Distributed Systems 2 - Project II
An implementation of Gossiping with Append-Only Logs

Massimo Mengarda 214290
Matteo Contrini 215037

December 2020

1 Introduction

This report shows the implementation of a gossip algorithm based on append-only logs, as described in the paper *Gossiping with Append-Only Logs in Secure-Scuttlebutt* [1].

The implementation was realised using the Repast Simphony framework, and it is based on the work of the previous project on the broadcast-only communication model. Some features were removed for simplicity, but the base communication implementation has been reused.

In the following sections we first introduce how the Secure-Scuttlebutt gossip algorithm works, we then present how the project was structured and how it is based on previous work, and finally we evaluate how the implementation performs to understand if and how the model can be improved.

2 The append-only gossip model

The gossip algorithm used in Secure-Scuttlebut and introduced by Kermarrec, Lavoie and Tschudin [1] is based on a model where logs are replicated to all the participants, with the objective of building social applications that are eventually consistent.

The gossip model acts as a middleware for the social application, and it can be implemented in two variants: the *open* model and the *transitive-interest* model.

The two cases share most concepts, and what changes is the algorithm that performs the gossiping and defines how logs are replicated.

Logs are sequences of events that are generated by participants, and each participant collects them in a *Store*. In this model, the gossip algorithm represents the way these logs are replicated in the stores of all the (interested) participants.

Logs are guaranteed to be *append-only*, because each event in the log contains a cryptographic signature based on the attributes of the event and the hash of the previous event in the log. Since the signature is computed with the public-key signature algorithm *Ed25519*, other participants can only verify the log (with the public key of the participants to which the log belongs) and not tamper with it.

As mentioned, gossiping can be done in two ways: with an *open* model or a *transitive-interest* model.

In the first case, each participant will broadcast its store to all the other participants. When a participant receives a store, it will look at the logs contained in it and update the local store, by creating missing logs and inserting missing events. This is done through specific operations defined on the store and on the log, such as the `frontier`, `since` and `update` methods.

With the *open* model, all the participants in the system will replicate all the events generated in the system. In the simplified model presented by the paper, what participants exchange is the whole store with all the logs and events, while a more efficient implementation would only share new events.

An improved model is the *transitive-interest* model, where participants explicitly signal through the *follow* operation that they are interested in other participants. Follows are transitive, meaning that if a participant A follows another participant B, A will keep track not only of B's events but also of all the events generated by participants that B is following. In social applications this behaviour is often called "friends of friends", but it can also be extended to consider a larger number of "hops" (e.g. friends of friends of friends).

The transitive-interest gossip algorithm achieves the above by introducing additional operations like *follow* and *unfollow* on the log. It also provides the possibility of blocking participants, so that participants that are transitively followed can be ignored if they are not of interest for a user.

3 Architecture and implementation

In this section, we explain how we architectured the solution and implemented the model. As mentioned above, the implementation was based on the broadcast-only model from the previous project, and we called this part the "transport" layer. We then implemented the gossip algorithm as the "application" layer, that makes use of primitives provided by the transport layer.

3.1 The "transport" layer

The relay implementation from the first project was partially reused. We took the `Relay` class and applied some tweaks, in particular to remove unnecessary features.

The only operation that the `Relay` class exposes as an API is now the `broadcast` method, that takes a message and broadcasts it as a perturbation. The point-to-point and pub-sub communication mechanisms were instead removed from the implementation.

We also removed the ARQ (*Automatic Retransmission reQuest*) mechanism, because the way the gossip algorithm works makes it acceptable to lose messages while keeping the eventual-consistency behaviour. This helped us to simplify the underlying protocol and to achieve huge performance gains: in fact, the ARQ mechanism was by far the most computational intensive task for the simulation.

As a consequence of removing the ARQ mechanism, we had to take care of the fact that in a dynamic network new relays that are added during execution do not have the full log of perturbations, so they discard everything they receive because the `nextRef` does not match with what they expect. We solved this problem with a workaround: if the relay was added dynamically to the network, it will consider the first received perturbation as the first in the log, thus ignoring whatever happened before.

When a perturbation has to be delivered, the Relay will call the *abstract* `deliver` method, passing the content of the perturbation to the "application" layer. The "application" layer will thus inherit from the `Relay` class, providing an implementation for the `deliver` method.

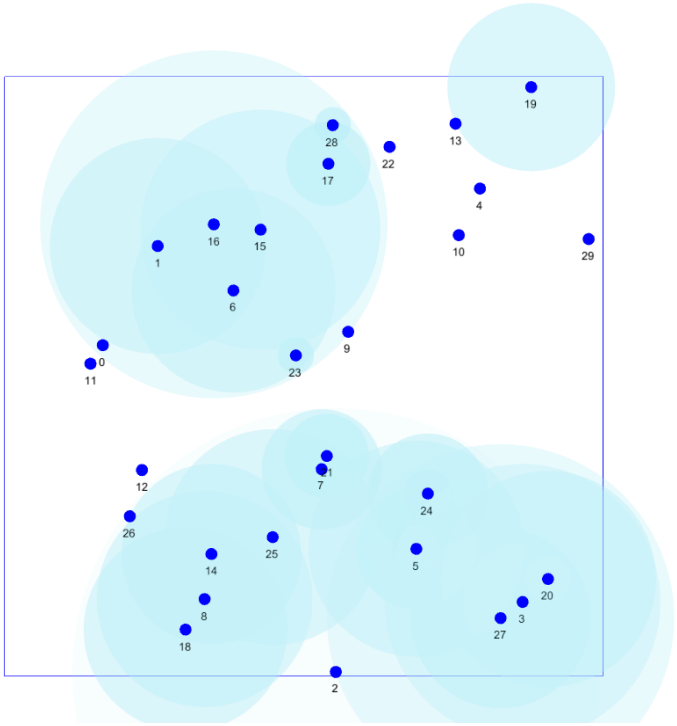

Figure 1: Graphical representation of the transport layer in a *Transitive-interest Gossip* simulation.

