Distributed Systems 2 - Second assignment

Georgiana Bud
University of Trento
Trento, Italy
214668
georgiana.bud@studenti.unitn.it

Massimiliano Fronza
University of Trento
Trento, Italy
220234
massimiliano.fronza@studenti.unitn.it

Enrico Soprana
University of Trento
Trento, Italy
215039
enrico.soprana@studenti.unitn.it

***Abstract*—In this report we explain our implementation of the Secure Scuttlebutt protocol that acts as a middleware for distributed applications. In particular we compare the "open gossip" variant with the "transitive interest" variant.**

***Index Terms*—distributed systems, middleware, peer, peer-to-peer, network , open gossip, transitive interest, Scuttlebutt, SSB**

I. Introduction

In this report we propose our implementation and analysis of the performance of the Secure Scuttlebutt (SSB) protocol with a network based on unicast, like presented in *Gossiping with Append-Only Logs in Secure-Scuttlebutt* [1].

We have decided to implement both proposed versions: the Open Gossip, which leads to a total replication of the data generated by the participants, and the Transitive Interest, based on follow relationships, which determines that only a subset of the total data generated is replicated, according to a participant's expressed interests.

The chosen programming language was Java, with the aid of the Repast Simphony simulation tool. The analysis were conducted mostly with the use of the Python programming language, and with the Microsoft Excel tool for drawing some of the graphs.

Before explaining the implementation of the protocol itself, we will describe the general structure and architecture of our simulator. In section II we discuss the assumptions that we have made before starting to implement the system, then in section III we present the architecture of the code, by looking closely at how we simulated the real world system through simulation data and how we divided the functionalities on the nodes by using a layered architecture. We will then see the specifications of the protocol implementation in section IV. Section V is dedicated to the visualization part of the simulation, while section VI will present the tunable parameters of the simulation. In section VII we present some analysis that we performed, and we conclude the report with a discussion in section VIII.

II. Assumptions

For the implementation of this protocol, we made some assumptions in order to provide a clean and smooth simulation and analysis of the protocol functionalities. These assumptions are based on the specifications of the paper and on our understanding of the problem. These are:

No lost packets: As specified in the original paper, this protocol is designed to work by using reliable channels like TCP and USB. For this reason in the simulation we assume that a packet is always received and never lost. Moreover, the chosen network topology is based on unicast, so point to point connections, and is fully connected.

Latency: We assume that the latency between the nodes is proportional to their distance in the simulation. Thus, we do not consider any latency that could be added by the underlying network which was abstracted away.

Ports: We decided to allow more instances of the protocol to run on a single physical machine, and we used the public keys of the protocol as ports for the underlying "TCP protocol". This decision was made to abstract away the mapping between an identity and the address at which it can be reached, mapping that a real protocol would have needed to implement. For brevity in the logs the public key/port is referred with a base64 encoding. We assume that given the small number of keys no collision can happen.

Number of protocols per each node: We decided to assume that the number of protocols is distributed like a gamma distribution. This allows us to have a distribution only for $x > 0$ which is mostly concentrated on the mean (as we would expect most networks to have a small amount of participants in our social network). At the same time, it allows a long tail with low probability of high number of participants on the same physical node (think for example of an organization with a large number of participants inside which are connected through a natted connection).
This would also be possible with a Poisson distribution but this wouldn't allow us to specify its variance independently from its mean.

Position of nodes: The nodes are in random positions, without any specific topology for the underlying network, simulating a real network of peers that can connect from anywhere

Data: As the data moved by the protocol is not particularly important, an incremental id was used as dummy data so to make tracking it easier.

III. Architecture

A. Model/Simulation data

In this model the entire simulation is driven by events,

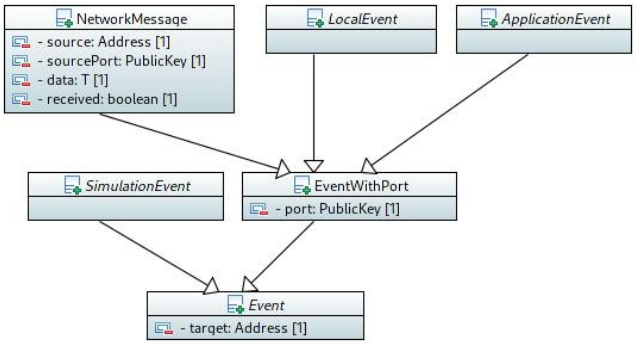

Figure 1. Architecture of `Events`[1]

