Distributed Systems 2 - Second Assignment

Riccardo Capraro - 203796
Riccardo Micheletto - 215033

January 3, 2021

1 Introduction

We present in this report our implementation and analysis of the Scuttlebutt protocol (SSB) presented in the paper *Gossiping with Append-Only Logs in Secure-Scuttlebutt* [1].

The scope of this document is to present **our** implementation of SSB. The reader will be introduced to the necessary notions in order to understand how SSB works, how we implement it, and to have a general idea of its applicability for our test scenarios. Out of the scope of this document is to reexplain the SSB protocol in detail, or to provide a review of all its applications; nonetheless, we always provide the reader with suitable references to grasp the logic and subtleties of SSB. Although not required for the comprehension of this document, a prior reading of the explanation of the original description of SSB [1] would be beneficial for the general understanding.

The purpose of the SSB protocol is to provide a middleware for social applications, by exploiting replicated *authenticated single-writer append-only logs*. The objective is to provide eventual-consistency data replication as an enabler for the development of social applications, while guaranteeing unforgeable message feeds: only the owner of a feed can write to the feed through an authentication mechanism.

While implementing the gossiping protocol we also explore some interesting use cases: we utilise a wireless-based protocol for the network layer that underlies the SSB implementation and provide ways to easily visualize logs replication, network traffic and protocol information.

We base our implementation on the Repast Symphony framework. For the network layer, we employed our implementation of the Wavecast protocol (Wavecast) [5], a broadcast system suited for wireless networks, that guarantees eventual per-source in-order delivery of messages, i.e. reliable FIFO broadcast. Although a short introduction of the wavecast protocol and the protocol parameters will be given in this document, we direct the reader to [4] for a thorough description of the protocol and to [5] for the description and source code of our implementation.

In section 2 we provide an overview of SSB properties, the requirements for the underlying network layer, and the motivation for introducing the Wavecast

protocol; in section 3 we present the reader with a view of the global solution architecture and defend the most important implementation choices. Subsequently, we introduce the simulation setup (4), analysis (5), to finally bring up our conclusions in section 6. Section 7 details how to install and launch the simulation.

2 Protocol

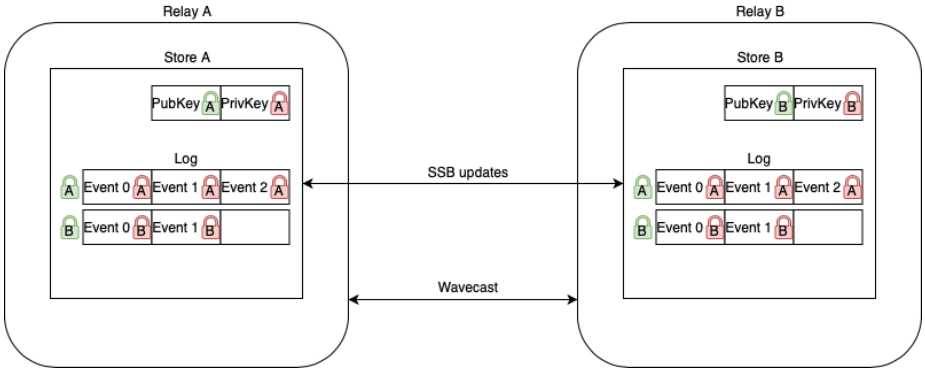


Figure 1: SSB protocol components.

2.1 Secure Scuttlebutt - SSB

SSB is based on the replication of events between **stores** (users), where each store contains a set of single-writer collections of events called logs. An asymmetrical encryption system guarantees that: (1) stores can append events to their logs by signing them with their private key, (2) P2P messages can be exchanged using the destination public key, (3) messages can be verified through a cryptographic signature of the events using the logs owner's public key.

2.1.1 Gossiping protocols

In the SSB outline [1] two gossip protocols are provided: open gossip and transitive-interest gossip. We present here only the protocol properties; the reader should reference the aforementioned SSB paper and the documentation [2] for a thorough description of the protocol details.

In the following paragraphs we will use the terms participant, store and user interchangeably; for the sake of simplicity we assume that every store is associated to a log, i.e. every store actively owns one of the logs of the system, and is associated with a cryptographic key pair, where the set of store ids (public keys) is equal to the set of log ids in the system. To simplify the implementation, we extracted an interface common to both implementations:

```
public interface Protocol {
  public abstract void processHeartbeat(Store store, PublicKey id);
  public abstract Frontier getFrontier(Store store);
}
```

Through the common interface, different implementations can be used by the store interchangeably, e.g. to create interesting test scenarios where different protocols are used by different groups of users.

The open gossip protocol replicates messages unrestrictedly. In particular, it guarantees total replication (all stores replicate all logs) and total persistence (once an event is added to a store, it will never be removed). Two stores are updated by comparing the stores frontiers; a frontier is defined as a set of pairs (id, lastUpdateIndex), where id is the id of the log and lastUpdateIndex is the index of the last event added to the log. By comparing two store frontiers we are able to compute the set of missing events, called news, required to be exchanged in order to perform the update. This mechanism ensures three properties: consistency, if a log id is contained in both stores then after the update the logs in the two stores will contain the same set of events; terseness, only missing events are exchanged; conservative replication, only common logs are exchanged.