3.2 The "application" layer

The gossip algorithm is implemented as the application layer of the system, and it is based on the broadcast implementation defined in the `Relay` class.

We defined the behaviour of a participant through an agent named `Participant`, defined in a class that inherits from the `Relay` class.

For the broadcast implementation to work, relays already had an integer ID associated to them, but in order to implement the signature verification introduced in the paper, we also added an *Ed25519* key

pair associated to each participant. The private key is of course private, while the public key is assumed to be known by all the participants.

Each participant keeps track of a `Store`, a data structure defined as an `HashMap` between public keys and `Logs`. A `Log` is instead defined as a `LinkedList` of `Events`.

The gossip algorithm consists in exchanging stores periodically (every 200 ticks) through broadcasts, and updating the local stores accordingly.

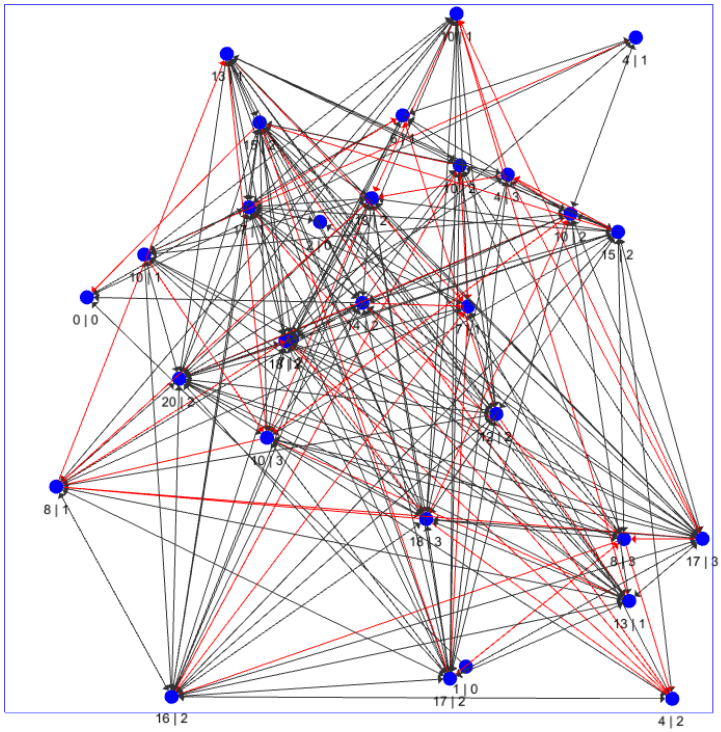

Figure 2: Graphical representation of the application layer in a *Transitive-interest Gossip* simulation.

3.3 The gossip algorithm

The `Participant` agent implements the abstract method `deliver`, where the `Store` carried by the perturbation is received from the layer below. The behaviour then differs depending on whether *Open Gossip* or *Transitive-interest Gossip* is being tested. This is defined through the Repast runtime parameter *Gossip Type*.

In the case of *Open Gossip*, the handling of the incoming store is very easy: we want to gather new events from all the logs contained in the received store, so we simply create the missing logs and then update them. The `update` method is implemented very similarly to how it is presented in the paper: the local frontier is obtained in order to understand which events are new in the received store and then the local logs are updated.

In the *transitive-interest* case, the implementation is much more complex. The code can be found in the `transitiveInterestGossipDeliver` method in the `Participant` class, and it has been adjusted from the paper to actually work in a real simulation environment.

To have a better understanding of the algorithm, let us assume that the participant A has received the store of the participant B. First of all, if A does not follow B, only the local logs of the known participants are updated, skipping entirely the transitive follows and blocks computation.

On the contrary, if A is following B, we look for transitive *follows* and add the logs of B's friends to A's store by creating the their logs, which will be then populated. For the transitive *blocks*, we take the set of the participant blocked by B and remove from it all the participants directly followed by A (we do not want to block our friend). Moreover, we remove from this set the participant A, otherwise a participant could block itself and remove its own log.

Another thing to note is that the pseudocode presented in the paper appears to take into consideration the local store when trying to find the transitive *follows* and *blocks*. This means that if the incoming store is more up-to-date than the local one (e.g. it contains follows that we still do not know about), we would keep it into consideration only in the next round, after the new events have been inserted in the local logs. An easy improvement would be to consider the incoming store instead of the local one to calculate transitive follows/blocks, in order to increase how fast consistency is achieved.

3.4 Friendships

We informally define a friendship as the fact that a participant follows another participant. This relationship is unidirectional, therefore not necessarily mutual.

In our implementation, follow relationships are managed in two ways:

- on startup, by reading an input file containing friendships data;
- during execution through a scheduled method that periodically follows/unfollows/blocks/unblocks participants randomly.

3.4.1 Loading friends through an input file

By default the simulator reads friendship data from the file `input1.txt` placed in the `data/` directory (configurable through a parameter).

This file must be structured such as each line contains the couple ID_i ID_j, to indicate that participant i follows participant j. Remember that the relationship is defined as unidirectional, and so it needs to be defined in both ways to have participants mutually following each other.