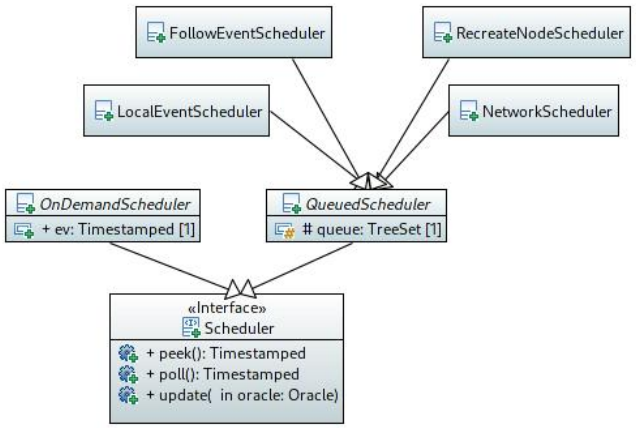

Figure 2. Architecture of `Schedulers`[3]

organized in two main macro-groups:

`Event` This group/class is used to dispatch/target an event to/for a specific machine. With this type we cannot specify that the event should be handled by a specific `Protocol` running on a specific port.

`EventWithPort` This class derives from `Event` and allows us to specify the port to target. In this way we can target a specific instance of the protocol or the application.

These two types of events are abstract classes and thus they cannot be instantiated. They only contain the information necessary to identify the target of the event through the address, in the case of `Event`, and the address and the port in the case of `EventWithPort`. This means that `Event` can only target a particular Machine while `EventWithPort` can be more specific and target a `Protocol` or an `Application` inside it. As these two types of events only represent how a handler is targeted and not what is actually targeted (for example both the application and the protocol can be targeted through the same address and port) these events were extended further to represent that. These derived classes map each event type with the class in which they are handled. These types are:

`SimulationEvent` This type of events is used for the events that should be handled in `Oracle` (the coordinator of the entire simulation). These events change property of the simulation like the number of nodes. The events of this type are: `CreateNode`, `KillNode`.

`LocalEvent` This type of events should be handled inside `Protocol` and in a real system would be local and would not leave the machine (like a timer or a periodic task). These events are used to trigger periodic behaviours like in the case of `TriggerUpdate`.

`ApplicationEvent` This type of events should be handled in `Application` and are used to instruct the application to create a new event or to handle an event going upwards from the protocol. These events are `GenerateEvent`, `Follow`, `UnFollow`, `Block`, `UnBlock` and `ApplicationData`

`NetworkMessage` This type of events should be handled by the protocol and passed through the network. They represent a packet sent from a protocol to another and can contain arbitrary data. In our case this arbitrary data is `ProtocolMessage`.

We can notice that these event types still don't specify what happened nor have no additional information about it.
From this event types some other classes were derived to express what happened and contain any additional data necessary for the handling of the event. For brevity we do not discuss all these "final" events but just show an example.
`CreateNode` is a `SimulationEvent` (which derives from `Event`) which uses the target address inside it to specify the address of the to newly create `Machine` (the machine targeted to be created). As this machine also needs to be initialized with some protocols inside it, it also contains additional information in the form of a list of `Identities`[2] (one for each protocol that will be created).
A more detailed list of all the events and their function is available in the documentation.

B. Scheduler

Schedulers represent a series of events. Each scheduler will serve the events of a specific type. The schedulers can be:

`OnDemandSchedulers` which keep only one event at a time (the next one) and at each step they will get an opportunity to generate a new one through a hook called update. This type of scheduler is used for `SimulationEvents` as we do not need to keep in memory more than one event at a time and they can be generated periodically.

`QueuedScheduler` which keeps an ordered list of events. This is used for `LocalEvents`, `NetworkMessages` and to keep the list of

[1]Methods are omitted for brevity

[2]These contain the public and private key yo be used by a protocol (and consequently also its portsection II)

[3]Most methods are omitted for brevity

temporarily offline nodes to recreate in a second moment.

1) Oracle: The class `Oracle` is a singleton and works as the coordinator of the entire simulation. At each step the oracle will ask the registered scheduler for the earliest event that they can serve, chose the earliest one, remove it from the scheduler and dispatch it appropriately using the mapping previously discussed (in section III-A).

