Gossiping with Append-Only Logs in Secure-Scuttlebutt

Conti A. (215034), Tonini F. (211961)

January 4, 2021

Abstract

Secure-Scuttlebutt (i.e. SSB) is a broadcast protocol that has pioneered the use of replicated authenticated single-writer append-only logs. The usage of this class of logs excludes from the process of fine-tuning most of the traditionally required parameters for gossiping.

This paper provides the implementation and the experimental analysis of two gossip models used for replication. Such algorithms can be used to build eventually-consistent applications driven by social signals. The first model is named Open Gossip, and was introduced by [1]: it has a thinner body wrt the basic implementation of SSB presented in [2] and its usage is suggested for reduced and trusted cliques. The second model is a transitive-interest model practically deployed by SSB that scales to thousands of participants and is spam- and sybil-resistant.

Our contributions are the following: (i) we implement and dissect the two above-mentioned gossip models, and (ii) present our simulation results with a study of the collected metrics.

1 Introduction

The protocol known as Secure-Scuttlebutt represents an early approach to replicated authenticated single-writer append-only logs, where networking performed with arbitrary data packets is replaced by networking with coherent data structures. Compared to other known broadcast-only communication models, SSB has fewer parameters to tune: part of the reasons for its efficiency are the absence of temporary buffer size and retransmission. In the current work, two models of SSB will be implemented and analysed, namely the Open Gossip protocol of [1] and the Transitive-Interest Gossip. The implementation of the two gossip models is built around the Repast Simphony framework and written in Java.

Similarly to the more famous structure of blockchain, append-only logs provide eventually-consistent replication. However, for the gossip algorithms presented in this paper, each relay has its own log and the system doesn't aim at reaching consensus on a single global view. Moreover, differently from blockchain, SSB is not applied to build social applications with anonymous

participants, but instead is involved in the creation of decentralized platforms interest-driven. These environments are built for particular communities of limited dimensions, that evolves in disjointed networks.

The first model presented is named Open Gossip, and is a simplified gossip model that aims at maximizing the diffusion of events by replicating them in all the relays. The model implements total replication and total persistence, and even if the protocol is open, private operations are possible. The second model is the Transitive-Interest Gossip model: this algorithm can handle a great number of participants and inherently support a publish-subscribe architecture, where participants can specify their interests and disinterests. Indeed, this implementation supports subjective interest-based replication as well as spam- and sybil-resistance.

The rest of the paper is structured as follows: in chapter 2 an introduction on the core concepts is presented, while chapter 3 focuses on the Java implementation alongside the description of the structure and options adopted by the authors. Going forward, chapter 4 describes the tests and the results of the different scenarios that have been tested. Lastly, in chapter 5 some future works and conclusions are drawn.

2 Related works

In SSB, events are the primitive structure of information and logs are collections of events. Each participant generates new events and stores them into its own log, hence a log is a sequence of events of the same replica. A set of logs of different participants is called a store.

An event is identified by the public key of the creator, the hash of the event that precedes it and an incremental index. Moreover, an event has an application-specific field, called content. Finally, each event contains its signature: the cryptographic initials of the other fields of the object. This last attribute is used to guarantee immutability.

In both Open Gossip and Transitive-Interest Gossip, the propagation of new events is performed by sending local stores. The difference of events in the received and the local stores represents the news contained in the message.

2.1 Open Gossip

Open Gossip is the first gossip model analysed in the current work. The model has the objective to maximize the diffusion of events by replicating them in all participants. Therefore, it implements total replication and provides total persistence. The model works best when used for applications about small and trusting groups.

While rather simple, this open model suffers a couple of limitations. First of all, the memory usage for each store grows proportionally to the total activity, hence increasing latency over time. Second, participants have no agency.

2.2 Transitive-Interest Gossip

Transitive-Interest Gossip is the second gossip model analysed in the paper. This model extends the previous one by making it so that replicas can decide to block or follow others. Whenever a node follows another, it will actively replicate its content to its store, while it will ignore it if the other node is blocked.

In the propagated stores, nodes' interest/disinterests are propagated as well. Doing so, both the content and where it should flow propagate at the same time through the same gossip. Indeed, a replica expresses its interests by recording a follow/unfollow or a block/unblock event in its log.

Given the transitive aspect of the protocol, more logic is needed to support also friends' interests. Below, all the rules used in the protocols are summarized.

1. following: the node will replicate the log.
2. not following: the node will not replicate unless a friend follows it.
3. blocking: the node will not replicate it.
4. friend blocks: the node will not replicate unless another follows it.

