Gossiping append only logs

Simone Degiacomi
211458
simone.degiacomi@studenti.unitn.it

Davide Tessarolo
211457
davide.tessarolo@studenti.unitn.it

Davide Zanella
211463
davide.zanella-1@studenti.unitn.it

I. Abstract

In this report we describe how we implemented the concepts presented in the "Gossiping with Append-Only Logs in Secure-Scuttlebutt" paper [1] assigned for this task, and an analysis of its performance depending on different configurations and scenarios.

II. Implementation

This project has been implemented using the Repast Simulation Framework, with the Java programming language. With these technologies it is easy to design and implement an actor based system to perform the simulation. Repast, as contrary to Akka, includes the concept of positions of actor in a space. We implemented two actors: the LAN actor, which does not move and represents the local area network of a point of interest visited by people, and Person, which represents a person moving around. A LAN has a radius property, and people can communicate between each other when they are inside the reception of the same LAN. Each person has a Store, which is the collection of Logs known by that person, mapped by the people public key. Each Log is an ordered collection of Events.

A. LAN Actor

The purpose of this actor is to represent a physical space where people can communicate together, so the implementation is really simple: it only has two attributes and a method. The first attribute is the station ID, and the second is the reception range, that we fixed at 5 space unit. The only method that it offer is *getConnectedPeople()*, which returns the Set of people currently inside the reception range.

B. Person Actor

Person is the actor that moves around the space and synchronizes Logs with other people. The original paper mentioned two synchronization algorithms, but did not specify how people connects to each other, to which people they connect and how frequently they generate events. To easily change those algorithms we created three interfaces to encapsulate the algorithms, using the Strategy Pattern [2]: SynchronizationStrategy, MotionStrategy and EventGenerationStrategy.

C. Events

An Event is a single interaction of a person with what we could call a "distributed social network". There are different types of event, all extending the abstract class Event:

- **StreamEvent**: is the only event that can be created in the Open Gossip Model and can be seen like a content sharing event, like a post on a social media. It has no further purpose besides being generated and shared among the actors in the simulation.
- **FollowEvent/UnfollowEvent**: an event that uses an Id to record an active interest in the person with that id. If interest in said person is lost, the FollowEvent can be cancelled by using an UnfollowEvent. Note that cancelling doesn't mean that the FollowEvent gets actively cancelled from the logs, but instead each pair of Follow/Unfollow events disable each other out. Only present in the Transitive-Interest Gossip model.
- **BlockEvent/UnblockEvent**: an event that uses an Id to record an active disinterest in id which, similarly to the FollowEvent, can be then cancelled by using an UnblockEvent. A BlockEvent makes it so that the store of a blocked id doesn't get synchronized with mine.

Events are generated by one implementation of the EventGenerationStrategy.

D. Motion

Since the simulation requires the Person actors to move around the grid, in order to enter LANs and exchange events, a motion strategy needs to be defined. In this project two motion strategies are available: RandomMotion and HabitMotion. The first is rather trivial and just defines a random point for each person: once the the person reaches that point, another one is generated and so on. This creates a simple system that however is not representative of a real-world scenario: usually LANs are in fact located in points of interests, like universities, restaurants, libraries and so on. While in the RandomMotion strategy people will only pass through these points of interest, it would make more sense for them to visit them and remain for an (even short) amount of time. This is why the HabitMotion strategy has been created: it defines the habit of each person, defining a home and different points of interest for each person. During the simulation people will travel from their homes to their favourite LANs and vice versa, making it a more solid motion model.