C. Nodes

Each node of the simulation is composed in a layered architecture by a `Machine`, a `Protocol` and an `Application`. This layering allows us to swap each layer with a different implementation of that layer. We used this to implement the two different versions of the protocol while maintaining the rest of the code the same. This way, we can also decide upon start of the simulation which version of the protocol to use, by choosing the appropriate parameter. This layering mimics the layering used in real world systems. The architecture used in this simulator allows us to use multiple protocols stacked upon each other and have multiple protocols running on the same machine in a tree-like structure. We ended up not using this features but it shows the extensibility of the simulator.

The layers are mapped like in the following:

Physical layer - `Machine` and `NetworkScheduler`;
Logical/protocol layer - `Protocol` and its subclasses;
Application layer - `Application` and its subclass;

A more detailed explanation of each layer will follow.

Physical layer: This layer is implemented in the packages `network` but it also linked with the `Machine` class of the `node` package.
The `Machine` class represents the network interface of a node while `NetworkScheduler` represents the underlying network as a whole. For this reason the `Machine` class mostly exposes to the upper layers methods through which they can communicate with other nodes. This is done by addressing the machine through its `Address`, which uniquely identifies it.
`NetworkScheduler` has the task of keeping a list of current messages in transit and to calculate the delay between the two interested nodes, to use once a message is scheduled.

Logical layer: The characteristics and the performance of this layer are the main topics of this report.
This layer is contained in the `protocol` package of which the main class is `Protocol`, an abstract class which has its implementation in two other classes:

- `SSBProtocol` which implements the open gossip "version" of the SSB protocol;
- `SSBTransitiveInterest` which implements the more complex transitive interest "version" of the SSB protocol.

Both implementations communicate through `ProtocolMessages`. The fact that they both inherit from a common parent allows for flexibility at runtime in choosing the correct protocol that needs to be used by the Application, based on the parameter of the simulation.

Application layer: The application is implemented in the `application` package and is the final destination of the data exchanged by the protocols. This level simulates an application in which events and data coming from the protocol layer would be shown to the user, and events from the user, like the generation of data, would be propagated through the network by accessing the interface of the protocol.
This is implemented in `RealApplication` by extending the abstract class `Application`.

Annotations and the `EventHandler` (and `MachineEventHandler`): To aid with the implementation of the handlers for the events at the different layers, the EventHandler class was created to automatically provide the basic blocks for the event handling. Specifically this class provides a handle function which, when called, automatically looks for a function inside the class which expresses its ability to handle a specific type of event by being annotated with the corresponding annotation.

The annotations map the previously introduced event types:

SimulationEventHandler - A function annotated with this annotation will be called whenever a `SimulationEvent` of the same type specified in the cls parameter is received.
NetworkMessageHandler - A function annotated with this annotation will be called upon receival of a `NetworkMessage` containing data with the same type specified in dataCls. In this case the type specified by dataCls can be anything (as NetworkMessage accepts any type as data).
LocalEventHandler - Similarly to SimulationEventHandler the function with this annotation will be called whenever a `LocalEvent` of the same type specified in the cls parameter is received.
ApplicationEventHandler - Same as LocalEventHandler but with `ApplicationEvent`s.

The EventHandler class was extended further, by `MachineEventHandler`, to allow an upwards and a downwards connection to another `MachineEventHandler`. The `Machine`, `Protocol` and `Application` classes all inherit from this class. Message passing between the layers can be achieved by using the corresponding upcall and downcall functions. If needed, the layer can use getDown() or getUp() to obtain respectively the list of layers above it or the single layer below it to call a specific function. For example this was used to call the follow function of the SSB transitive interest protocol when a `Follow` event reaches the application.

IV. Protocol Implementation

In order to implement the protocol, we started from the specification given by the paper. Just as a recap, we had

to implement a gossip protocol, based on a replicated **authenticated single-writer append-only** logs, providing an eventually-consistent replication of **events**[4], which are the data generated by participants.

A. Identity Management

In order to provide authentication, and possibly confidentiality, participants and their SSB identities are identified by a public key, corresponding to the private key that they possess. To achieve this, we have used the libraries provided by `java.security.KeyPairGenerator`, `java.security.PublicKey` and `java.security.PrivateKey` to generate RSA key pairs, and a custom `Identity` class that contains the public and the private keys of an identity. These are distributed when a protocol instance is instantiated in the `Oracle`, and so Oracle acts as a central, trusted authority which distributes user credentials.[5]

A protocol instance running on a machine is uniquely identified by its `Identity`, and the peers can reach each other through their public keys. Moreover, the public key of an Identity is also saved in the `SSBEvents` generated by that user, linking the Identity with the contents that it generated.