*ID*s in the input file are integer numbers and do not need to be consecutive, because they are remapped at runtime when loading the file. Depending on the number of participants set through the Repast parameter *Participants Count*, only a subset of N participants will be read from the file, together with all the participants they follow (and that are contained in this subset).

The `input1.txt` file that we provide as an example is based on real data, coming from a dataset of social circles extracted from Facebook in 2012.[1]

The relevant code that reads data from the file is found in the `DSProjectBuilder` class, in the `readFriendsFromFile` method.

3.4.2 Dynamic friendships

Since the simulation is dynamic (participants can join and leave randomly during the execution), we also introduced a way to alter friendships after the simulation is started.

We do this every 50 ticks, by probabilistically determining whether we should follow, unfollow, block or unblock a participant. The relevant code for this part is in the `manageFriends` method in the `Participant` class.

One thing we noticed while testing the system is that in some particular cases, for example if the simulation is initialized without specifying a friendships file, there are not enough follow relationships in the network to make the gossip and dynamic friendship algorithms work.

The problem rises from the fact that a participant initially has no logs in the store, and because of how the transitive-interest algorithm works logs are not replicated if there is no interest towards other participants. Therefore, in this situation a participant never replicates any log.

To solve the problem we could decide to remove the "follow" check in the *Transitive-Interest* gossip algorithm (the paper presents the algorithm in this way), but doing so we would break the transitive-interest concept and undermine its benefits: every participant would replicate the logs of friends of friends, which could potentially lead to open gossip, which is what we want to avoid.

In our simulation we therefore assume that in the case of *Transitive-Interest* gossip the network is always initialized with enough follow relationships (via input file or with a first phase of *Open Gossip*, in order to replicate some logs).

3.5 Events generation

Participants generate new events every 5 ticks with a parametrized probability. The content of events is fixed, since it is not relevant for the evaluation of the implementation.

After an event is generated, the store is immediately broadcasted to let others know about the updates.

In addition to the content, each event contains the following fields:

- *creatorId*: the public key of the participant that generated the event;
- *index*: an integer that is incremented within each participant when a new event is generated;
- *type*: the type of the event, i.e. message, follow, unfollow, block, unblock;
- *previous*: the hash of the previous event in the log. In our implementation we use the Java `hashCode` method;

[1] `https://snap.stanford.edu/data/egonets-Facebook.html`

- *signature*: the base64-encoded *Ed25519* signature based on the *creatorId*, *previous*, *index*, and *content* fields. The signature is validated before adding a new event to a log, so that no one can alter logs unlawfully.

3.6 Runtime parameters

The following parameters can be tuned at runtime to control the simulation:

- *Participants Count*: the number of participants to be added to the context when initializing the simulation;
- *Friends Data File*: the path where the friends data file is located, e.g. *data/input1.txt*. The amount of data read from the file is affected by the *Participants Count* parameter;
- *Topology*: the way participants are placed when initializing the simulation, e.g. *random*, *ring*, *star*, *star+*;
- *Spawn probability*: the probability of creating a new participant at runtime, every 10 ticks. This parameter is used by the participant manager and it affects the simulation only if the *random* topology is selected;
- *Kill probability*: the probability of removing a participant at runtime, every 10 ticks. This parameter is used by the participant manager and it affects the simulation only if the random topology is selected;
- *Perturbation Spread Speed*: defines how much the perturbation circle should grow at each tick, in units;
- *Event Probability*: the probability of creating a new message event;
- *Gossip Type*: determines the gossip algorithm to be executed: *OpenGossip* or *TransitiveInterest-Gossip*;
- *Follow Probability*: the probability of following a random participant every 50 ticks in transitive-interest;
- *Unfollow Probability*: the probability of unfollowing a followed participant every 50 ticks in transitive-interest;
- *Block Probability*: the probability of blocking an unfollowed participant every 50 ticks in transitive-interest;
- *Unblock Probability*: the probability of unblocking a blocked participant every 50 ticks in transitive-interest;

The type of gossip can be changed during the simulation, so it is possible to start the simulation with *Open Gossip* and then switch to *Transitive-Interest Gossip*. The follow/ unfollow/block/unblock probability parameters can still be set at the beginning of the simulation but they will be considered only when the type of gossip is switched to *Transitive-Interest*.

3.7 Displays

We defined two Repast displays to provide a graphical representation of the simulation.

The first display is named *Transport* and it represents relays and perturbations in the transport layer. Relays are labelled with their ID and represented as small circles; perturbations are self-expanding circles that fade as their radius increases.

The separation of the transport layer display allowed us to define a more complex application layer display using a Repast *Network*. In the *Application* display participants are represented as nodes and labelled with the number of followed and blocked participants.

The participants they follow or block are drawn as coloured edges using the network. Since "follows" or "blocks" are the only possible relationships between participants, we decided to give them two different colours: **dark gray** for follow relationships and **red** for block relationships. As mentioned, since the relationships in this model are unidirectional, the edges are directed.

During the creation of this display, we experienced a bug in Repast: the "tick rate" would drop rapidly during execution, until the simulation would stop entirely with an out of memory exception. We found

out that the problem derived from the `sensePerturbation` method in the `Relay` class: the method was being scheduled using a *Watcher*, a Repast feature that allows to query a particular field of an agent (in our case, the `Perturbation.radius`).

For some reason, the watch feature does not work well with Repast Networks, so we switched to a less idiomatic implementation where each relay collects at every tick all its surrounding agents (within the maximum perturbation radius), filters them by agent type (since we are only interested in perturbations) and computes the distance to each perturbation. This implementation did not show performance issues even with the network.

