Assignment 2 Report

Gossiping with Append-Only Logs in Secure-Scuttlebutt

Luca Fregolon, 211511
luca.fregolon@studenti.unitn.it

Alessandro Sartori, 215062
alessandro.sartori-1@studenti.unitn.it

I. Introduction

In this report we present our work for the second assignment of the course Distributed Systems 2, which consists in an Actor-Based simulation of a distributed protocol using the Repast Simphony framework. The mentioned protocol [1] consists in two replication models that derive from the already existing Secure Scuttlebutt (SSB). The first model is a simplification of SSB and works best if employed in small groups, while the second model corresponds to the technology which SSB already deploys. The main difference between Scuttlebutt and other similar *authenticated single-writer append-only logs* protocols is that synchronization between nodes only occurs by transmitting the latest missing events instead of entire logs.

II. Description of the Protocol

A. Concepts

The fundamental concept in SSB is the **event** entity, which represents a user action. Events are concatenated in **log**s, which are a single-writer append-only sorted collection of events. Each user virtually ”owns” one log that contains their set of actions (each signed with their private key), and each of these participants possesses a local **store** in which it replicates pertinent logs from other users (depending on the replication model adopted). One last useful concept is the one of **frontier**, which is the set of the most recent event known for each log in a store. Frontiers allow efficient comparison between stores to determine the contents of the update messages they will exchange to synchronize.

B. Open Gossip

Open Gossip is the simpler paradigm of the two and is used to illustrate the basic mechanics of the protocol. Essentially, every node replicates all logs unconditionally, maximizing the diffusion of messages and achieving total replication. As the authors state, however, this approach poses an important limitation, i.e., resource usage (and in particular, nodes' memory) is proportional to the *total* activity of the system. This means that this replication model cannot be used efficiently outside of small groups of users, especially if shared contents involve multimedia content (such as videos or music) which would generate an inconveniently high amount of data.

C. Transitive-Interest Gossip

The Transitive-Interest approach refines one drawback of Open Gossip, that is, a node is forced to replicate data even if it is not interested in the contents published by some of the nodes. Therefore, this second model introduces the concept of **following** and **blocking** where, ideally, a node only receives those contents for which it has expressed an interest with a *follow* action. The actual implementation, however, requires nodes to also replicate transitively followed stores (i.e., those followed by one of the followed nodes), up to a tunable number of hops, with the obvious reason of providing enough data replication to ensure eventual delivery. At this point the need for the *block* action arises, given that a node may want to completely avoid dealing with a specific user's data. Blocking a user allows the node to legally discard its data during store synchronization, even if it is part of the transitive graph. Follow and block actions are normally recorded in logs just like any other event.

One considerable advantage of this second gossiping method is the resilience to spam users, since their data won't be replicated if not to nodes that have explicitly requested so.

III. Architecture of the Implementation

A. General Structure

The entry point of the program is the `SimBuilder` class, where the simulation space is initialized and populated with nodes. This class is also responsible for random insertion or failure of the nodes. There are two kinds of nodes, which both inherit from a common interface, named `Node.java`. These two classes (`OpenGossip` and `TransitiveInterest`) implement the two different behaviors described in the paper.

There are some support classes, such as `Store` and `Log`, which deal with data storing and which provide some methods which implement the functions described in the paper, such as *append, update, since, frontier*. The nodes generate an RSA key pair instead of Ed25519, since it is easier to use in Java and the result is the same. A `CustomEdge` class has been implemented in order to manage the delivering of the packets, of which the time to deliver depends both on the distance and on the size of the packet. The propagation speed is implemented to be low and fixed, as our goal was to simulate the behavior in a limited environment, such as a building. The bandwidth is, instead, customizable with a parameter. Packets are stored in an infinite queue before being delivered, since the paper states that the stores are linked through reliable connections such as TCP. For that reason we do not simulate losses or corruption of packets. A `Helper` class has been created in order to contain utility variables and methods which are useful in many places during the simulation. It acts as a container for some static classes, which encapsulate the parameters of the simulation, give the reference of the node by public key, implement a Logger utility and hold the references to the context, space and networks.

B. Links

As previously mentioned, the links in our simulator are implemented as `CustomEdge`. When a node wants to send a packet, it passes it to the edge, which put it in a waiting queue. When the queue is not empty, a packet is retrieved (with FIFO order) and it is put on the link with a delay that vary with the respect to the packet size and the bandwidth. In this way, we can put data on the link with a rate that is defined by the bandwidth. Once the packet is on the link, the delivery is scheduled, based on the propagation time. The propagation time, is based on the distance between the two nodes.