The transitive-interest protocol allows to apply SSB to bigger groups of users, where total replication is either too expensive or not required. In particular, a given store may be interested in only a few logs generated by other stores in the network. The transitive-interest protocol allows the user to follow/unfollow and block/unblock other network participants; this actions are executed by storing specific events (e.g. a block event) in a user's log. Transitivity happens following a model that allows a store to replicate logs followed by other participants. In SSB, and in our solution, the transitivity model enforces three properties: (1) two-hop transitive followship, (2) ignore transitive blocked nodes unless (3) no transitive follow exists. In section 3 more details will be provided about how we implemented the transitive-followship mechanism. The transitive interest protocol provides a subjective interest-based replication that guarantees eventual consistency of shared logs. Interest graphs (or similar approaches) ensure spam and sybil resistance: users have to convince other users to follow them, thereby requiring spam and sybil nodes to be able to generate interesting content for other participants. Mechanisms can be put in place to detect sybil identities and block them.

2.2 Network layer - Wavecast protocol

The network layer should guarantee the eventual per-source in-order delivery of messages (reliable FIFO broadcast). Although TCP suits this protocol very well, we decided to extend our previous work on the wavecast protocol by testing SSB on a wireless-based setup. The wavecast protocol guarantees eventual global delivery of generated messages. A set of relays (antennas) is distributed on a 2-dimensional space, with the main parameter being the relay range (also

called wave propagation distance). Messages are propagated from a relay in all directions, and received by all relays in range after a distance-based propagation interval: the farther the neighbour relay, the longer it will take for the message to reach the destination. The wavecast protocol guarantees FIFO delivery, lost messages recovery, and generally provides a robust infrastructure for wireless reliable FIFO broadcast.

SSB protocol messages are completely handled by SSB stores, serialized at the source, packed in waves (the wavecast name for message), and propagated by the underlying network. In particular, the wavecast components have no knowledge about the waves content, nor the SSB stores have any information about the network delivery mechanism, as shown in Figure 1.

3 Solution Design

We chose to simulate the SSB protocol in the most distributed way we could; in particular, most of the implementation effort was put in making the update and followers tracing operations distributed and based on message exchange rather than on shared information; we took inspiration from the actual implementation of the SSB protocol, while simplifying many components to fit into the scope of our work.

The solution is split into two main packages: **scuttlebutt** and **wavecast**. The wavecast protocol implementation coincides with the one presented in [5], with the introduction of the interface to send waves as a replacement for the random wave generation of the original protocol and the wave content being a *String*, containing the serialized (encrypted) SSB payload.

It is relevant to mention that two main steps are key to the SSB implementation: encryption and serialization. The various components of a message payload undergo a chain of serialization and encryption steps at the source and deserialization and decryption steps at the destination, as depicted in Figure 2. These steps are key to guarantee the privacy of message contents and to understand to whom they were intended for.

The hierarchy of the project involves having the Wavecast as a subcomponent of SSB: wavecast objects (the relays) and wavecast charts/metrics are isolated into a Repast Symphony (Repast) subcontext of the simulation context. The subcontext is called *wavecast*, while the whole content is called *scuttlebutt*.

Similarly, the context builder of the whole simulation, found in the scuttlebutt.context package, builds the *wavecast* subcontext using the Wavecast builder in wavecast.context. During contexts creation, the relationship between store and relays is established.

3.1 Scuttlebutt

SSB is composed of several pacakges, as shown in Table 1. We designed the system to be as modular as possible (e.g. see how we abstracted on the gossip protocol interface), while allowing common functionalities to be used through