3 Implementation

The implementation of the open and transitive-interest protocols is part of a simulation built with the Repast Simphony framework and the Java language. The simulation is implemented as a 2D space where potentially infinite nodes may join.

3.1 Protocol

The main components of the protocol are *events*, *logs*, and *stores*. Because of the nature of the Java programming language, each of the above entity is implemented as a plain Java object with properties and functions that represent the primitives associated with each component. Moreover, as per the protocol requirements, two methods that implement the gossip models are developed. Additionally, an update function is necessary to terminate the gossip exchange.

3.1.1 Events

Each instance of an object of type *Event* contains several properties such as:

- ID: public key of the observer who created the event.
- $Previous$: hash of the previous event in the log. This field can be empty if the current event is the first one of the log.
- $Index$: a sequential number that uniquely identifies each event in the log.
- $Content$: application-specific message.
- $Signature$: cryptographic initials of the above attributes. It ensures immutability of events.

- *Type*: represents the class of event. An event can be of type *MESSAGE*, *FOLLOW*, *UNFOLLOW*, *BLOCK*, and *UNBLOCK*.

Upon creation, the event is signed using the private key of the owner and is then stored in its log.

3.1.2 Logs

A log is a collection of events from an observer. Each object of type *Log* is responsible for storing and updating the append-only structure that collects all events created or propagated by a given observer. While the internal structure is trivial, the key elements of a log are the primitives that handle the log structure. Such primitives are:

- $Append(event)$: add a new event to the log.
- $Update(event_list)$: add a set of events to the log.
- $Get(start_index, end_index)$: returns a set of events in the given range.
- $Follow(node)$: register a new node as followed.
- $Unfollow(node)$: unregister a node from being followed.
- $Block(node)$: register a node as blocked.
- $Unblock(node)$: unregister a node from being blocked.

Each observer has a personal log and a set of logs from other observers. Such logs are embedded in a store.

3.1.3 Stores

A store is a map of logs, one for each observer. The main responsibility of an object of type *Store* is to maintain each log up to date with the latest news received by other stores. Moreover, the store maintains a "frontier": a set that contains the latest known event of each log. The store's primitives are:

- $Add(log)$: add a log to the store.
- $Remove(log_id)$: remove a log from the store.
- $Get(log_id)$: returns a log from the store.
- $Ids()$: returns the id of each log stored (i.e. observers' public keys)
- $Frontier(log_ids)$: return the current frontier.
- $Update(events)$: update the logs in the store

3.1.4 Updating

After the gossip function has been completed, the observer is ready to update the local store. The main goal of the update function is to calculate the current frontier of the local store, to retrieve the events happened-after the frontier from the remote store. The set of news is then appended to the local store. At this point, the local store and the remote store are up-to-date for each log that they both contain.

3.1.5 Open Gossip

Upon arrival of a store object from a remote observer, the local observer calculates the local ids set and the remote one. Thanks to this procedure, the observer can determine which ids have not been added yet to the store. If there are logs that have never been stored locally, the protocol adds an empty log for each one of them. This last step is critical since not adding them prevents the update function to calculate the correct frontier.

3.1.6 Transitive-Interest Gossip

While the main steps of the two gossips are identical, they mainly differ in what is logged. In fact, upon arrival of a store object from a remote observer, a node calculates the followed set of its local store from its point of view. Then, it calculates the blocked set in the same way. The followed set contains a list of relays that must be stored, while the blocked set contains observers that the node can safely remove from the local store.

3.1.7 Dynamic membership

Both the gossip models allow dynamic membership. More precisely, there are both a creation and deletion probability that are evaluated at fixed intervals to decide whether to remove or add new nodes. When a node joins, it does not know anything of the network apart from its clique, but most importantly others don't know of its existence. Hence, to address this problem, a *discovery* message is broadcasted and propagated in the network so that everyone in the network can target the new node as the recipient of future gossips.

3.2 Environment

Like many works in the Repast Simphony environment, the "tick" simulates time passing. *Observers* are the main agents of the simulation; each one of them is identified by a public key and is made of multiple objects that define its state at any instant. Each node decides to append new messages to its local log every multiple of 50 ticks. The message is embedded into an object of type *event*.

Every 250 ticks each relay starts a gossip exchange. During a gossip exchange, the observer sends its local store to others in the simulation, so that the recipient can update its local store. The recipient of the local store, will then

send back its store to the sender of the first one, to terminate the two phases of the protocol.