Describing more precisely the HabitMotion strategy, every person has its own characteristic: a slightly different number of preferred points where it is used to go, and an amount of time it usually spends in that place. To assign this values two Gaussian distributions were used, in this way, sampling from the distributions, we can be sure that the assigned values are

slightly different between each other and not always equal. Once the preferred LANs and the waiting time values are assigned, the person starts to move from its home to a preferred LAN sampled randomly. It will stay there for the waiting time plus a random tick value between zero and three, to make the simulation more realistic. After the waiting time is passed, the person goes home and, once again, waits there for the waiting time plus a random value. From this point, the logic just explained is continuously repeated: the person moves from home to a preferred LAN, waits some time and then returns home. Note also that, using this strategy, home points of Person actors and LANs can overlap: while this could seem wrong at first, it makes the simulation even closer to a real world scenario, considering how some people live in/near points of interests with unprotected LANs, or how sometimes public LANs are available.

E. Synchronization

When two people are connected (because they are in the same LAN) their stores get synchronized. Synchronization is managed by the SynchronizationStrategy interface, which has two implementations: OpenSynchronizationStrategy, which implements the open synchronization algorithm, and TransitiveInterestSynchronizationStrategy, which implement the transitive interest algorithm. Every person in a LAN will synchronize with every other person in the same LAN. During a synchronization the person that run the synchronization is called ”local” and the other person ”remote”.

When OpenSynchronizationStrategy is used, the local person will download all available logs from remote. Local will then create unknown logs or update existing ones (if needed).

When TransitiveInterestSynchronizationStrategy is used, local will compute three sets:

- friends: the set of ids followed by local
- friendsOfRemote: the set of ids followed by the remote person
- blocked: the set of ids blocked by local

Then, it will compute the set of ids to synchronize, which is defined by:

$$\{remote.id\} \bigcup friends \bigcup friendsOfRemote \setminus blocked$$

Finally it will synchronize the set of computed ids, using the same update logic used by OpenSynchronizationStrategy, which is implemented in the Store class and follow the algorithm described in the original paper.

In our model we did not limit the number of events that can be synchronized in a single tick: this comes from the assumption that the speed of LANs is usually high, both in download and upload speed, and that Stream events are generally lightweight, since they are composed only of plain text.

F. Log collection and data analysis

To efficiently run many simulations and summarize the results into graphs, we used the batch run feature of Repast together with Python scripts. Once all the logs were collected, three different Python scripts were used to draw the plots:

- **countEvent**: plots the graph of the type of events created during a single simulation run: it is thus possible to see how and when the different types of events (in the case of a TransitiveInterest simulation) are created during the course of a simulation.
- **diffusion**: plots the diffusion of an event during the simulation. This means that the script takes as input the log of a single run, then proceeds, for each event, to plot the number of people that received that event. The result allows us to see how an event is shared during the course of the simulation, for each one of the events created.
- **latency**: either plots the latency for each events of a single simulation run (measured in ticks) or append its value to a csv file to be used with the next script. Latency is calculated by subtracting the mean arrival time of each event to that event's creation time.
- **3dlatency:** plots a 3D graph that takes as input the csv file created with the previous script. This generates a graph on three axis, for example: number of LANs, number of persons and latency, allowing us to see how the latency changes with respect to the LANs and persons on the simulation.

III. Running the simulation

From the Eclipse IDE, it is possible to run the model, which will open the windows in Fig. 1.

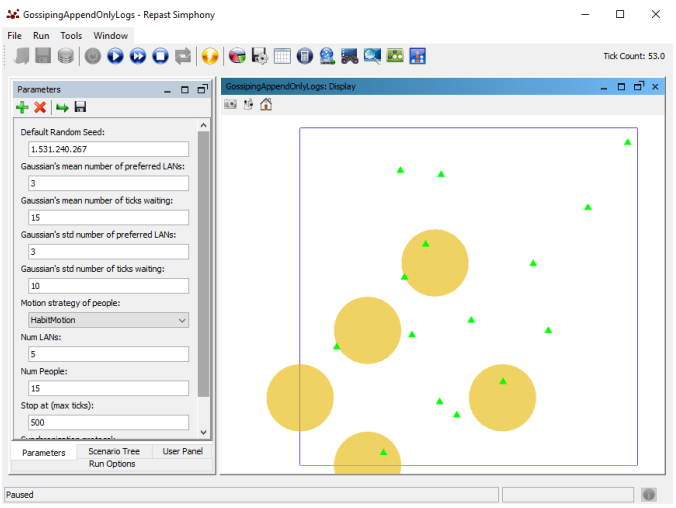

Fig. 1. GUI for the simulation.