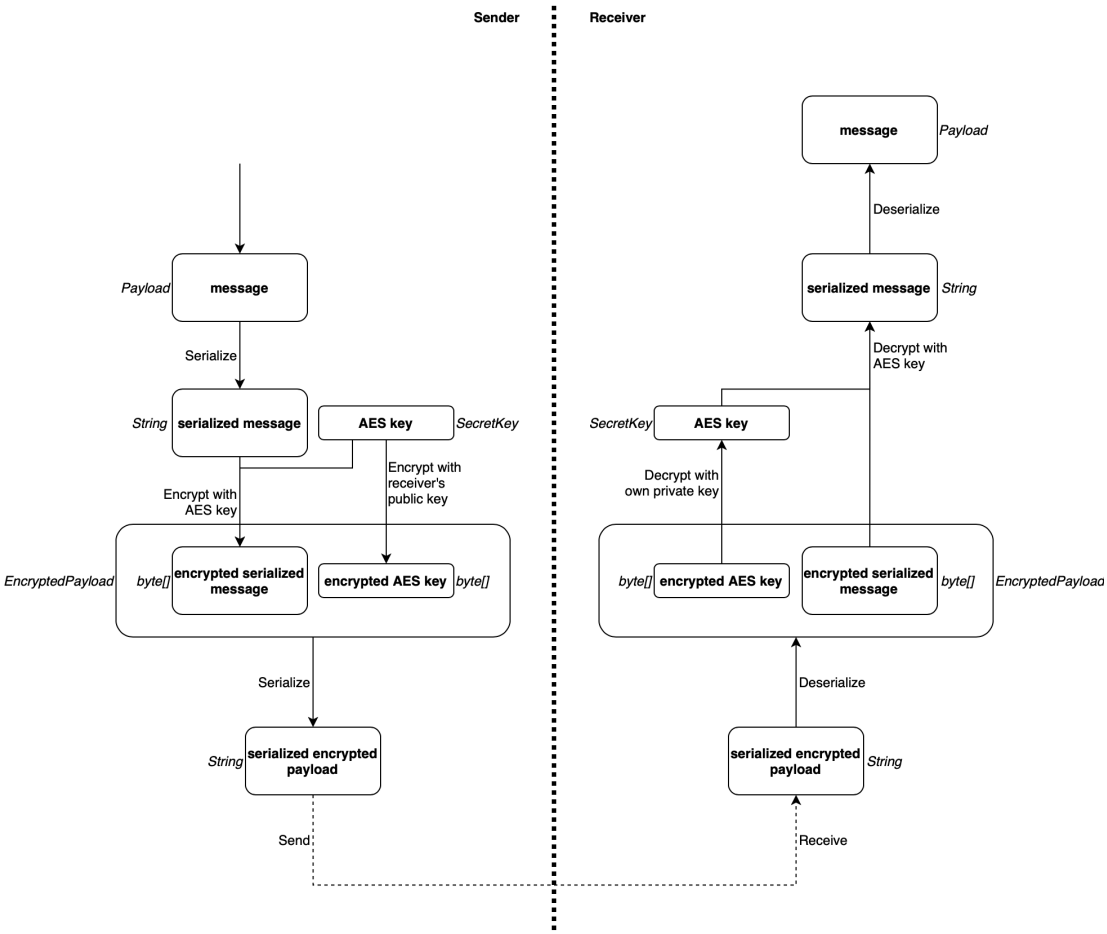


Figure 2: Encryption and serialization process

shared implementations (e.g. our implementation of serialization and encryption).

Heartbeat The choice for a user to subscribe to a given log can be simulated in different ways. We chose to have stores broadcasting to all stores in the system a periodic non-encrypted heartbeat message that contains the store's public key. Upon heartbeat receipt, the receiver can choose to add the new log. In the Open Gossip protocol (Open Gossip) the store always chooses to subscribe, whereas in Transitive interest (Transitive Interest) instead of directly adding the log we perform four different operations with different probabilities: follow, unfollow, block, and unblock.

analysis	contains sources for scuttlebutt-related charts and metrics
configuration	wrapper for simulation configuration parameters
context	contains the simulation ContextBuilder
crypto	encrypt/decrypt utilities
event	log event definition
exception	custom exceptions
log	logs definition
payload	messages payload definition
serialization	serialization utilities
store	contains the store definition
store.protocol	gossip protocols implementation
styles	custom styles for Repast displays

Table 1: Scuttlebutt packages

Update Stores periodically schedule updates once every simulation period, assigning the arrival time of the update task at a random tick within the next period. In other words, at every instant of time $t = k * T, k >= 0$, with T the period of the simulation, a set of updates requests to be sent to other nodes are scheduled at time $t' \in [t, t + T)$. The update follows a three step approach, as depicted in Figure 3: given two stores A and B, and store A starting an update towards store B, A will send a PULL request with its frontier; the content of the PULL request will be encrypted using B private key, to ensure only B can receive the request. Upon a PULL receipt, B will compute the set of news needed for A to perform its update, based on the received frontier; it will then send the news back to A, encrypting them using A private key, along with its frontier, using a PUSH-PULL message. Upon receipt, A will update its logs using the news it has just received, and send back to B the set of news B needs for its update in a PUSH message.

Summarizing, four are the types of messages that travel the system as **Payload** of the waves: PUSH, PULL, PUSHPULL and HEARTBEAT. Although we use an enum to distinguish between them in our code, the content of the message is what determines the type: all messages contain the senderID; if a message is not encrypted then it is a HEARBEAT, otherwise if it has only the frontier it is a PULL, if it has only the news it is a PUSH, and if it has both it is a PUSHPULL.

Encryption To implement encryption we use a two step encryption model: every store has a RSA key pair that is used to authenticate the message exchanges, whereas to encrypt the messages payloads we use a one-time symmetric key generated using AES and exchanged using the RSA mechanism. When a message has to be sent, the **Payload** is encrypted using a newly generated symmetric key, put in an **EncryptedPayload** object, along with the symmetric key encrypted using the public key of the receiver. Upon reception, the store will use its private key to decrypt the symmetric key for the transaction and

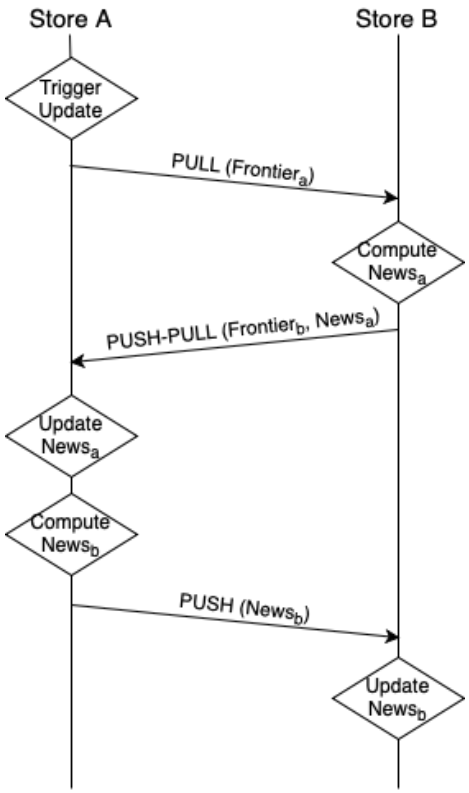


Figure 3: Update message exchange

use it to decrypt the message payload.