In addition to the two main jobs of the observer, each agent is capable of following or blocking others every 200 ticks. In other words, each node can decide to not store events from specific observers as well as announce active interest in others.

The simulation can remove and add new observers to the space so to simulate dynamic membership. A uniform probabilistic distribution determines the likelihood of changing the number of observers every 100 ticks.

The underlying communication mechanism is the one described in the authors' previous project. In their last work, messages are exchanged using the concept of solitons and by propagating them as waves in the space.

3.3 Simulation parameters

Many of the variables that define each simulation are user-configurable without modifying and recompiling the model. Such parameters change key aspects of the simulation such as:

- Clique[1]: number of observers that are followed by a node at initialization.
- Create event probability: the likelihood to create a local event.
- Follow/Block probability[1]: the likelihood to follow or block an observer.
- Grid size: side of the space; the area is evaluated as (grid size)x(grid size). The space is represented by a 2D square and its size never changes during simulation.
- Kill observer probability: the likelihood that an observer crashes.
- New observer probability: the likelihood that a new observer joins the simulation.
- Observers count: the number of observers that populate the space at the start of the run.
- Topology: defines how to position each observer in the space; can be either normally or randomly distributed.
- Perturbation radius and speed: defines the speed and radius of each perturbation (those two values will be part of the TTL calculation of the perturbation).

4 Experiments

In this section, some evaluations on the two gossip models are performed. In regards to the limits of the SSB protocol, the authors suggest the reading of their

[1]This option is valid for transitive-interest gossip only

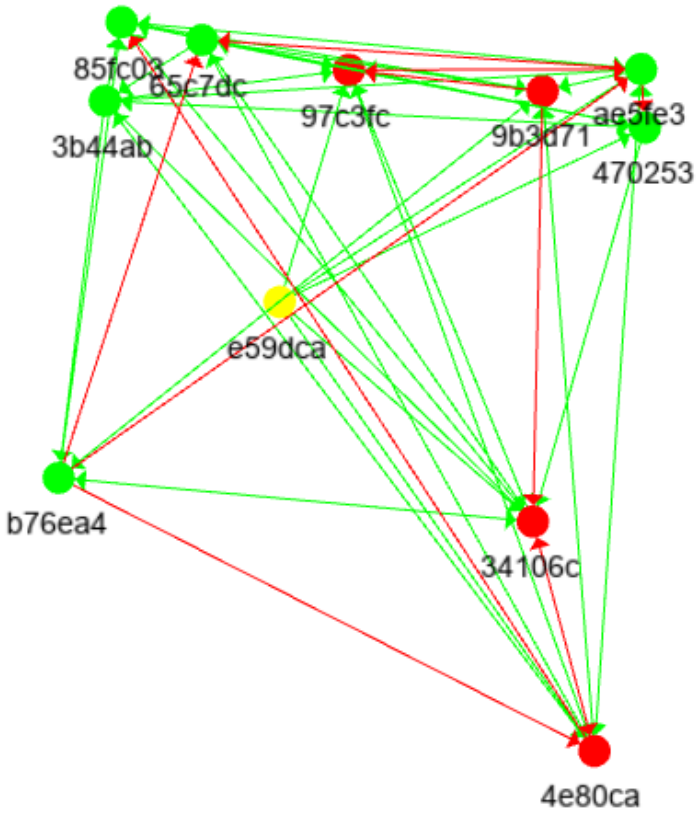
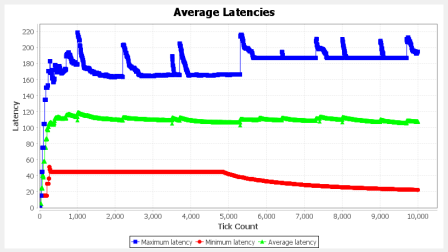
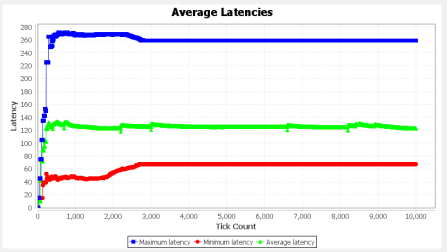


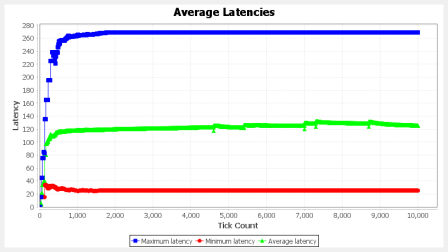
Figure 1: An example of a simulation with Transitive-Interest gossip. The green edges are follows while red ones are block. Red observers have crashed. Yellow ones just joined the simulation.