C. OpenGossip

`OpenGossip` class implements the homonym behavior. When a node is created, the node creates its own store, generates its key pairs, and schedules the creation of new events and synchronizations, with intervals depending on the parameters. When a new event is created, it is cryptographically signed and added to the node's own log. The synchronization between two stores starts when a node starts a pull request to another node, by sending a `FrontierMessage`, which contains its Frontier. The node replies by doing the same and by sending all its messages that are newer with respect to the received Frontier. Upon reception of an update, the node checks if the signature is correct (by decoding the message with the public key of the sender, which signed the message with its private key) and if the index corresponds to the next expected message, since the log has to be ordered. When a node has finished to send the news, it sends an `End` message to close the connection. The connection is completely closed after both of them have declared the end.

D. Transitive Interest

The `TransitiveInterest` class implements the second behavior described in the paper. When a node is created, it performs all the operations seen before, but it also starts following (by writing the interest in its log) a set of nodes (depending on the parameter chosen). It does the same also for nodes to block. When it needs to perform a synchronization, before sending its frontier, it updates the store using the function *updateStore*. This function implements **Algorithm 3** of the paper [1]. Every node in the *following* set which is not already present in the store is added to it, while every node in the *blocking* set, if is present in the store, is removed from it. In order to compute the following and blocking set, we decided to conform to the SSB standard, which is to replicate data from 2 transitive follows. Our implementation, however, is a bit different since it prefers transitive blocks over transitive follows and it does not make restrictions about follows if a node is transitively blocked. Since, as the name of this protocol suggests, the choice to follow some nodes is transitive, the nodes can change its follows and blocks during the simulation, with an interval defined by a parameter.

E. Simulation Parameters

The simulation parameters that we added to the simulator are the following:

- **Bandwith (kbps)**: which defines the maximum bandwidth for each link
- **Failure Probability**: a number between 0 and 1 that defines the probability of a node to fail
- **Generation of Messages Interval**: the mean of the Poisson distribution regulating the generation of new messages
- **Insertion Probability**: a number between 0 and 1 that defines the probability of a new node to be inserted in the simulation
- **Interval of Insertion of new Nodes**
- **Interval of Node Failure**

- **Number of Nodes**: the number of nodes created at the beginning of the simulation
- **Number of nodes that each Node blocks**
- **Number of Nodes that each Node follows**
- **Probability to change a block**: a number from 0 to 1 which defines the probability to change someone a block
- **Probability to change a follow**: a number from 0 to 1 which defines the probability to change a follow
- **Simulation Type**: an enumeration with values:
  - OpenGossip
  - TransitiveInterest
- **Store sync interval**: the pull interval to synchronize a store with another node
- **Time interval between changes in block**
- **Time interval between changes in follow**

IV. Analysis of the Results

After implementing the simulator, we mainly tested the model by varying the parameters and observing the consequent changes on the resulting behavior. The simulator, for each run, creates a CSV file, which name includes the most representative values chosen for the simulation parameters. The file starts with `op_` or `ti_` depending on the simulation type. If the case is the latter, also the number of followed nodes is reported. The other parameters reported are, in order, the bandwidth and the number of nodes. The plots that will be presented in the following paragraphs are obtained by analyzing these CSV logs and computing some statistics using several scripts, some written in R and some others in Python.

Parameter	Default Value
Bandwidth (kbps)	500
Failure probability	0.1
Interval of message generation	10’000
Insertion probability	0.2
Interval of new insertions	60’000
Interval of failures	60’000
Number of Nodes	30
Nodes that each node blocks	1
Nodes that each node follows	5
Prob. to change a block	0.1
Prob. to change a follow	0.2
Simulation Type	TransitiveInterest
Store sync interval	100
Interval of follow changes	60’000
Interval of block changes	60’000

TABLE I
DEFAULT PARAMETERS

A. Latency

We tested message delivery latency with various settings, which varied in simulation type (OpenGossip and TransitiveInterest), bandwidth, and number of followed nodes (for TransitiveInterest only, naturally).