In order to simplify the simulation, we have assumed that Identities can communicate with any other Identity, which they obtain randomly by asking the `Oracle`(the Oracle also acts in a sense as an abstract DNS).

B. SSBEvents, SSBLogs and Stores

The data generated by a participant at the application level, contained in an `ApplicationData`, is saved as a `SSBEvent` at the protocol level. An SSBEvent contains the following information, as specified in [1]:

- idKey, the public key of the creator;
- previous: the signature of the previous event in the log of the creator (the hash can always be computed afterwards, so we decided not to include it);
- index: the position of the event in the log;
- content: the `ApplicationData` received from the application;
- signature: the cryptographic signature of the other fields, obtained using the private key of the generator. The hash of the contents is first computed and then signed via RSA. By signing the event, we ensure that its contents are immutable and that it was really originated by the holder of the given public key.

`SSBEvents` generated by one identity are saved in the generator's own log, which we called `SSBLog`, identified by its public key. This class implements the authenticated single-writer log operations indicated in Table 1 of the paper [1][6]. The goal of the protocol is to correctly replicate the `SSBLogs`.

This class also contains methods to compute the hash of the contents of `SSBEvents`, to sign them and to verify the signature. The computation of the hash and of the signature are used when appending a local event, to ensure the single-writer property. The signature verification is used when checking the events created remotely and that need to be added to the local store.[7]

A `Store` is a set of `SSBLogs` maintained locally. The corresponding class implements the operations indicated in Table 2 of the paper [1].[8] The store organizes the `SSBLogs` in a `HashMap`, where the public key of an identity is associated with its `SSBLog`. Thanks to this organization, we obtain faster operations of lookup, addition, removal and update of elements. The `Store` has its own identity, as specified for the transitive interest protocol. The own identity of the store is used to create a local log, for the events generated locally. From the `SSBlogs` contained in a `Store` we can compute a **frontier** of the latest known indexes of the logs in this store. We have used a `HashMap` computed dynamically for this purpose.

The `Store` directly uses the API provided by `SSBLog`, and it provides in turn methods that the implementation of the protocol can call in order to perform operations on it. The operations that it provides are the same for both `SSBProtocol`, the Open Gossip implementation, and `SSBTransitiveInterest`, the Transitive-Interest Gossip implementation, which take care of the communication between different stores and with the application that stays above.

C. Open Gossip

We have implemented the Open Gossip protocol as a two parties' protocol, inside the `SSBProtocol` class. It is connected with an `Identity`, for which it maintains a local `Store` with a local `SSBLog` initialized upon the initialization of the protocol itself. It acts as a middle layer between the Machine and the Application. In fact, it sends to the application events intended for it, received from the simulation, and the content of new `SSBEvents` that it receives from peers after an update. It also creates local events to store the application data locally, in the own log of its identity, upon a downcall from the application.

Its main functionality is that of doing the update step when connecting with a peer, in order to exchange new ssb-events,

[4]In order to differentiate these events from the simulation events that we previously described, we have chosen to call them `SSBEvents`, and this will be the name that we will use from now on.

[5]Since we are in a simulated environment, we did not consider signing certificates or setting up a separate PKI, but we only distributed key pairs. In a real world scenario, the management of keys has to be determined by the organization deploying the application, and this is beyond the scope of our implementation.

[6]In the code, the methods have slightly different names and signatures than the ones indicated in the paper, because of the specification of the Java programming language and for clarity. Please check the documentation of the code for the precise correspondence.

[7]For simulation purposes, we only check the signature of the first event that we receive in a list. In a more dynamic environment, we would add the check on every single event.

[8]The signature and the precise names of the methods are here as well different, for the specifications of the Java language and for clarity. The correspondence with the indicated methods is given in the documentation.