Serialization To simplify serialization steps, we use a templatized class called **Serializer** with the following interface to serialize and deserialize objects:

```
public static String serialize(Serializable serializable)
    throws SerializationException
public T deserialize(String serialized)
    throws SerializationException
```

Frontier computation While it is easy to calculate the frontier of a store in the Open Gossip, because it only requires to look at the last event index of all the logs contained in the store, for Transitive Interest it is different: we need to know the logs we follow, the ones our followers follow, and nodes that we need to exclude because blocked or because we decided to unfollow them. In the SSB implementation [3] the two-hop transitive interest based on the rules defined in section 2.1 is computed by navigating a weighted graph that stores the information; to simplify this approach, we compute the frontier every time

by simply looking at the store logs: in particular, to compute the frontier of a given store A, one can retrieve the information about nodes followed and blocked by A from its own log, in linear time with respect to the log length. Unfollowing and block operations result in the removal of the log from the store. Transitive followees are instead computed with the same mechanism by looking at the logs of other stores locally persisted by A. This is obviously a less efficient yet simpler approach than maintaining a stateful view of the store relationships. It is worth mentioning that analyzing the locally stored logs of a store in order to transitively track the followees of other stores is correct: indeed the algorithm guarantees that *eventually* each store receives the events of all its followees, thus also the follow and block events.

3.2 Wavecast

The Wavecast protocol is based on relays, and for simplicity every relay contains only one SSB store. Relays can join the simulation based on wavecast parameters. In particular, every time a relay joins the simulation a store is also added; the impact of joining nodes can be substantial if this happens on the very last periods before the heartbeat process is stopped to compute convergence time. However, a very small improvement, such as starting an update against the heartbeat sender whenever a log is added to the store, will cut down this startup time dramatically, given that the heartbeat sender is also the log owner, hence making it the store with the most updated information about the log in the system.

As mentioned, there is complete decoupling between Wavecast and SSB. The interface below shows how stores receive from the underlying relays a string containing a serialized **EncryptedPayload** through the deliver method, whereas relays receive the same object through the sendWave method.

```
//Store
public void deliver(String encryptedPayload)

// Relay
public void sendWave(String payload)
```

4 Simulation

As explained above, our Scuttlebutt implementation has been built on top of the Wavecast broadcasting system. Therefore, during the simulation phase, most of the Wavecast simulation parameters, displays and charts are still present. The reader is redirected to a report on our implementation [5] for more details about those Wavecast-specific components, while here only the ones related to the Scuttlebutt algorithm will be explained. In this implementation, the user will not find the parameters related to the wave generation anymore: waves are generated after message are sent by the upper Scuttlebutt layer instead.

4.1 Parameters

SSB	Wavecast
followProbability	randomSeed
unfollowProbability	relaysNumber
blockProbability	spaceHeight
unblockProbability	spaceWidth
eventGenerationProbability	wavePropagationDistance
updatesPerPeriod	messagesPerTick
eventsGenerationMaxPeriod	tickAccuracy
heartbeatRatePeriods	loadColorScale
protocol	relayJoinMaxPeriod
storeLoadColorScale	relayJoinMaxNumber
	relayJoinProbability
	latencyRatio
	lossProbability

Table 2: A list of the simulation parameters

A summary of the simulation parameters is presented in table 2. We describe SSB parameters in this section, whereas the description of Wavecast parameters can be found in [5]. By changing the **eventGenerationProbability** parameter, the user can choose the probability with which an event is generated by a store every tick. No more events are generated after **eventsGenerationMaxPeriod** periods, allowing to verify whether the system converges and all the updates are properly propagated. Each store triggers **updatesPerPeriod** updates every period.

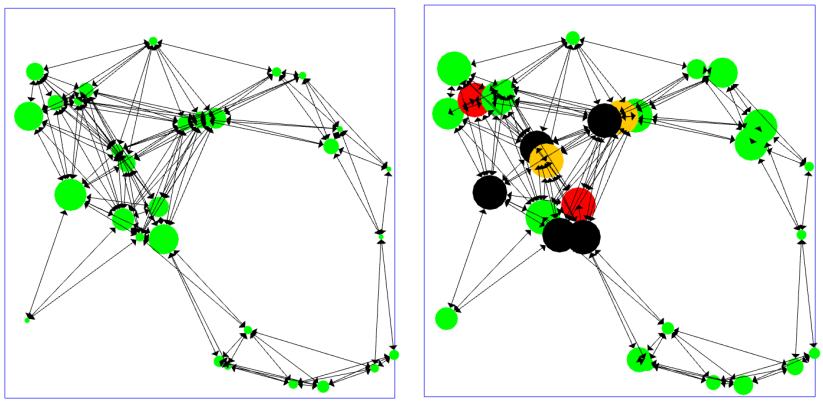
A number of heartbeats equal to **heartbeatRatePeriods** is sent every period, while when an heartbeat is received a store follows the sender with **followProbability**, unfollows it with **unfollowProbability**, blocks it with **blockProbability** and unblock it with **unblockProbability**. We should note that in a social application these operations are usually carried out by the user based on their interests, and that we simply provide a very approximate way to simulate the user behavious. Through the **protocol** parameter the user can choose between the *OpenGossip* and *TransitiveInterest* gossiping mechanisms.