There are many parameters which allow the simulation run to be customized, namely:

- **Num LANs**: the number of LANs that will be present on the simulation. This is a fixed parameter that cannot vary during the run of the simulation.
- **Num People**: the number of Person actors in the simulation. Similarly to the previous parameter, this number cannot change during the simulation run.

- **Motion Strategy**: the different motion strategies are described in paragraph II-D and define the way the Person actors move in the simulation's grid.
- **Synchronization Protocol**: the synchronization protocol defines how events are exchange between people. The two different ways of synchronizing event are explained in detail in paragraph II-E.
- **Gaussian mean preferred LANs**: used in the Habit Motion strategy, indicates the mean number of favourite LANs that a person has.
- **Gaussian std preferred LANs**: used in the Habit Motion strategy, indicates the standard deviation number of favourite LANs that a person has.
- **Gaussian mean ticks waiting**: used in the Habit Motion strategy, indicates the mean number of ticks a person will stay in one of his favourite LANs.
- **Gaussian mean ticks waiting at home**: used in the Habit Motion strategy, indicates the mean number of ticks a person will stay at home.
- **Gaussian std ticks waiting**: used in the Habit Motion strategy, indicates the standard deviation number of ticks a person will stay in one of his favourite LANs.
- **Gaussian std ticks waiting at home**: used in the Habit Motion strategy, indicates the standard deviation number of ticks a person will stay at home.

In the simulation we can see the two actors depicted in section II: the LANs are shown as big yellow circles, while the Person actors are the green triangles.

IV. Results

The result section will gather all the data generated through the simulation that has been later analyzed through the scripts. The results have been grouped in different subsections for readability. Each unique configuration was run 20 times, and all the results come from the average of those runs.

A. Event creation

The logs produced via batch runs where analyzed via the use of the scripts described in paragraph II-F. Starting from the countEvent script, we can see how and which events are created during the course of a simulation. In Fig. 2 we see how, in a simulation with default parameters (15 people) and a Open Model, the created events tends to be around 2 every 10 ticks. The initial spike in event creation is a normal behaviour caused by the creation of people and scheduling of their *generateEvent()* function all at the same time during simulation start. Some minor spikes can be seen even later, but these are only caused by the probabilistic nature of the event generation function.

The same simulation has been run also with the Transitive Interest Model: in this case the number of generated events is higher, due to the StreamEvents and the events related to the Transitive Interest Model having two separate probability distributions. In Fig. 3 we can see how Stream and Follow events dominate the event generation scene: this is caused

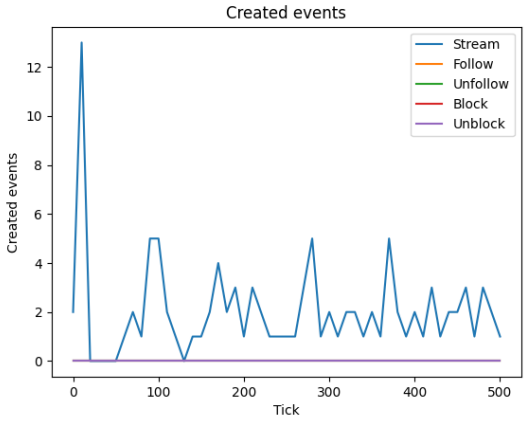

Fig. 2. Plotting the events created with default parameters. Open model. In order to have a cleaner view of the graph, the ticks have been grouped in groups of ten. Thus, every point in the graph refers to a period of 10 ticks.