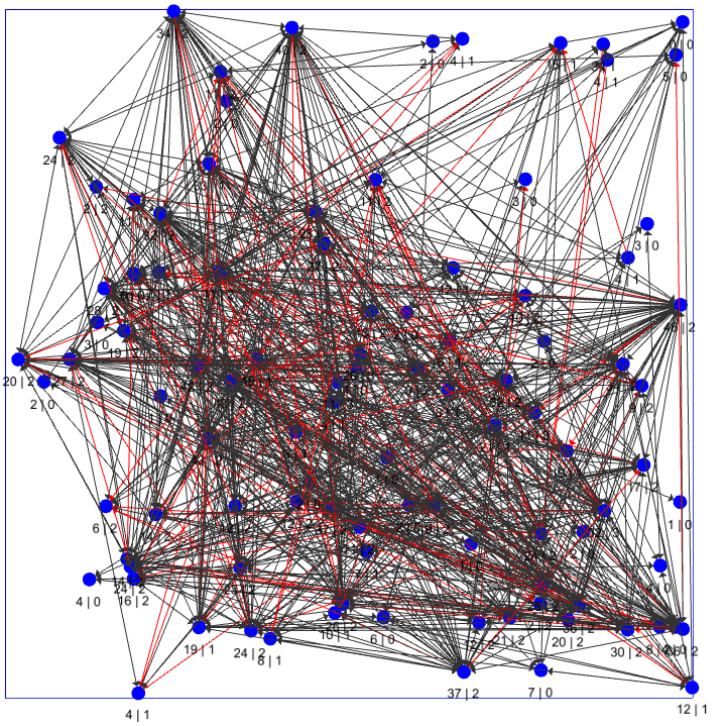

Figure 3: Application layer display with a network of 100 nodes. Edges in black represent follow relationships, edges in red represent block relationships.

4 Analysis

In this section, we present how we analysed the results of our simulations and explain the charts and tables that we used to evaluate the model.

4.1 Participants Count

The first chart represents the number of participants in the simulation at each tick. The number of participants depends on the probabilities explained in the parameters sections and can be changed at runtime, therefore affecting the behaviour of the simulation.

Since participants and relays are the same objects, the number of participants also represents the number of relays in the transport layer.

4.2 Known Participants Count

This chart represents the maximum, minimum and average number of participants known by each participant in the system. This number is computed by counting the number of logs present in the store of each participant minus one, to avoid counting the participant itself.

To analyse this chart, we need to distinguish the two types of gossip: with the *Open Gossip* algorithm, at some point each participant will know every other participant. This is because on each gossip update the whole set of logs in the received store is merged with the local store, therefore each participant will get to know every other participant rapidly (in Figure 4 around tick 200).

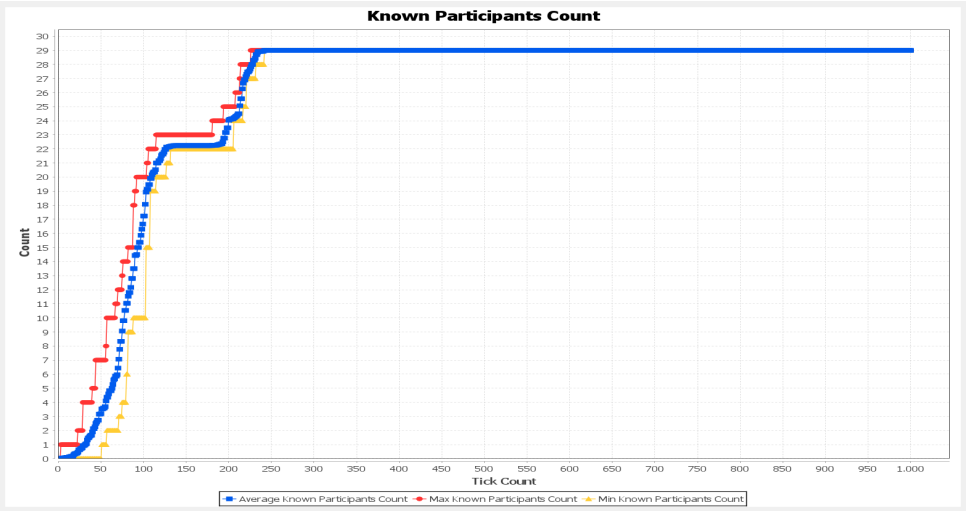

Figure 4: Known participants after 1000 ticks in an *Open Gossip* simulation with event, follow, and unfollow probability of 0.05.

On the other hand, with the *Transitive-Interest Gossip* we cannot expect everyone to know everyone because of the block feature, which has the effect of removing logs from the local store.

In the first part of a *Transitive-Interest Gossip* simulation initialized with some input file, the participant with the highest number of follows will represent the maximum number of known participants, while the one with the lowest follows will represent the minimum. As mentioned above, if the simulation is started using *Transitive-Interest Gossip* without initialization (i.e. with no input file or in *Open Gossip*), participants will not be able to follow other participants, and so its participants counter will remain to 0 (Figure 5, yellow line).

Moreover, it can be noticed that in this model some participants start with a number of known participants that is already greater than 0: this is due to the fact that simulation is initialized through the friendships file.

Another consideration that can be drawn is the fact that a higher event probability leads to a steeper increase of the number of known participants.