The **storeToHighlight** parameter allows to choose which store to visualize in the **followee** display. Further details will be provided in the display's description; about the parameter, it is sufficient to know that it must be set specifying the ID of a store, in the format *Store@12345* (the output of the toString method computed on the store object), that can be retrieved by double clicking on the store shown in the display. Finally, the **storeLoadColorScale** can allow to adjust the scale on which the store dimension display coloring is based on.

4.2 Displays

Two displays have been created to show relevant information about stores.

Store dimension This display allows to visualize the number of events saved in each store, to understand if portions of the network suffer from an excessive growing, that could cause a denial of service due to heavy storage consumption. The display shows a graph, where the edges represent the underlying Wavecast connections between neighbours. It is important to recall that in the Wavecast protocol neighbours are not really connected, but the graph is used as an abstraction to portray who relays can contact, based on the position of the relays and the wave propagation distance. Nodes of the graph, on the other hand, represent the dimension of the stores, i.e. the number of events appended in each store. The bigger the number of event, the larger the dimension of the circle that is representing the store. Circles grows up to a limit of 5 times the parameter *storeLoadColorScale*; after that, the node changes colors to show increasing dimensions following multiples of *storeLoadColorScale*, following the ladder green, yellow, orange, red and black as described in Figure 4.



(a) First indications based on store size (b) Further indications based on colors

Figure 4: Store dimensions display with different visual indications of the number of events the store contains. For example, assuming a loadColorScale of 100, the colors correspond to green $\leq$ 500 events, orange $\leq$ 600, red $\leq$ 700, black $>$ 700.

Followee After setting the **storeToHighlight** parameter mentioned above, the display shown in Figure 5 uses different colours to show relationships between the specified store. The selected store is depicted in blue, while stores directly followed by it are painted in green. Transitive followees (stores followed by the directed followees) are yellow, and blocked stores are red. Black nodes represent stores without any particular relation with the specified one.

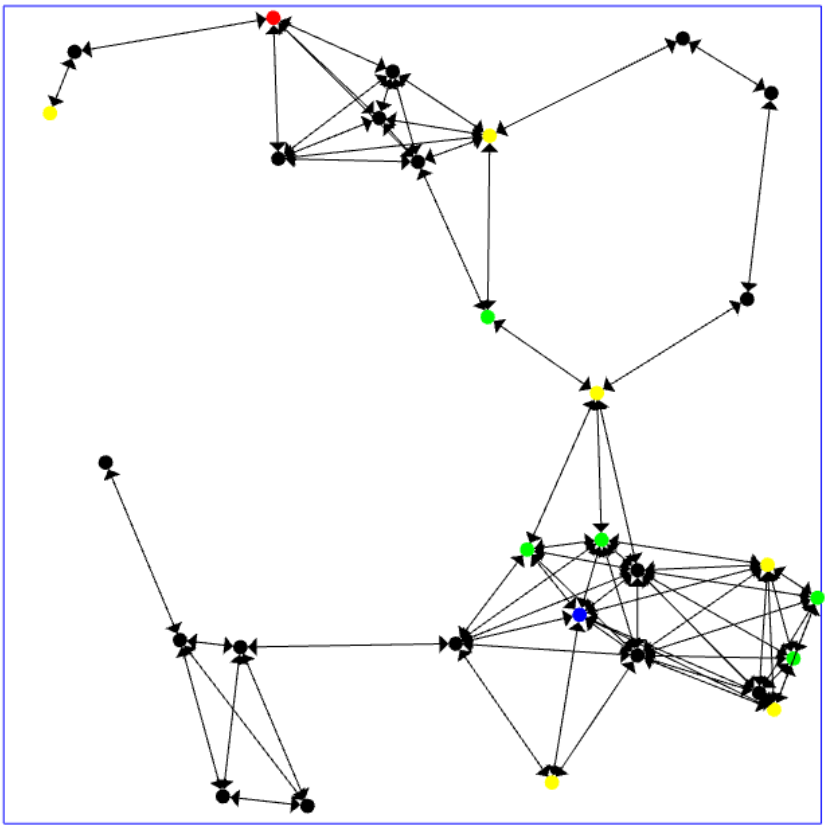


Figure 5: Store relationships chart. In blue we have the highlighted store with its direct (green), indirect (yellow) and blocked (red) followees. Transitively blocked stores are not displayed.

4.3 Charts

News received This time series chart (Figure 6) shows the average number of news received by the stores in each tick, allowing to monitor the trend of the amount of data exchanged during time.

Out of sync stores This chart (Figure 7) has been added to highlight the number of stores that are not yet completely updated, i.e. the number of stores that will receive some news in the future. By looking at this plot it is possible to estimate the time that the system take to converge: when no event are generated anymore (after **eventsGenerationMaxPeriod** periods), updates keep being exchanged, leading all the stores to eventually gather all the needed news.

