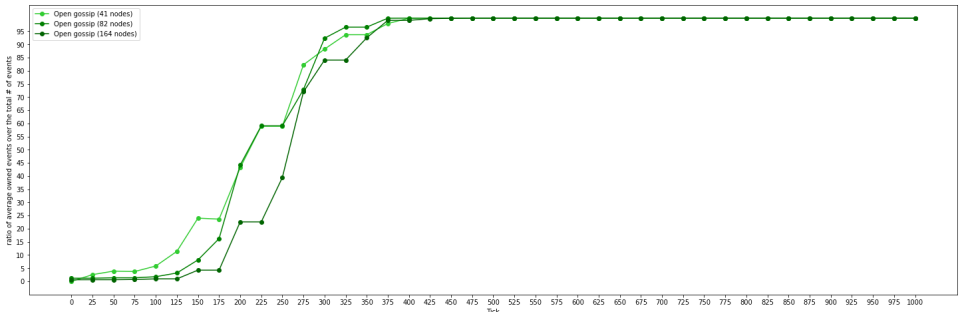

Figure 15: Convergence time with different number of participants

Firstly, we decided to measure the convergence time of the network in scenarios with a different amount of participants: 41, 82 and 164. All the other values instead remained constant. The results, with reference to *Figure 15*, show that the convergence time does not depend on the amount of participants. In fact, even if more peers have to be updated on the others' log, there are also more peers providing information. Furthermore, we changed the interval of time in which the participants were able to generate new events. In the same way as before the other values are kept constant.

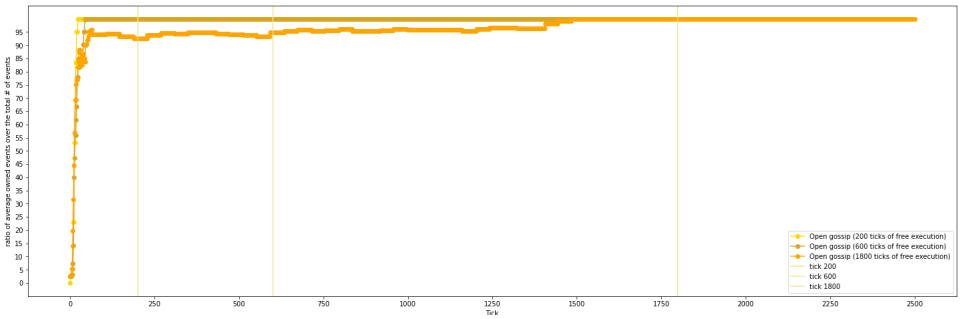

Figure 16: Convergence time with different time intervals for generation of events

As it is possible to see in *Figure 16*, the network updates itself quickly, resulting in a completion of the various stores even before the moment in which participants stop generating new events (exhibited in the chart as vertical lines, positioned at tick 200, 600 and 1800). This denotes how the protocol, deprived of the issue presented in the *News* sections, has high performances and is able to keep a network fully updated (w.r.t. our scenario).

5 Conclusions

This simulator implemented successfully both the *open* and the *transitive interest gossip* protocols, using the primitives proposed for managing the *log, store* and *frontier* in the original paper [2] and a set of real network conditions such as churn and bandwidth limitations. We have extended the proposed concept by adding a handshake procedure, in order to manage conflicts and maximize news exchanges. In the implementation section, we have discussed how certain aspects impact the performance (latency vs. bandwidth and low handshake probability vs high handshake probability). Finally, we have run a number of experiments in order to back up our previous statements, find the optimal values for some parameters or simply show their order of magnitude. We have learned that the implementation surely offers adaptability to the application needs and robust data replication. Both protocols offer excellent scalability as well in the presence of low churn since the converge time stays constant when the number of participants is increased. The *transitive interest protocol* has shown to have a significant decrease in the number of successful connections, but it surely becomes an acceptable trade-off when there's the need to have control over the stored logs. We have also highlighted the two situations where the protocols could be improved: in presence of high churn and low bandwidth, none of the algorithm scale well since congestions might arise; furthermore, *line 4* of *Algorithm 3* makes inefficient use of network resources leading the *transitive interest gossip* protocol to oscillate between 90% and 100% of convergence.

6 How to install the simulator

Once the *setup.jar* file was downloaded from the link found in the repository's *README*, it's possible to follow these steps in order to install and execute the simulator:

- Make sure JRE 11 is installed on your system.
- Run the *setup.jar* using the system GUI or by typing *java -jar setup.jar* in the terminal. Follow the wizard steps in order to accept the license and choose the installation directory.
- Launch the simulator by executing *start_model.bat*. A Java application will be displayed on your screen. If you are using an Apple Mac device then you have to execute *start_model.command*.
- Set the values of the parameters from the *Parameter* window. Once you have chosen the parameter values, it is possible to save them for further executions.
- Finally, you can launch the simulation by clicking on the *Initialize Run* button placed on the upper bar. You can start the simulation by clicking on *Start Run* button or by clicking on the *Step* button. Simulation execution can be slowed down by increasing the *Schedule Tick Delay* in the *Run Options* window.

References

[1] Constantin Popa Andrea Traldi. Implementation of a broadcast-only communication model based on replicated append-only logs in repast symphony. pages 8

[2] Anne-Marie Kermarrec Christian F. Tschudin, Erick Lavoie. Gossiping with append-only logs in secure-scuttlebutt. pages 2, 17, 19

[3] Dominic Tarr. Designing a secret handshake: authenticated key exchange as a capability system. pages 2, 5

[4] Christian F. Tschudin. A broadcast-only communication model based on replicated append-only logs. pages 2