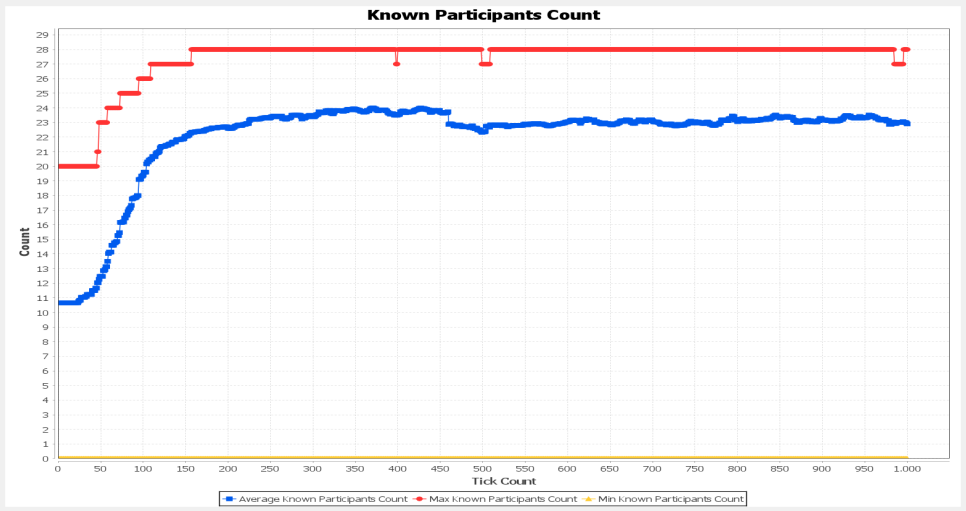

Figure 5: Known participants at tick 1000 in an *Open Gossip* simulation with event probability of 0.05.

In both protocols, when a new participant is spawned the minimum number of known participants will drop to 0, since the store of the new participant is initially empty.

4.3 Friends Count

Every 50 ticks, each participant can probabilistically decide to follow or block another participant. The total number of these relationships is represented in the *Friends Count* chart, where the gray line repre-

sents the number of total follow relationships while the red line represents the number of total blocked participants.

Figure 6 shows a comparison between the number of followed/blocked participants and known participants. In this case, the simulation was run for 100 ticks with *Open Gossip*, then the type of gossip was switched to *Transitive-Interest* and the simulation went on for another 900 ticks. In this way, participants quickly got to know everyone in the first part of simulation, while in the second part they started to follow, unfollow, etc. other participants.

In the known participants count chart, we can also notice that the number of known participants starts to decrease after tick 100 because participants start being blocked and thus their logs get removed from stores.

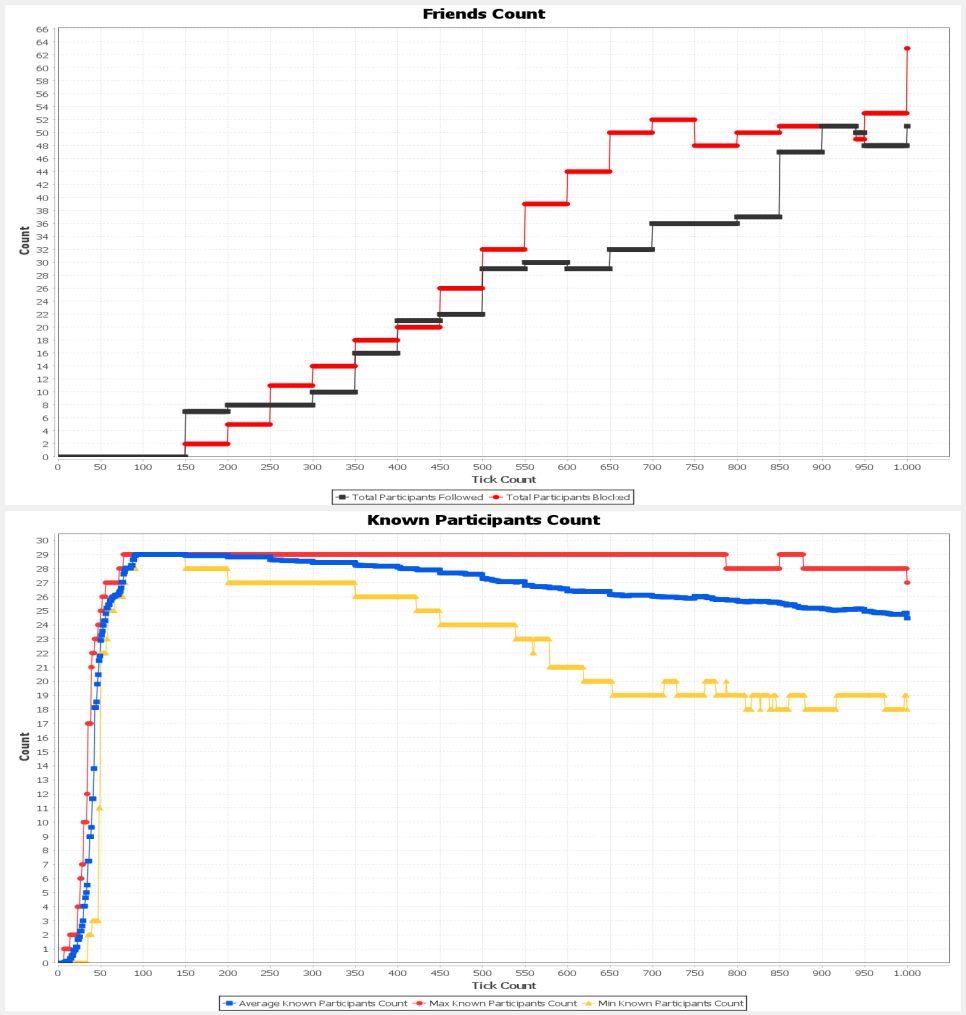

Figure 6: Comparison between the number of followed/blocked participants and the known participants in a *Transitive-Interest* simulation. The event probability was set to 0.2 and for the first 100 ticks the gossip type was *Open Gossip*, then switched to *Transitive-Interest*.