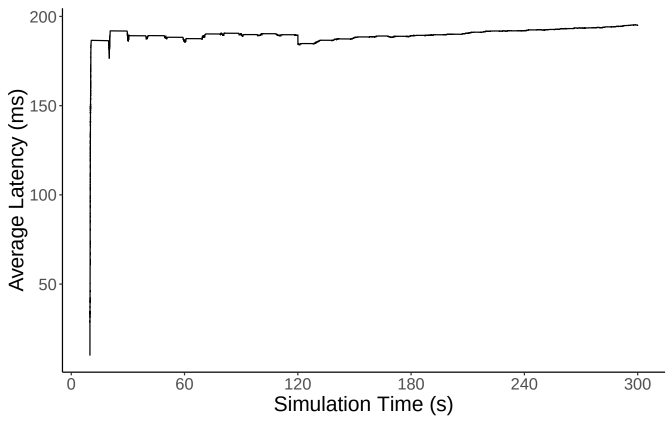

Fig. 1. OpenGossip with over provisioning and 30 nodes

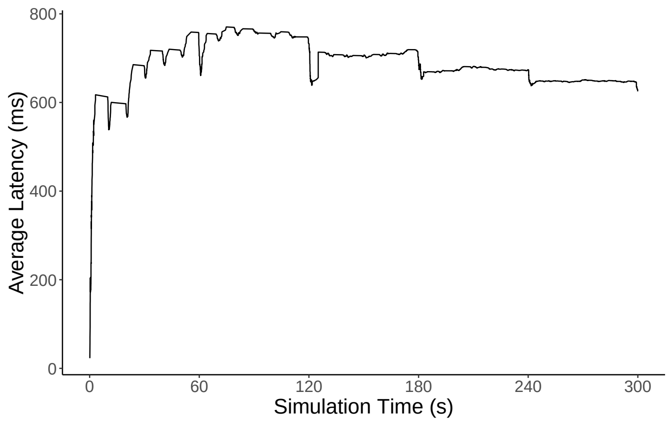

Fig. 2. TransitiveInterest with 1 followed node and 30 total nodes, follow change every 60s

OpenGossip, as we expected, is the model that has the lowest latency, at least with over-provisioning of bandwidth (Figure 1). This is pretty obvious because every node replicates every message, so almost every pull results in something new to retrieve. For TransitiveInterest, instead, some connections can happen between two nodes that have few or no common interests. Therefore the node will need to wait until the next sync to try with another node, with maybe more luck. This waiting interval will then have an important weight in the latency measure. We also observed that, for the same reason, the latency of TransitiveInterest with a higher number of followed nodes (Figure 3) is lower than the latency

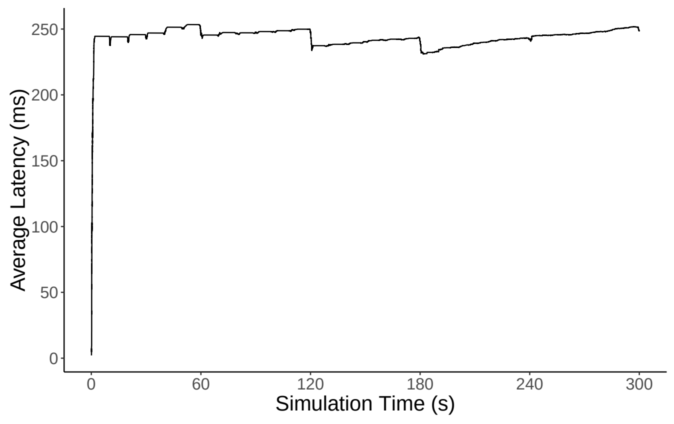

Fig. 3. TransitiveInterest with 5 followed node and 30 total nodes, follow change every 60s

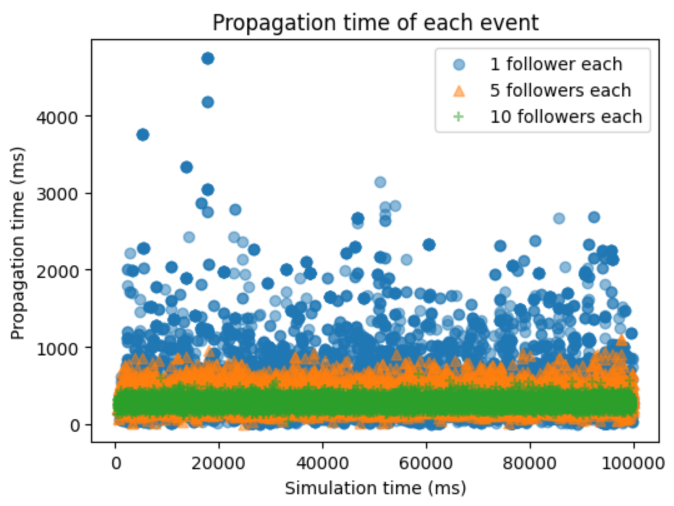

Fig. 4. Times for events to fully propagate

of the same type with a lower one (Figure 2). This is because the less nodes are followed by a node, the lower the chance to contact a node with the same interests, or which, at least, has some information that the node is interested in.