by two factors: StreamEvents having a separate probability distribution for their generation and FollowEvents having the highest probability of being created among all Transitive Interest related events. The latter was shaped after the realistic assumption that, in a social network environment, the "follow" action is the most used, while the "unfollow" one is more rare. The "block" and "unblock" actions follow the same line of thought. Overall, the probability given to each one of the Transitive Interest related events is:

- Follow: 50%
- Unfollow: 30%
- Block: 10%
- Unblock: 10%

These probabilities plus the fact that "unfollow" and "unblock" actions can only be performed if the person is actively following/blocking another person, make "unblock" actions very rare altogether. Running a simulation for a longer amount of time (1000-1500 ticks) would help in the creation of "unblock" events, as it would give time for people to block others first.

B. Event diffusion

The diffusion script allows us to see how the event generated during the simulation are shared between the different people as time passes. Here we can see how events can initially take quite a lot of time to be initially shared: in the best case scenario, an event is created when the person is on a LAN, resulting in the event getting shared the next tick. The worst case scenario is instead the one where an event gets created right after a person has left a LAN in order to return home: this means that, in order to the event to be shared, the person will have to travel home, wait there and then travel back to a LAN where other people are waiting. We can see both cases in Fig. 4, where some events can take up to 150 ticks for them to be shared, while others are shared immediately after the creation.

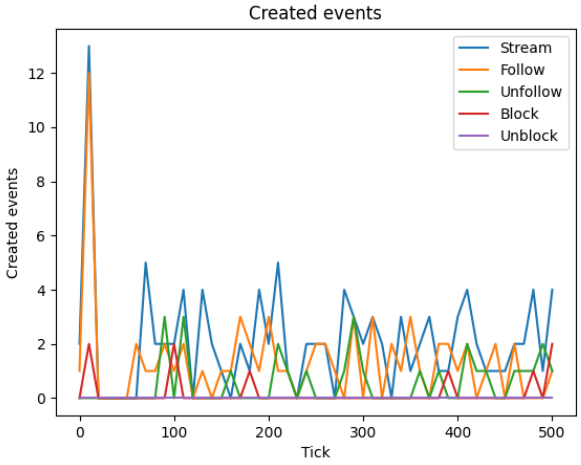

Fig. 3. Plotting the events created with default parameters. Transitive model. In order to have a cleaner view of the graph, the ticks have been grouped in groups of ten. Thus, every point in the graph refers to a period of 10 ticks.

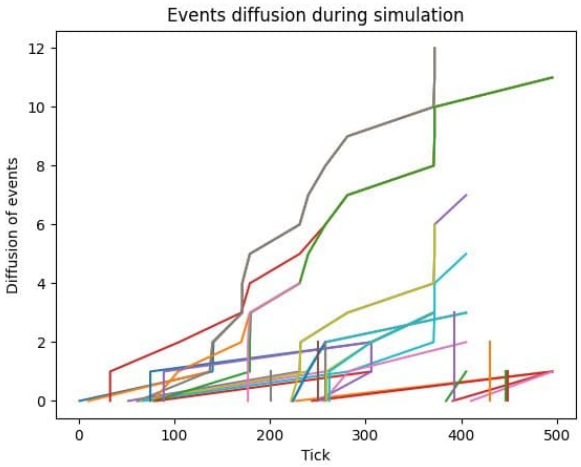

Fig. 4. Plotting the diffusion of the different events created. Open model. In order to have a cleaner view of the graph, the ticks have been grouped in groups of ten. Thus, every point in the graph refers to a period of 10 ticks. The probability of generating new events has been, in this case, reduced in order to allow an easier readability of the graph.