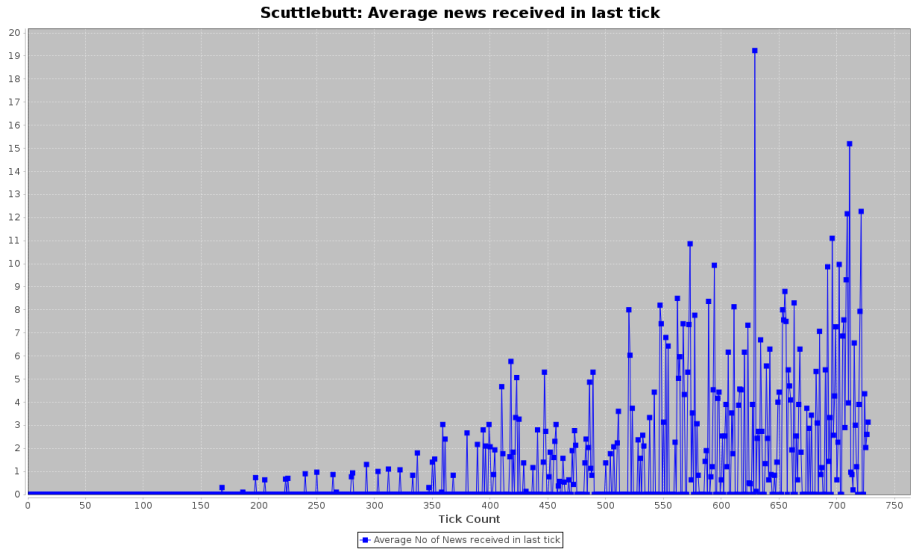


Figure 6: Average news received

Store info Several metrics are plotted in this chart (Figure 8): the average number of events memorized in the stores, the average number of logs memorized in the stores, the average number of directed and transitive followees, and the average number of blocked stores. Note that the number of logs in a store is equivalent to the set of direct and indirect followees.

5 Analysis

The focus of our analysis will be on the performance of the protocol in the wireless scenario we introduced with the support of the wavecast protocol. We want to provide a clear message on the computational complexity of the implementation, while pointing out directions on how to improve it in the presented scenario.

We would like to specify that, although the analysis provided in this document is mainly theoretical and focused primarily on the computational complexity, it has been formalized starting from the numerous practical simulations that we have conducted in this phase and from the plots and graphs that we have generated.

5.1 Cryptography

Cryptography has a great impact on the simulation perfromance. In particular, the initialization of pseudorandom generators takes considerable time. Nevertheless, in a production environment this setup cost is negligible wrt. the protocol.

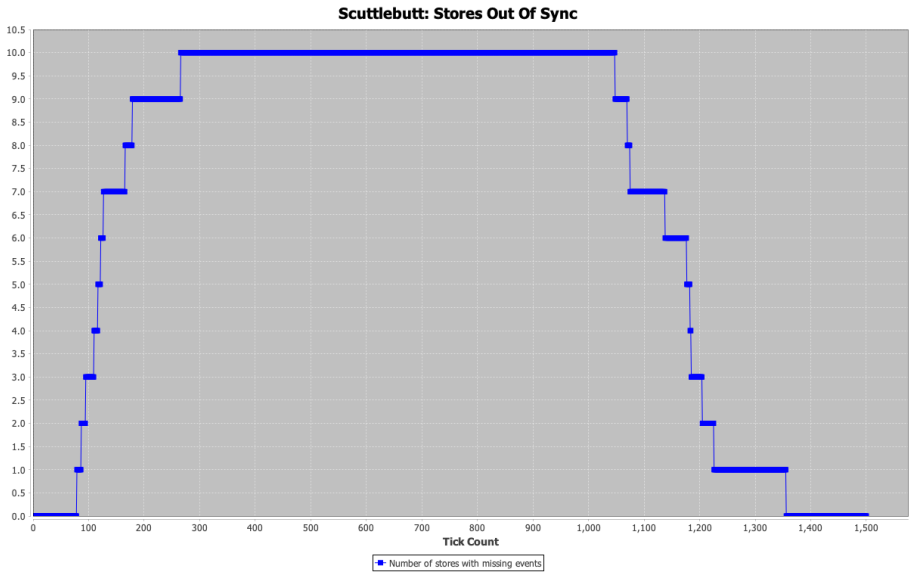


Figure 7: Stores out of sync

5.2 Wavecast

The wavecast protocol introduces overhead in the message exchange. In particular, the benefits in terms of number of messages traveling the network of using TCP over Wavecast is obvious. However, what we saw is that the real overhead in the simulation is not due to the wavecast protocol itself, but mostly to the large number of SSB messages that are exchanged.

Currently, we can informally provide an estimate on the average number of messages exchanged at any period:

$$O(\text{stores} + \text{stores} * \text{updatesPerPeriod} * 3 * \text{avgOutDegree} * \text{stores}) \quad (1)$$

The first term of the expression accounts for the global broadcast of heartbeat messages, when we assume heartbeats are sent once per period. In every reasonable scenario this term is always lower than the second, hence we can omit it from the complexity computation. The second term instead accounts for the updates protocol: every stores performs a given number of updates per period $\leq$ *stores* against other stores; this number is usually low wrt. the number of stores and can be treated as a constant (the parameters **updatesPerPeriod** in our simulation). Every update involves 3 messages being propagated through the network (PULL, PUSHPULL, PUSH). A valid estimate for the wavecast performance for a single wave propagation is given by the average out degree of the graph (of relays) times the number of relays (same as the number of stores in our case); this estimate is provided by the fact that every relay that receives a new wave forwards it to all its neighbours.