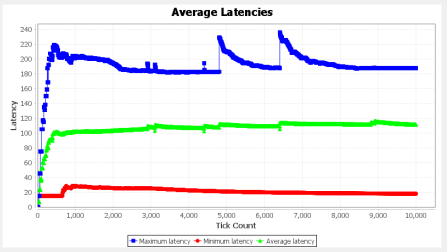
(a) Open gossip, 20% creation prob.



(b) Open gossip, 100% creation prob.



(c) Transitive-Interest gossip, 20% creation prob.



(d) Transitive-Interest gossip, 100% creation prob.

Figure 2: Average latencies for the two gossip models with either 20% or 100% creation probability.

previous work. Indeed, in the last paper, an extensive analysis on the limits of the broadcast append-only protocol has been performed. In the following, the lens will focus on the Open and Transitive-Interest Gossip models instead of on the SSB limits.

Three experiments will be discussed below regarding both the models, while an extra one will only be carried out exclusively on the Transitive-Interest model. The presence of the special experiment is due to the addition of simulation parameters on the second gossip model. To be more precise, the fourth case study will consists in evaluating the performance of the Transitive-Interest model under different follow/block probabilities.

Unless otherwise stated in the experiments, the default values of the different parameters for each simulation will be the following. The space will be defined by a 20x20 grid, and each observer will be randomly positioned in it according to a random topology. Each one of the 20 observers will start with 5 friends (i.e. relays it follows), and each perturbation will have a radius of 10 cells and a speed of 1. Each round, a relay has a 50% chance to create a new event and a 10% chance to update its relationship with another node (i.e. change its state to follow, block or neither). Finally, each round there's a 10% chance that a new node enters the network or that an old node leaves it. Each experiment is stopped after 10'000 ticks.

4.1 Create event probability

In the current experiments, the authors will test the behavior of the two models when faced with different creation probabilities. This experiment was carried out to understand how the two models manage different quantity of news. Indeed, by increasing the probability of creation, the dimension of the stores grows at faster speed. This should be a limit of the Open Gossip model, where stores are copied everywhere. Both the protocols are tested, and the values used are 20%, 40%, 60%, 80% and 100% probability at each round.

For briefness, in figure 2 only the first and the last values of the tested range are presented. Still, the reader can expect a linear increase/decrease in the results of the omitted steps. As can be noted from the presented images, in Open Gossip all the latencies increase with an increase in the creation probability, but the curves get more stable. On the other hand, for the Transitive-Interest model, while the latencies more or less stays the same, the metrics get more unstable.

The reason behind the behavior of the first protocol can be explained by understanding that with an increase in probability for the creation of events we have bigger stores to perturb and therefore the latencies increases. In regards of the stability, since the number of created events gets more and more deterministic, the curve suffer from less deviation. About the Transitive-Interest protocol, with an increase in the total number of created events, there is a more harsh filtering process due to nodes' interests, hence the curve gets less stable.

4.2 Growing and decaying networks

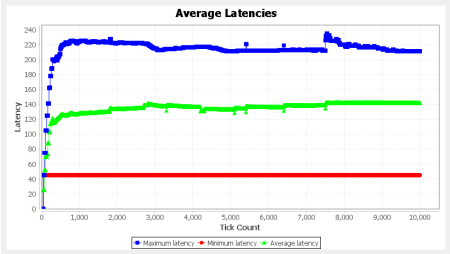
In these two experiments, the Open and the Transitive-Interest models will be analysed in a situation of growth and decay in the number of nodes. These tests were performed to analyse the resistance of the two gossip models to the variation in the number of participants. For these studies, the tests are performed first with a new observer probability of 50%, and then with a kill observer probability of 50%. Since the variation in latencies seems to not differ too much between the two models, below the authors will instead analyse the average updates. Still, an example of how the latencies vary in growing and decaying networks applied on the Transitive-Interest gossip model, can be found in figure 3.

In respect of the average number of updates (which refers to the average number of news found in a received store), the four cases studied can be found in figure 4. Moreover, the number of observers by their type is also presented for the Transitive-Interest model in figure 5. Finally, also an analysis on the payload types is visible in figure 6.