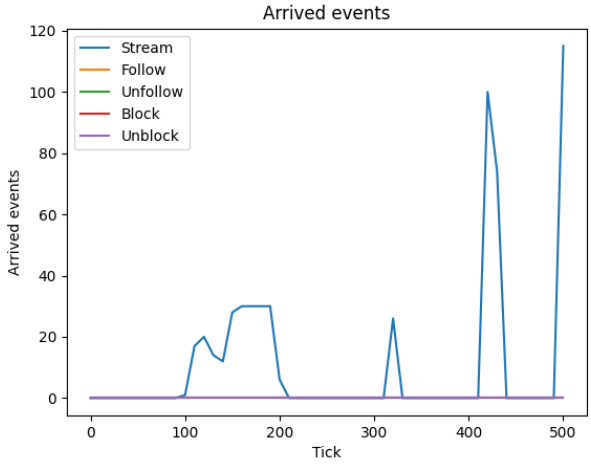

Fig. 5. Plot showing the number of delivered events throughout time for an *Open* model using the *Random motion* strategy.

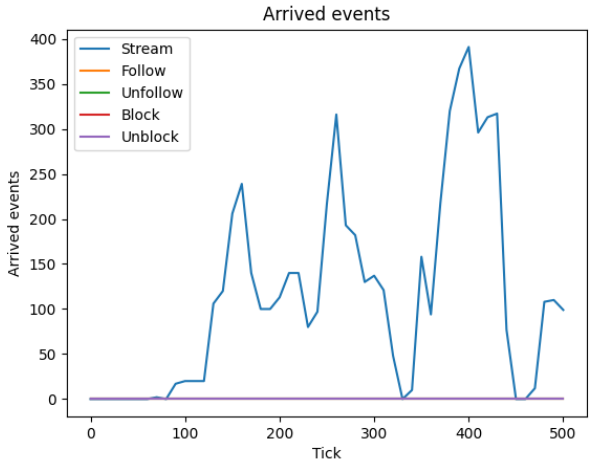

Fig. 6. Plot showing the number of delivered events throughout time for an *Open* model using the *Habit motion* strategy.

By looking at the graph we can see also how, after the initial share of an event, subsequent sharings for that event happen more frequently, but this doesn't allow an event to be shared with all people in the simulation (15 people, 500 ticks). Additional details on the diffusion of events will be provided in the following paragraphs, by also specific the type of motion strategy, synchronization strategy and parameters used.

C. Different motion strategies

In this section we want to analyse the difference between the two motion strategies implemented in our model. As previously explained, the random motion strategy lets the people travel in the space changing the destination point in the space randomly every time they has reached it. The habit motion strategy, instead, assigns some preferred LANs and a home point to every person so they will travel between them every time. To analyse their difference we can focus on Figure 5 for the Random motion and Figure 6 for the Habit motion strategy. Both the simulations were made using the same parameters in order to have some comparable results. The plots show how the number of events exchanged from the people changes throughout time. In the Random motion plot since people walk randomly they have less chances to arrive in a LAN and meet other people, so the number of exchanged events is always less. Looking at this results, the Habit motion strategy seems to perform better and to better represent a real situation, where people tends to go mainly in the same places. For this reason we decided to use only the Habit motion strategy for the next simulations.

D. The impact of the number of LANs, preferred LANS and waiting ticks

In this section we analyze the impact on latency obtained by changing the number of LANs, of preferred LANs and waiting ticks. To collect those results we run many simulations according to Table I, where with the syntax *(start, end, step)* we mean that we run simulations with the specified parameter varying from *start* to *end*, incrementing by *step*. We then used a three-dimension plot to show how varying the parameters impact the latency with different number of people.

1) The impact of the number of LANs: From Fig 7 it is immediately clear how increasing the number of LANs and people decreases the latency when using the Open Model. While this could seem an obvious result at first, and it is if only considering the increasing number of people, the initial expected behaviour with the increasing number of LANs was different: the expectations were that, by having too many LANs on the grid, different people would have a harder time meeting on a common LAN to exchange the events. While this concept is true (having too many LANs makes it less likely for two people to choose the same one as favourite), it also makes more likely for two people to meet under a random LAN while moving for reaching their objective. This behaviour is highlighted when watching the graph: increasing the people with a low number of LANs tends to result in lower latency values, but has a more irregular curve, while increasing the LANs tends to decrease the latency in a more consistent way. In the case of the Transitive Interest Model, by looking at Fig 8 we can see how the surface is more irregular, but the graph as a whole tends to follow the results of the one for the Open Model.