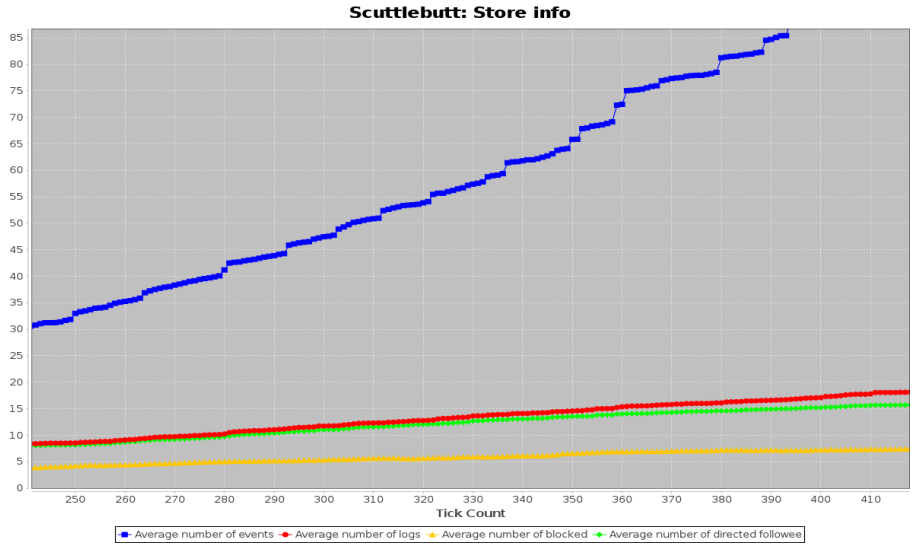


Figure 8: Store info

The formula can then be reduced to

$$O(\text{stores}^2) \tag{2}$$

In fact, the multiplier given by all the factors that we have omitted is a pretty big constant, that with our simulation setup (30 stores) results in a final complexity of $O(\text{numberOfStores}^3)$ for the execution of updates and heartbeats of all stores in a single period. Note that the number of events exchanged as news is not taken into account since we assume that only a single packet with all the information is exchanged per update: this is of course an approximation of what will happen in a real-world scenario.

5.3 Frontier computation in Open Gossip

The computation of a store frontier in the Open Gossip protocol is straightforward and executes linearly wrt. the number of logs contained in the store: the store checks the index of the last event for every store in constant time and returns the set of pairs (logID, lastEventIndex).

5.4 Frontier computation in Transitive Interest

Every time we compute the frontier of a given store in the Transitive Interest protocol we have to take into account the direct and indirect followees of the store. Our current implementation requires parsing the store log to get a view of the direct followees and blocked stores, and then parsing all the direct followees' logs to compute the indirect followees/blocked set. The complexity of

the computation for a given store S is $O(|log_S| + \sum_{s' \in followees_S} |log'_s|)$, given by parsing the log of S and the logs of all the stores that S follows; this is of course $O(stores * avgLogLength)$, which happens when a store follows all logs. If we extend the previous formula to the case when we have to compute the frontiers of all the stores in the system, it becomes $O(stores^2 * avgLogLength)$. This increases linearly wrt to the number of events stored in the logs, and becomes impracticable in a real system, when the number of logs becomes too big. In the SSB authors' protocol implementation a graph of interest relationships is kept at every store and updated on logs updates. To maintain this graph consistent, it is only necessary to add information about new logs, and then navigate the graph to compute the frontier. The complexity of keeping the graph updated are linear wrt number of events in the store, thus constant at every iteration, while the frontier computation can be done by navigating the graph $G = (V, E)$ in e.g. $O(V + E)$, resulting in a lower final complexity of computing all the store frontiers of $O(stores * (stores + 2hopsneighbours))$, where 2hopsneighbours is the number of edges needed to compute the 2 hops neighbours, usually a low number when talking about graphs with sparse connections. For the purpose of our simulation we used the easier implementation that involves recomputing the frontiers from scratch every time, whereas obviously a good extension would be to implement the interest graph.

5.5 Convergence time

One of the metrics we wanted to consider for the protocol analysis is the convergence time, which is the time the system takes to stabilize. We defined this interval as the interval between when the event generation stops (after **eventsGenerationMaxPeriod** periods, as explained in 4.1) to the time when all stores have collected logs of all those they are following.

The convergence time is certainly influenced by various parameters, for example by the follow probability: trivially, if the stores have more followees on average, it will take them longer to retrieve all the news they are waiting for.

However, this parameter in a real case scenario does not depend on the system, but on the users, who decide which stores to follow. For this reason we preferred to investigate the impact of another parameter, which depends on the system configuration and can be adjusted during the setup phase: the number of updates that the system performs each period. As can be seen from figure 9, a higher number of updates per period leads to shorter convergence times, improving the performance of the algorithm. However, it is a necessary tradeoff between the performance of the algorithm and the load of the underlying network, since increasing the number of updates per period results in many more messages being transmitted. Indeed the simulator that we have developed is much slower when this parameter is increased.

An example of the difference in convergence time is shown in Figure 9.