4.4 Total Events Count

The number of total events is computed by each participant by summing the number of events of each log in the local store. The chart then represents the maximum, minimum and average value of these counts.

Also in this case we need to distinguish between the two gossip algorithms. When using *Open Gossip*, we can observe that the three values shown in the chart will have approximately the same value, which can be computed through the following formula:

$$v(t_i) = \frac{n(t_i) \cdot (n(t_i) \cdot p\_event \cdot \frac{t_i}{5})}{n(t_i)} = n(t_i) \cdot p\_event \cdot \frac{t_i}{5} \tag{1}$$

where n is the number of participants in the system at tick i and p_event is the probability of generating an event (once every 5 ticks).

At the numerator, we calculate that each participant keeps a copy of all the events generated by other participants: at every tick, each participant will generate on average $p\_event \cdot \frac{t_i}{5}$ events. Since the formula computes an average, we have the factor $n(t_i)$ at the denominator, which can then be simplified resulting in the simpler, final formula. Figure 7 and Table 1 (found in the appendix) confirm the result.

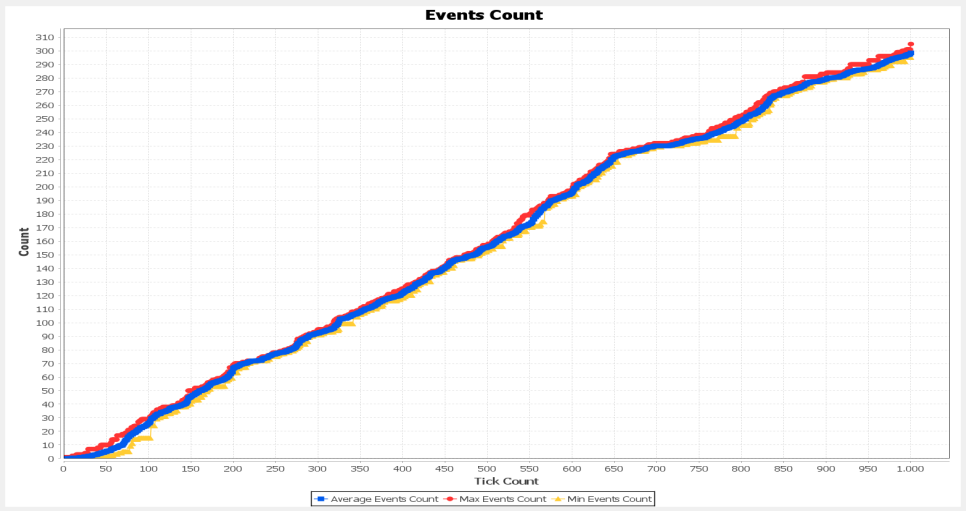

Figure 7: Events count chart in an *Open Gossip* model with 30 participants and event probability set to 0.05. The expected number of events at tick 1000 is 300, which is roughly the observed value.

For the *Transitive-Interest Gossip*, an analytical model would be much more complex to obtain, since we would need to take into account the fact that each participant could block different other participants, therefore removing their logs from the local store. The *Transitive-Interest Gossip* behaviour can be observed in Figure 8.

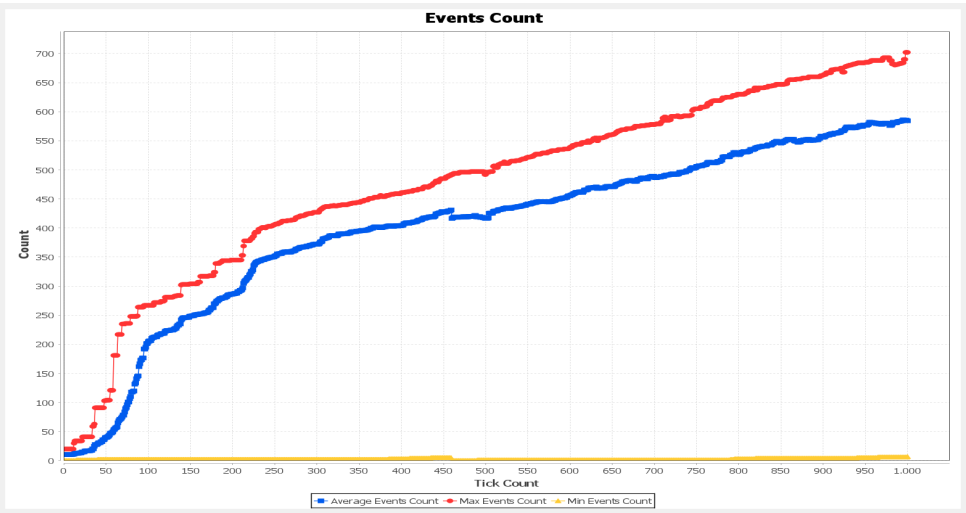

Figure 8: Events count chart in a *Transitive-Interest Gossip* model with 30 participants and event probability set to 0.05.

The behaviour of the system after spawning new participants can be observed in figure 9, where we can see a drop to 0 of the minimum number of events, represented by a participant with empty logs. The drop is quickly recovered because after the first received perturbation the store is updated with all the events.