Following these observations, we thought and proved that they also extend to the concept of event propagation, in the sense that not only messages take less time to be delivered in highly connected graphs, but also events take less time to be completely propagated to all interested users. Figure 4 compares under this point of view three simulations with 1, 5, and 10 average followers for each node, where every point is a generated event and its time taken to reach every intended store. As premised, with smaller numbers of followers the graph is more sparse, leading to a less efficient replication strategy. For completeness, Figure 5 also compares these three results with a more compact and immediate boxplot.

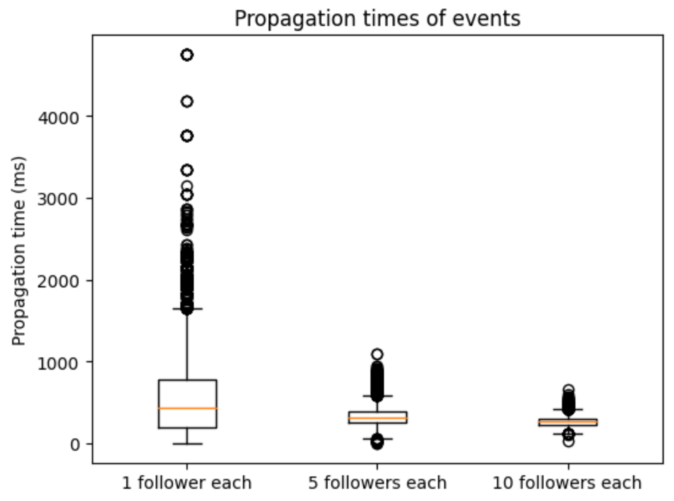

Fig. 5. Time required for the complete propagation of events

B. Store Sizes

On the other hand, though, a smaller number of followers leads to other types of advantages. Above all, nodes are not required to continuously replicate all data, but only those logs they are directly interested in or, for tunable efficiency purposes, those of interest of their followers. A number of dedicated experiments immediately revealed the magnitude of this advantage, visible in Figure 6.

1) Low bandwidth: In conditions of low bandwidth available, the increased propagation delay introduced by Transitive Interest with low numbers of followed nodes, is compensated by the advantage that it has to ask for much less updates. In fact, with low bandwidth (simulated to about 100kbps), the model that performs better is Transitive Interest with 1 follower (Figure 8), which outperforms both Open Gossip (Figure 7) and Transitive Interest with 5 followers (Figure 9).

2) Larger number of nodes: We tested the simulator also with different numbers of nodes. With a significantly larger number of nodes the store size difference between the various types of simulation accentuate. An example with 100 nodes can be seen in Figure 10, where sizes also differ more because of a relatively smaller area of the network being covered. In other words, when following 20 nodes out 30, almost the whole network is covered by the follow graph, while when following 20 out of 100 nodes, a much smaller portion is affecting the logs.