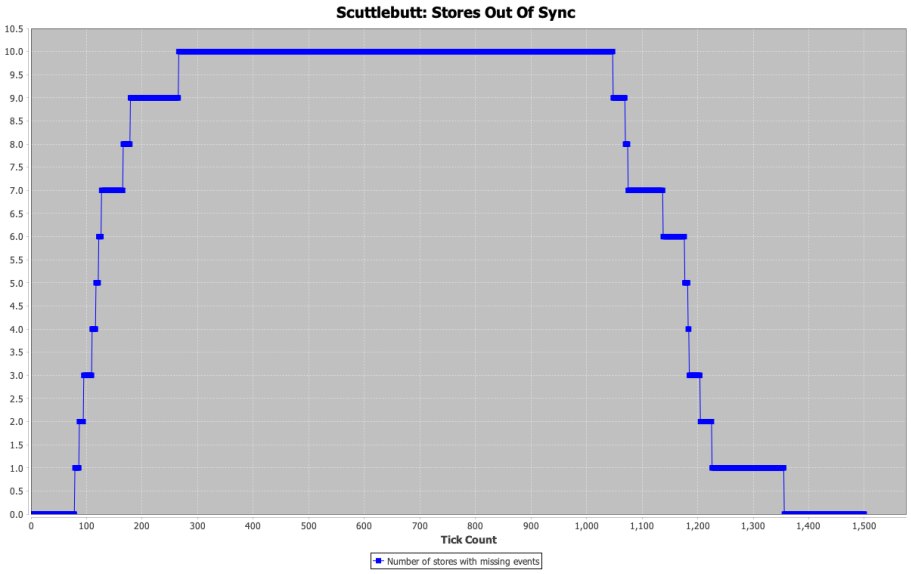
6 Conclusions

At the beginning of this project we wanted to explore many different scenarios, and the opportunities for further work are still multifold; eventually, we focused on ensuring the reliability of the implementation instead of patching it with other untested scenarios. We mention here two of the most promising experiments: testing convergence time when stores are temporarily unavailable and caching blocked/unfollowed logs. The temporary unavailability of stores (or users) is a common scenario in social applications and should be tested to provide reliable implementations; in particular, we wish to prove how robust the gossip protocols are when many of the stores are unavailable, and how long it takes for stores that wake up to synchronize. About caching, blocking or unfollowing some logs might be a temporary operation, depending on users behaviour; moreover, caching logs shifts the state towards synchronization: even if a store stops actively following a given log, in the short term it might still provide other stores with valuable news about the log. The question is how long and how good is caching with respect to the global performance.

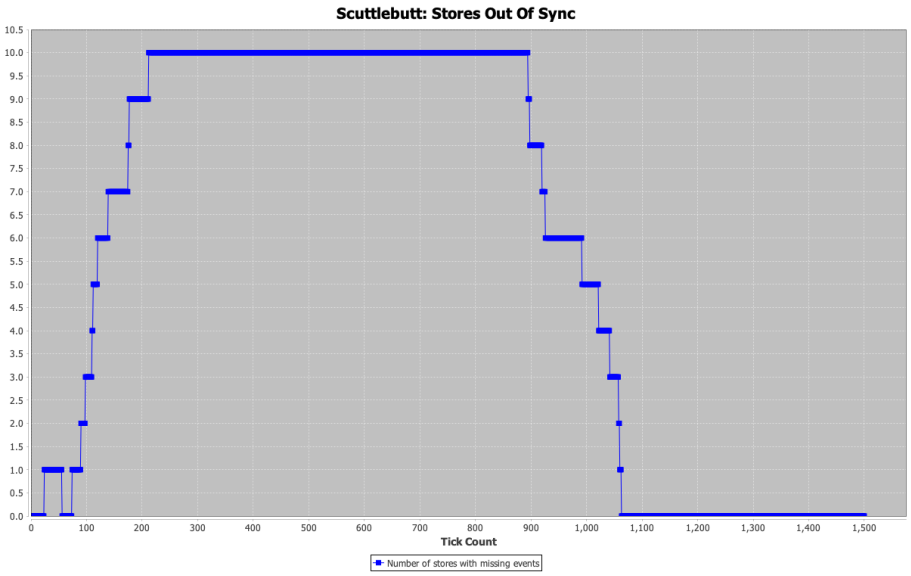
At the end of this work we wish to remark that our implementation provides proof of the Scuttlebutt protocol applicability to wireless scenarios supported by Wavecast, and more broadly that well-designed protocols as SSB can be easily extended to cooperate with other components and applied to a broad set of use cases.

7 How to install

To run the code you can either import the project in eclipse or run **java -jar installer.jar**, where the installer.jar can be downloaded from `https://drive.google.com/file/d/1C4QKHOmLCdd8p6zBUNOJzhC9yVjv8Akk/view?usp=sharing`. Since the project was compiled with Java11, be sure to use a compatible version. When you run the command above, an installer will allow you to install the program on your system: choose a folder and there you will find the program *start_model* that you can use to start the model.



(a) Convergence time with 1 update per period $\sim$ 9 periods



(b) Convergence time with 8 updates per period $\sim$ 5.5 periods

Figure 9: Different convergence times with different update rates

References

[1] C. T. Anne-Marie Kermarrec, Erick Lavoie, Gossiping with append-only logs in secure-scuttlebutt, *International Workshop on Distributed Infrastructure for CommonGood (DICG'20)* (2020). https://doi.org/10.1145/3428662.3428794.

[2] Secure scuttlebutt deocumentation. Available at https://ssbc.github.io/docs/.

[3] Secure scuttlebutt implementations. Available at https://github.com/ssbc.

[4] C. Tschudin, A broadcast-only communication model based on replicated append-only logs, *ACM SIGCOMM Computer Communication Review* **49** (2019), 37–43. https://doi.org/10.1145/3336937.3336943.

[5] Wavecast implementation capraro-micheletto report. Available at https://github.com/antbucc/DS2-2020-2021/blob/FirstAssignment/Capraro-Micheletto/report.pdf.