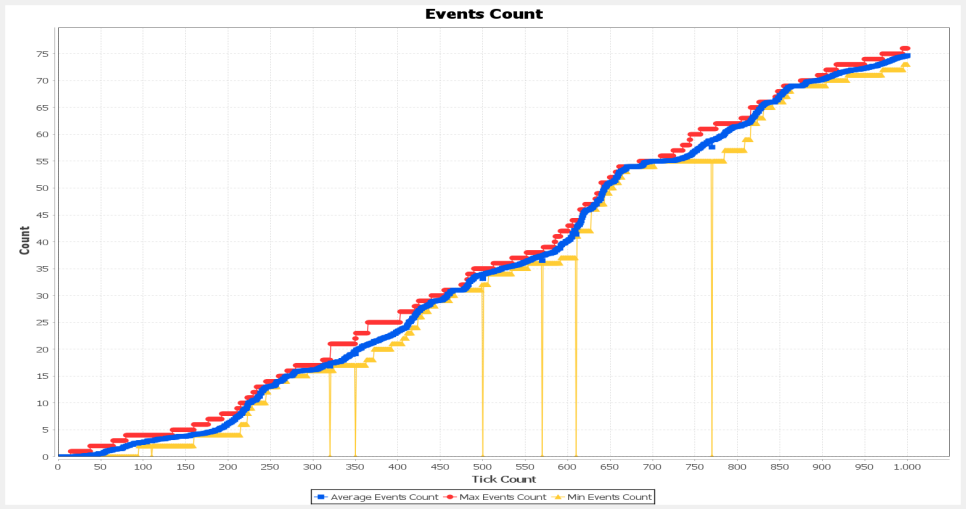

Figure 9: Events count chart in an *Open Gossip* model with 30 participants and spawn probability set to 0.2.

4.5 New Updates Count

This chart represents the average, maximum and minimum number of events applied by each participant at every tick.

In the *Open Gossip* model, as we can notice in Figure 10, the highest peak of new updates can be observed at the beginning of the simulation, when the participants get to know each other.

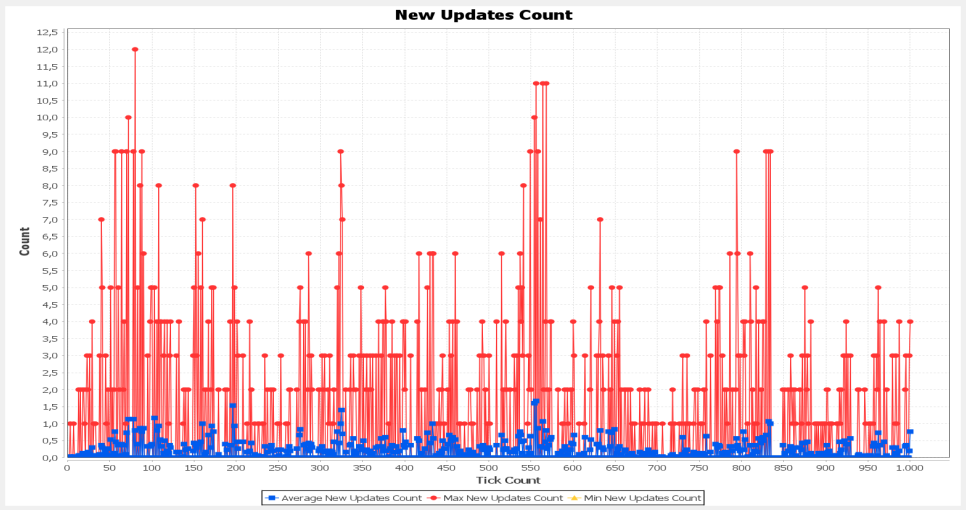

Figure 10: New updates chart in an *Open Gossip* model with 30 participants and spawn probability set to 0.05.

In the *Transitive-Interest Gossip* model, the highest peak is reached again at the beginning of the simulation (Figure 11), but in this case it reaches a much greater value. This is due to the fact that the simulation has been initialized with the friends data file, therefore many follow events need to be propagated to many participants (which can also be seen in the number of events in Table 2).

For the same reason, also the other peaks in the chart are larger than the ones of the *Open Gossip* model.

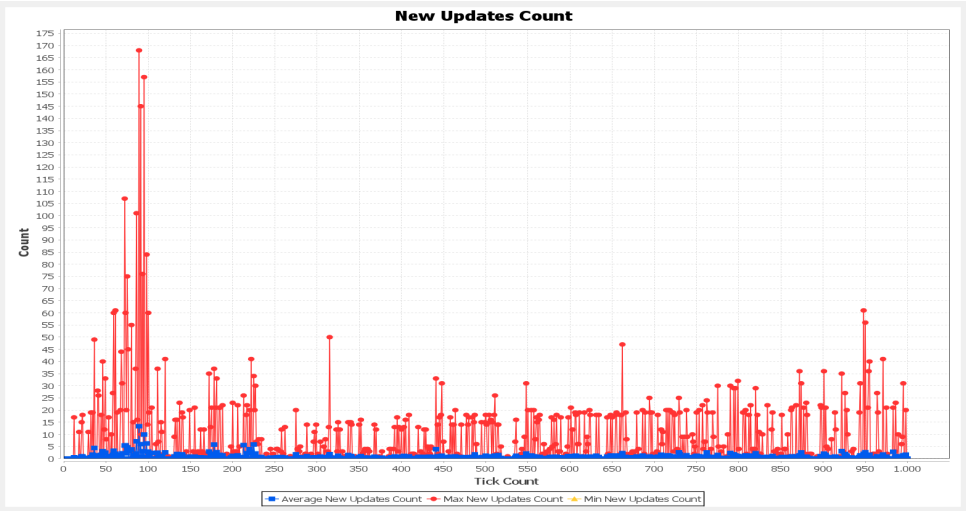

Figure 11: New updates chart in an *Transitive-Interest Gossip* model with 30 participants and spawn probability set to 0.05.