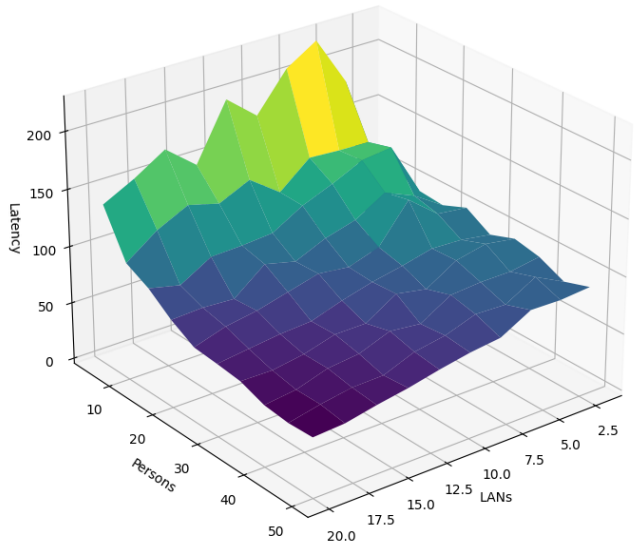

Fig. 7. 3D plotting for latency wrt number of persons and LANs in Open Model

2) The impact of the number of preferred LANs: By looking at Fig 9 (for the Open Model) and Fig 10 (for the Transitive Interest Model) we can see how changing the number of favourite LANs per person doesn't consistently raise or lower the latency values. This results in a graph whose behaviour is dominated by the number of people, allowing us to understand how the mean of favourite LANs doesn't affect the latency, especially in the Open Model.

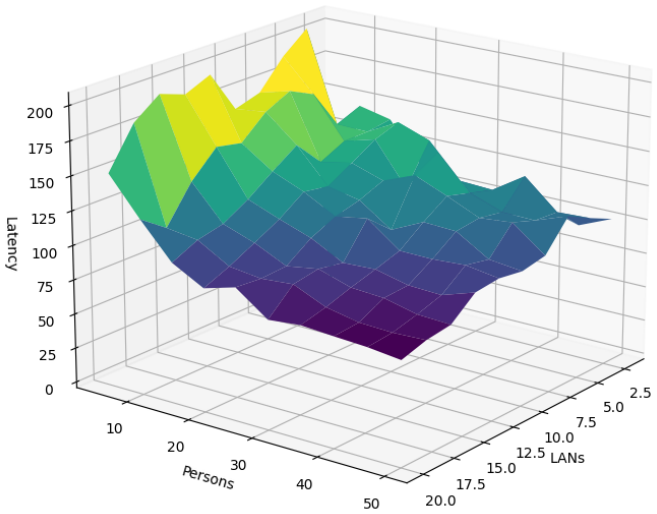

Fig. 8. 3D plotting for latency wrt number of persons and LANs in Transitive Interest Model