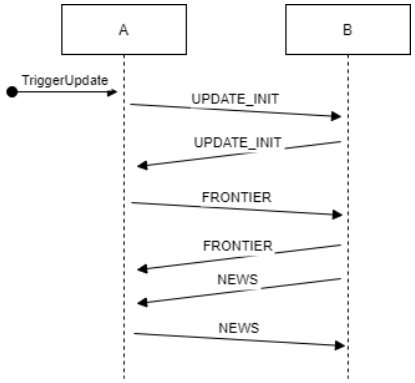

Figure 3. Message exchange between two peers during the update step.

created locally or by other identities. The update is initiated upon the receival of a simulation event, the TriggerUpdate, handled in the corresponding annotated method. The peer with which to connect is chosen randomly by asking the Oracle to pick a public key and its physical address, among the ones present in the simulation. The peer may be anyone, even another protocol running on the same machine (e.g. simulating a protocol instance running on a machine connecting with a protocol instance on a USB support). This update is triggered at the beginning by the Oracle, and then it is scheduled locally by the protocol itself, with the period specified with the corresponding simulation parameter, which we will explain later.

We have considered that, in a real system, the update step of more pairs of peers may happen simultaneously, and that the same protocol instance may be in two different "update connections" at the same time, e.g. one initialized by itself and another initialized by someone else[9]. Thus, we have decided to account for the possibility of concurrent connections and temporarily save the ongoing connections in two lists, the ones initialized locally and the ones that someone else has initialized. We also save information about the exchanged ids and frontiers with the peers in these connections, and we check for the correspondence of messages in the different phases of the update. This way, we ensure that the update is stateful and we also account for malicious entities which may send news to their peers without being asked for, e.g. with the aim of denial of service attacks.

D. Update step

We made a clear distinction of the various steps of the update, dividing it into ids exchange, frontier exchange and news exchange, as to map closely the implementation of Algorithm 2 and Algorithm 1 [1], as explained in the following paragraphs and shown visually in fig. 3.

For the information exchanged in these steps we used a custom class, `ProtocolMessage`, which can contain three types of information: a list of ids, a frontier and the requested news. The `ProcolMessage` is contained inside a `NetworkMessage`, which provides its delivery to the destination. We used three separate types of `ProtocolMessage`:

- UPDATE_INIT, in which we send only the ids,
- FRONTIER, in which we send the frontier,
- NEWS, with which we send the news[10].

These types are important as they allow us to check what type of data arrived to the destination and to handle it accordingly. For clarity, we explain here the details of how an update is performed.

The update is initiated by the node to which a `TriggerUpdate` is sent. This node, call it A, picks a random peer and sends its ids to it, in a `ProtocolMessage` of UPDATE_INIT type.

The node which received the UPDATE_INIT message, call it B, first of all prepares its own known ids and sends them to A, and then handles the message received as depicted in line 3 of algorithm 2 of the paper, creating empty logs for the missing ids. Peer A receives the ids and handles them accordingly (line 4 of algorithm 2), and then sends its frontier to B. In this frontier, also the new identities are listed, with a value of -1 as last known index, meaning that the list is empty. B receives the frontier and sends his own frontier to A, and then handles the received frontier by collecting the newer messages with respect to it, and sends them back to A. A then receives the frontier of B and can compute the news and send them back.

In the peers, the news are added to the corresponding `SSBLog` by calling the `update` method on the `Store`. Different checks are performed before calling it, in order to make sure that the news correspond to the ones requested. Moreover, for the first message in a list of news that we receive, we check that it was signed by the corresponding entity and that the signature corresponds to the message contents, to ensure the authenticity and integrity of the message. Since the methods proposed for updating a store are done on lists, we did not consider checking the authenticity and integrity of every single message received, because there were no specifications about it. In any case, in an application which would be deployed in a real system, we would add this check to make it more secure. We note that the check of the signature of every message may slightly slow down the processing of the news, as cryptographic operations are quite expensive.

As last operation in the update step, the news that were added to the store are sent up to the application, in order to notify the user of new ssb-events.

E. Transitive Interest

The second version of the protocol is based on a similar logic of message exchange, and we used the same `ProtocolMessages` to exchange the frontier and the news.

[9]If a protocol instance is already doing an update initiated by someone else, and it is its turn to start an update, we have added a check such that it cannot chose the same peer with which it is already communicating, because they would not exchange any other news that those already exchanged.

[10]We did not consider the size of the data packets that need to be exchanged: we abstracted it away as this can be taken care by TCP fragmentation in a real world scenario.

In this protocol, only the SSBLogs of identities that are in the transitive interest of the corresponding instance are replicated, according to the specifications in the ssb-friends module [2][11]. The difference with respect to the Open Gossip is that the UPDATE_INIT step and ids exchange do not take place, because the missing ids are computed locally based on the new information about the followed identities that arrived to the protocol from the application and from the peers. The way of dealing with frontier and news is the same as previously explained for the Open Gossip implementation.