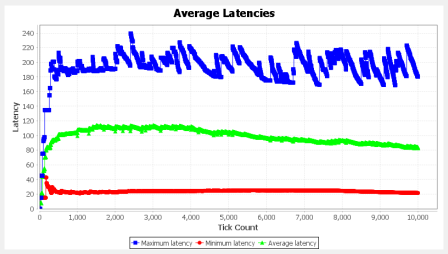
4.3 Follow probability

The last experiment considers only the Transitive-Interest gossip model. It consists in perturbing the follow probability, which is the chance by which an observer will update the relationship it has with another node. This test is performed to understand better the resistance that the gossip model has against highly-mutable networks, where follows and blocks varies really fast.

As for the first experiment, the follow probability varies from a value of 20%

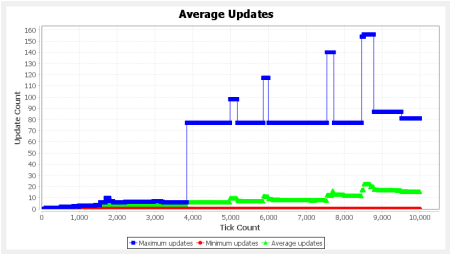


(a) Decaying Transitive-Interest gossip.

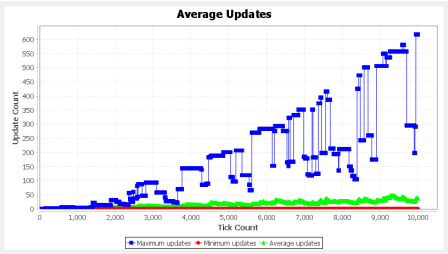


(b) Growing Transitive-Interest gossip.

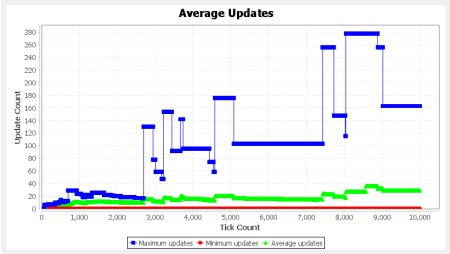
Figure 3: Average latencies for the Transitive-Interest gossip when the number of nodes strongly decreases/increases over time.



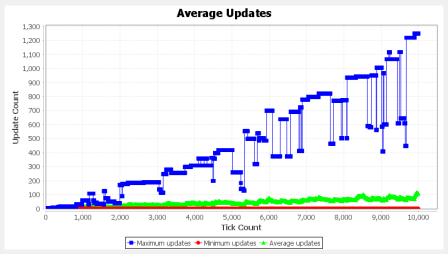
(a) Decaying Open gossip.



(b) Growing Open gossip.

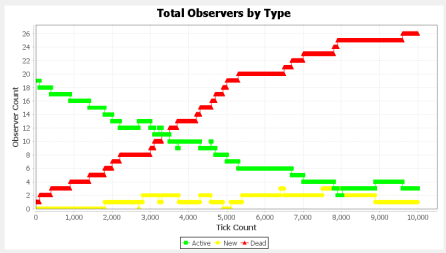


(c) Decaying Transitive-Interest gossip.

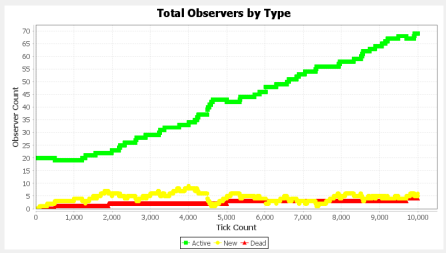


(d) Growing Transitive-Interest gossip.

Figure 4: Average updates for the two gossip models when the number of observers decreases/increases over time.

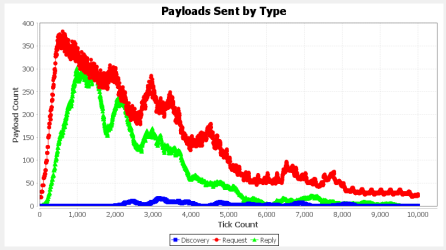


(a) Decaying Transitive-Interest gossip.

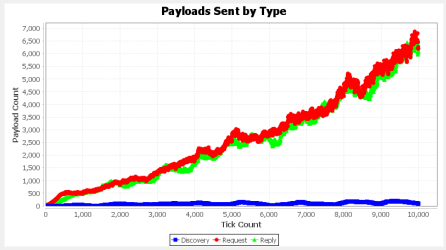


(b) Growing Transitive-Interest gossip.