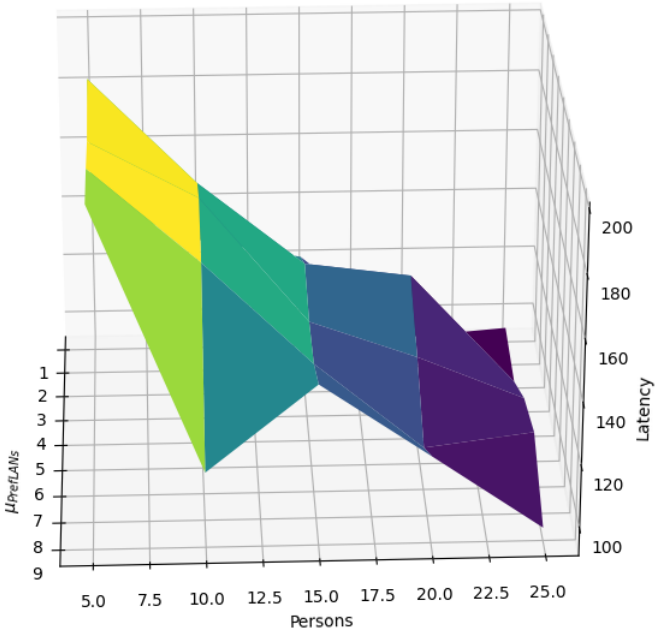

Fig. 9. 3D plotting for latency wrt number of persons and mean of favourite LANs in Open Model

3) The impact of the number of waiting ticks: Similarly to the number of LANs, the amount of time people are waiting in their favourite LANs is a parameter which requires some thought: on one hand, by increasing it we increase the probability of a person in a LAN meeting additional people

TABLE I

#	Num LANs	Num People	$\mu_{prefLANs}$	$\sigma_{prefLANs}$	μ_{wait}	σ_{wait}	$\mu_{waitHome}$	$\sigma_{waitHome}$
A	(2, 20, 2)	(5, 50, 5)	3	3	5	5	5	2
B	10	(5, 25, 5)	(1,10,2)	1	5	5	5	2
C	10	(5, 25, 5)	3	1	5	(1,10,2)	1	2

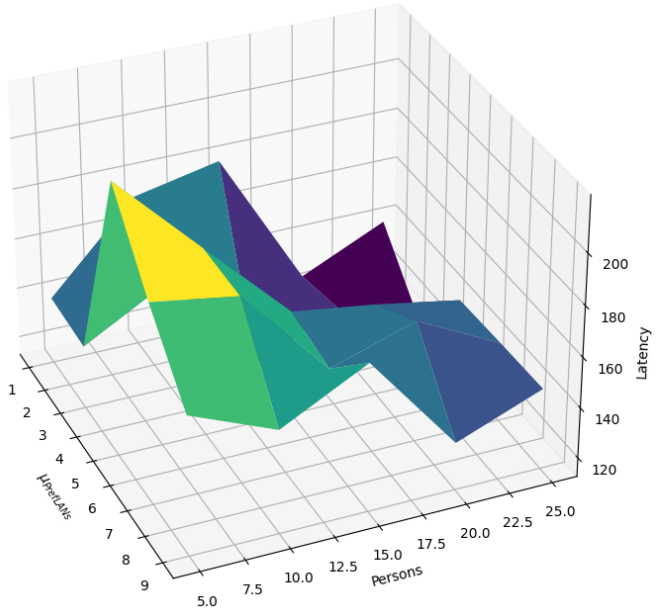

Fig. 10. 3D plotting for latency wrt number of persons and mean of favourite LANs in Transitive Interest Model

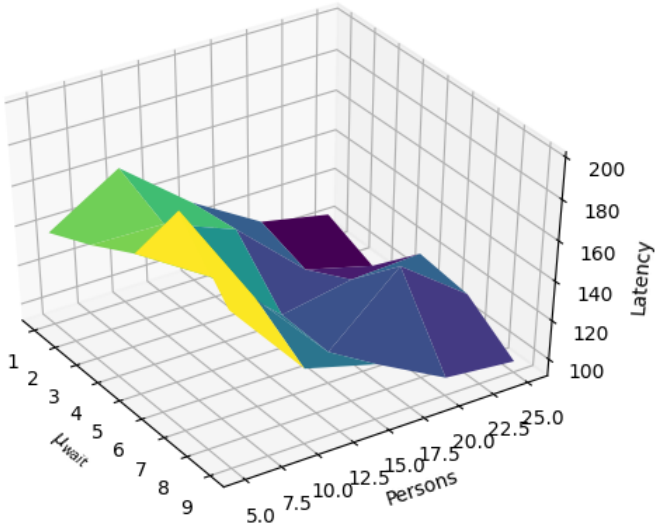

Fig. 11. 3D plotting for latency wrt number of persons and mean of waiting time in Open Model