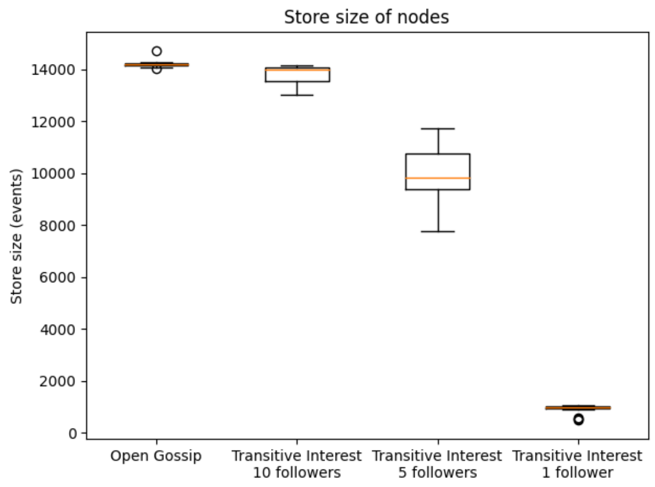

Fig. 6. Comparison of store sizes for different numbers of average followers

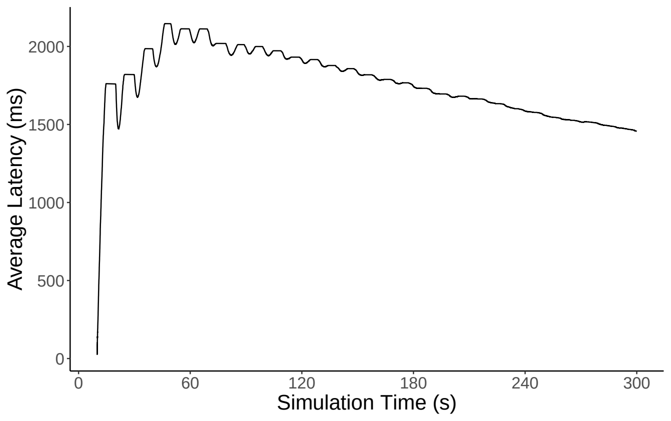

Fig. 7. OpenGossip, 50kbps

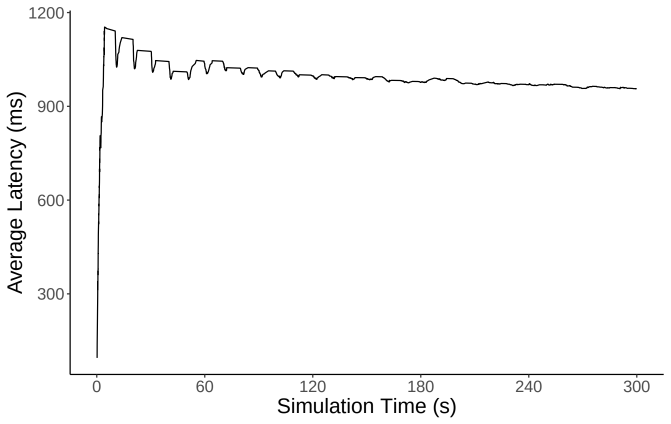

Fig. 8. Transitive Interest with 1 follower, 50kbps

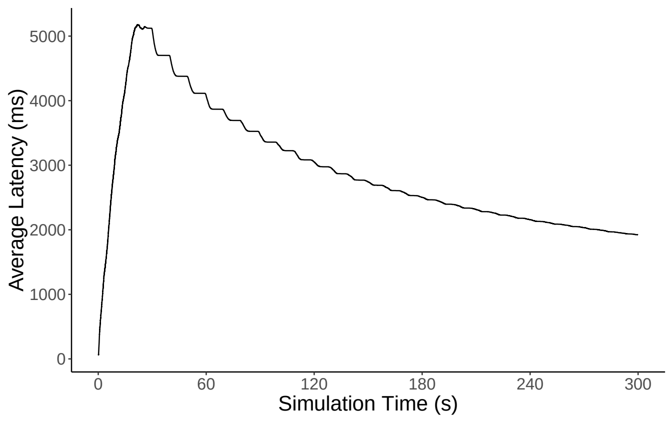

Fig. 9. Transitive Interest with 5 follower, 50kbps

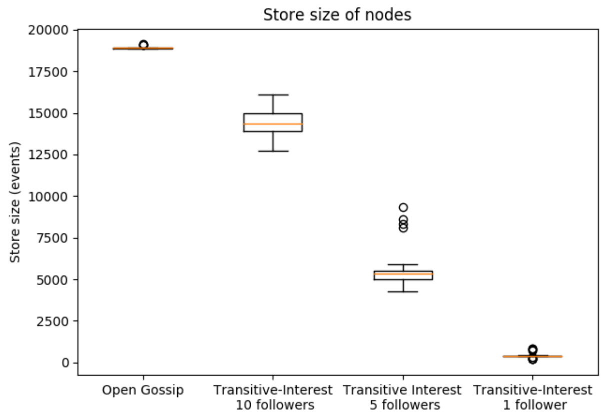

Fig. 10. Comparison of store sizes with 100 nodes

V. Conclusions

The results we obtained with our simulator confirm that the Algorithm proposed in the paper [1] is effective in reducing the store size of the nodes and is then well suited for contexts with a high presence of nodes. Our measures shows that the latency is generally higher than in the Open Gossip model, but the situation overturns in situations with low bandwidth, since the benefits of requesting less news overcome the waiting time until something interesting is found. The limitation of this protocol seems to be that, if a node is not popular, there could be long delays before getting news from it, since almost all the request lead to a miss. This could be solved by using a less random selection of nodes to pull from, by contacting more frequently the nodes for which a node expressed an interest.

References

[1] Anne-Marie Kermarrec, Erick Lavoie, and Christian Tschudin. *Gossiping with Append-Only Logs in Secure-Scuttlebutt*.