Figure 5: Observers by their type in a decaying/growing network with Transitive-Interest gossip model.

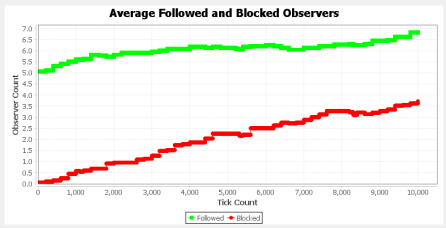


(a) Decaying Transitive-Interest gossip.

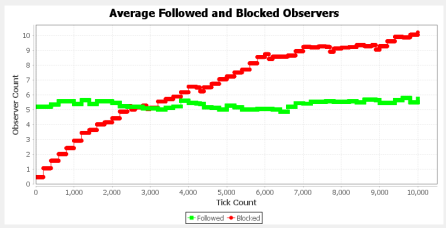


(b) Growing Transitive-Interest gossip.

Figure 6: Payloads by their type in a decaying/growing network with Transitive-Interest gossip model.

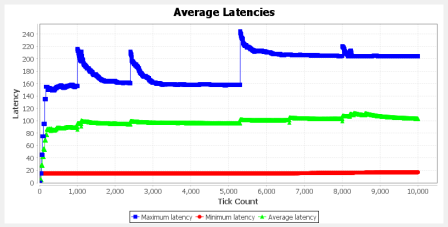


(a) Transitive-Interest gossip, 20% follow prob.

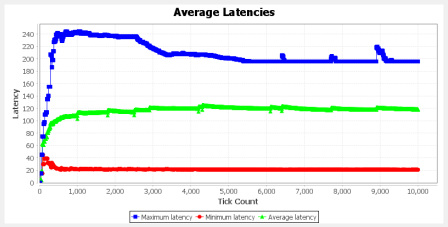


(b) Transitive-Interest gossip, 100% follow prob.

Figure 7: Average number of nodes followed and blocked with the Transitive-Interest gossip model with varying follow probabilities.

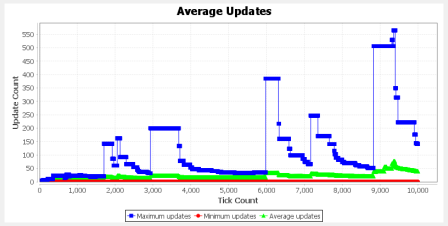


(a) Transitive-Interest gossip, 20% follow prob.

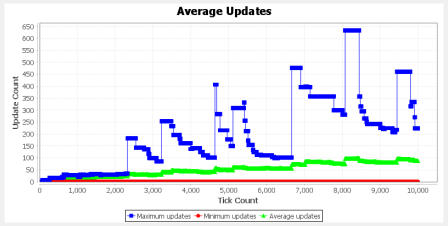


(b) Transitive-Interest gossip, 100% follow prob.

Figure 8: Average latencies in networks with Transitive-Interest gossip with low and high follow probability.



(a) Transitive-Interest gossip, 20% follow prob.



(b) Transitive-Interest gossip, 100% follow prob.

Figure 9: Average number of news per received store in networks with Transitive-Interest gossip with low and high follow probability.

to a value of 100%, by 20% at each simulation. For this experiment, a new metric is collected, namely the average number of nodes followed and blocked. This new insight is summarized in figure 7; once again only the minimum and the maximum tested probability are reported for brevity. Moreover, for these two cases, the average latencies and the average updates are presented in figure 8 and 9.

5 Conclusions

In this work, two gossip models proposed by [1] and [2] were implemented and extensively tested.

This implementation based on the SSB protocol is clearly superior to the previous work presented by the authors. Nonetheless, it better describes the potential of fully decentralized solutions and single-writer append-only logs.

Moreover, the fewer parameters to tune are both an improvement and an advantage for developers. In fact, bootstrapping and network deployment requires less testing time and analysis.

The abstraction of SSB as a backbone permits the creation of multiple gossip models as those seen in the current work. Indeed, although existent gossip models are proved to be efficient in different scenarios, SSB provides for willful developers the framework to develop their own proprietary protocol.

References

[1] Anne-Marie Kermarrec Erick Lavoie and Christian Tschudin. Gossiping with Append-Only Logs in Secure-Scuttlebutt. 2020.

[2] Christian F. Tschudin. A broadcast-only communication model based on replicated append-only logs. Technical Report 2, 2019.