who are headed to it. On the other hand, we also increase the dead time for which a person doesn't contribute to sharing new events: this is because events are sharing with the other LAN participants as soon as the person enters said LAN. In the case where no new people enter the LAN, the time spent waiting is wasted, which makes for a higher latency. This behaviour can be seen in Fig 11 (for the Open Model) and Fig 12 (for the Transitive Interest Model): increasing the wait time brings, once again, inconsistent results, not allowing us to clearly determine which is the best parameter.

V. Conclusion

Analyzing the results obtained we can see how the gossiping with append-only logs protocol is only really useful in areas where the only concern is the decentralization of communications and control over the data produced by the system's participants. The high latency times make this model highly inefficient for environments that require response times under the tens of hours. The gossiping and log-based nature of the model makes also for a inefficiency in the case of large-scale networks of participants, where high-volumes of logs are exchanged, even in the Transitive Interest model. The Open Model can be used in small networks of people where there is mutual trust between the participants: this makes for an easier version of the protocol that also allows lower latency

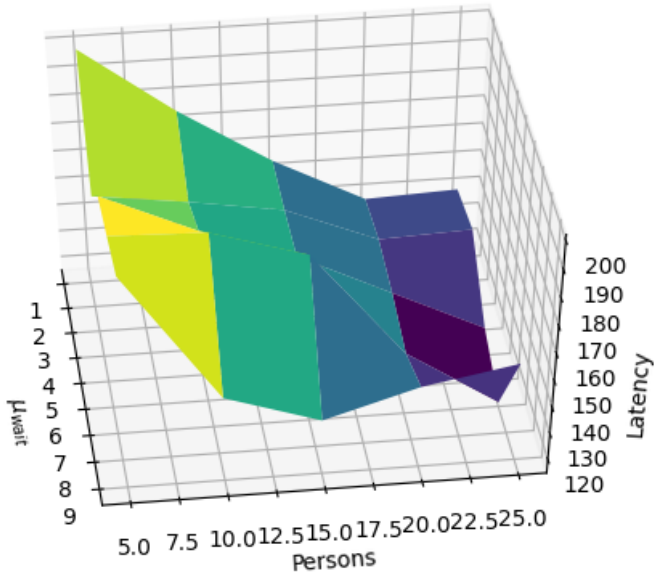

Fig. 12. 3D plotting for latency wrt number of persons and mean of waiting time in Transitive Interest Model

values. Meanwhile, the Transitive Interest Model is aimed at lowering the replication of logs in situations where the number of participants is unbearable for the Open Model, at the cost of higher latency values due to the lower diffusion of created events.

Deriving a global consideration from all our simulations

of this protocol, it is evident how a high number on people combined with a high number of LANs always leads to lower latencies and better performances. The choice between the type of protocol (Open or Transitive Interest), as said above, should be based on the objective of the system and on the given requirements. All the simulations done to create the plots showed in the results section, were executed at least 10 times using the same parameters and then averaging the obtained metrics.

HOW TO INSTALL

In order to install our simulator, you have to follow these steps:

- Download the *installer.jar* file from google drive. You can find the download link in the Readme file of our repository on GitHub;
- Execute the *installer.jar* file. From the console you have to type *java -jar installer.jar*;
- An installation wizard will pop-up, just follow it;
- Once installed, you can execute it by opening the *start_model.bat* file or the *start_model.command* one. Adjust the parameters and then enjoy the simulations!

REFERENCES

[1] Anne-Marie Kermarrec, Erick Lavoie, Christian F. Tschudin, "Gossiping with Append-Only Logs in Secure-Scuttlebutt", DICG'20, December 7–11, 2020, Delft, Netherlands

[2] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994). Design Patterns: Elements of Reusable Object-Oriented Software