5 Conclusions

Our implementation shows that the proposed models can be easily implemented in practice, even though they need some adjustments with respect to how they are presented in the paper.

One of the main issues of our implementation is that it generates a lot of network activity. Since with time the stores can become quite large, exchanging all the events so often is not very efficient. This could be improved by instead exchanging only frontiers, so that other participants know which information they are missing and then request only that information. This variant of the protocol would be much more efficient in presence of very big stores, because the number of events sent at each gossip broadcast would be much lower.

To simplify the implementation, in our project we introduced some level of centralization, for example by having a `ParticipantsManager` to spawn and kill participants and by providing the public keys of participants in a centralized way. To make the implementation more "distributed", a possible improvement could be to introduce a *Discovery* mechanism, so that new participants introduce themselves to others by exchanging their public keys.

One further thing we noticed is that with the *Transitive-Interest* algorithm, stores contain a significant number of follow/unfollow/block/unblock events that need to be replicated and frequently scanned to find friendships. A proposal to improve this would be to keep the state of the relationships not through events but in another data structure that could be replicated more efficiently.

Finally, as mentioned above, if no friends data is provided no participants will be able to create follow or block relationships because they do not know any other participant. A possible solution with respect to the simulation would be to initially create some random relationships between participants, in order to automatically initialize the network. In real social applications, however, this is not necessarily a problem if we assume that participants can manually find and follow other people to initialize the network.

References

[1] C. Tschudin, A.M. Kermarrec, and E. Lavoie. *Gossiping with Append-Only Logs in Secure-Scuttlebutt.* URL: `http://ericklavoie.com/pubs/dicg2020.pdf`.

A Participants Tables

ID	Public Key	Known Participants	Total Events
0	styurMg9W+	29	59
1	VI2LZbRIXo	29	60
2	9EaImzuOa+	29	59
3	db4sf7ajon	29	58
4	JnAShKvFGr	29	60
5	w1s4C1PHWw	29	60
6	5WVQs6BKCF	29	61
7	pQE2pXuUqX	29	59
8	iTspGcBu3m	29	60
9	+oIx1w+a8N	29	60
10	HraPKJ/c5/	29	59
11	CckG0l5KKv	29	60
12	PNIS40G+ta	29	59
13	B8USKG6gz6	29	59
14	8UNoKoghmL	29	59
15	LmWZ19P+sj	29	60
16	Vu2k5oMOPx	29	59
17	5aYJM9Em5b	29	60
18	III7bvr7fs	29	58
19	/PM/NvXDhb	29	59
20	2bQcM5HGYR	29	61
21	q/Lc+ThJwk	29	61
22	eMV5z+oBPY	29	59
23	qm/9+KoSba	29	59
24	kf4/XTySmN	29	59
25	kagoFYJKEg	29	60
26	ifuTKgMXxO	29	61
27	0E7RpvG5p8	29	59
28	U33/UFZl8I	29	60
29	aew+cdHVbU	29	59

Table 1: Table representing 30 participants during an *Open Gossip* simulation. The data was taken at tick 1000, while the simulation was running with event probability set to 0.05. It can be noticed that up to this point each participant knows all the other participants and that the average of total events is the expected value that we could have obtained using Formula 1.

ID	Public Key	Known Participants	Total Events	Followed	Blocked
0	GqPoeeQQ37	28	383	14	0
1	X3a9RboyeO	28	386	12	0
2	xNGw0NX+6Q	28	386	17	0
3	+GU1/QgwBh	28	388	14	0
4	ySrgqIZAfF	1	5	1	0
5	XLKtWqdDoe	0	3	0	0
6	2YVVZ6M3sX	27	380	20	0
7	DLhUxI3ASn	26	371	16	1
8	V1cYNaztg4	23	360	5	0
9	YCng6LCwfP	28	384	15	0
10	VjJisAdXB2	23	348	6	0
11	+lY42Ag5aR	26	379	7	0
12	VMTEr6ZT/B	17	253	4	0
13	Q7Riu8y0DF	27	374	11	0
14	+MsvcblH3K	26	371	17	0
15	XJd+HX/cxC	27	385	10	0
16	3JGcUUjWkV	24	353	7	0
17	xvgDtAFKbL	19	306	2	0
18	0OJI1TWuzE	25	374	10	0
19	EBffnOWjh3	26	377	18	0
20	0q1cQVVS5x	27	372	13	0
21	Lb0PMrEB5J	27	380	15	1
22	yEXs+PsxAl	25	358	8	0
23	kF3c4F0jUA	26	375	12	1
24	VWO2/+xs4k	22	338	4	0
25	h0ijHJfkoT	24	360	8	1
26	hsK7s5mgwW	27	380	10	0
27	RehnACtfI1	26	371	16	0
28	aJoUq1l3Ci	27	381	9	0
29	NyMeN6aMUh	28	385	18	0

Table 2: Table representing 30 participants during a *Transitive-Interest Gossip* simulation. The data was taken at tick 1000, while the simulation was running with event probability set to 0.05. Differently from the *Open Gossip* model, there are some participants who do not know many other participants: these participants will have a very small total events count, because they will not replicate any other store. It can be noticed that the number of total events of participants that follow many other participants is very high when compared to the *Open Gossip* model.