The new operations introduced by this implementation, (table 3 of the paper) are follow, unfollow, block and unblock. We decided to use some application events, already mentioned in section III-A, in order to simulate a user action, who decides to follow or to block someone.

For this reason, we have written a custom class, UserActions, which extends the Id (for the ordering and tracking of the information), and which contains the four given actions. When the corresponding event arrives to an application from the simulation, a random identity is chosen to be followed/unfollowed/blocked/unblocked, and an object of this class is instantiated with the specified user action. Various checks are done when choosing the target identity, in order to obtain the desired, logical, behaviour (e.g. we ensure that we ask to follow only someone who we do not follow yet). This content is sent down to the protocol to be saved in the SSBLog. The user actions are also processed by the SSBTransitiveInterest to maintain updated sets of locally followed and blocked identities, which form the "first level" followed and blocked identities, those for which the user wants to see the news.

We have decided to maintain also a "second level" of followed/blocked identities, which are the transitively followed and transitively blocked identities. The news for these identities are handled according to the already mentioned ssb-friends module, as explained also in the next paragraphs, but are not directly sent to the application. Since they are held in the store, the application may access them locally, but we have decided not to implement the corresponding code for this, since it goes beyond the scope of the simulation.

For each of the peers that a participant directly follows (i.e. his friends) there is a list of identities that those peers follow and block. These are saved in the followed2nd and blocked2nd hash maps. These two lists are updated when messages coming from the friends arrive to the protocol. When a follow message from a friend is registered, the identity newly followed by this friend is added to the corresponding list connected with my friend in the hash map. When an unfollow arrives, the list is updated by removing the identity which is unfollowed. Similarly for the block and unblock operations.

The new interest (follow) or disinterest (block) of my friends are registered when the corresponding message first arrives. The protocol will start gathering information about the newly added identities only from the next update. When preparing the store before the frontier exchange, it will take into account the local followed and blocked identities, and the transitive followed and blocked identities, by checking the lists and adding/removing logs from the store accordingly. At every update the sets are recomputed based on the information that was exchanged at the previous update, in case the situation has changed. Specifically, for the transitive follow, the store will replicate ids followed by at least one of the participant's friends, excluding the identities that were blocked by the participant. For the transitive block, the protocol will block those identities that are blocked locally and those identities that are not followed by itself or by one of its friends, and which are blocked by at least one of the friends (e.g. if a friend follows and one blocks the same identity, the identity is not blocked).

We have decided to stop at the second level of interest, because of lack of clarity in the relative explanation of the paper. In any case, the implementation can be extended to allow also third level follows and so on. This implementation could be further extended in order to obtain the resistance against sybill attacks and spams.

[11]We simplified the computation of the followed and blocked sets, by using locally updated lists in which we hold the interest of the friends, instead of the proposed Dijkstra algorithm.

V. VISUALIZATION

The visualization-related part of our project has its main management performed by the *DisplayManager* class. This is initialized in the *Oracle*, and the various graphic methods are called from this and the *RealApplication* class.

For each step, we first clear the space from all graphic elements previously created by removing them from the context, or from the network in case of the edges; then we proceed to analyze the *NetworkMessage* passed to *DisplayManager*, dividing the outcomes in this way:

- if one between destination or source of this packet is dead, it will disappear from the space and there will not be any graphic element;
- if source and destination are the same, meaning there are two identities communicating inside the same machine, only a light-blue triangle will displayed on that node;
- message content of *UPDATE_INIT* type: only happens when using the Open Gossip communication protocol; expresses the sending of ids from one node to the other at the beginning of the update. It is represented as a direct green edge;
- message content of *FRONTIER* type: happens both with the Open Gossip and the Transitive Interest protocol; stands for an entity sending its frontier to the other node involved in the exchange. It is a light-blue direct edge;
- message content of *NEWS* type: used to identify an exchange of events in the final phase of the update between nodes. As the previous case, is part of both the communication protocols. It is a blue direct edge;

We do not represent only network messages that are exchanged in the network, but also various other events that happen locally on the nodes. For example, when a new local event

is created, an orange plus sign ("+") appears on the node producing it. This is called both from *SSBTransitiveInterest.java* and *SSBProtocol.java*. In *RealApplication.java*, whenever a follow/unfollow/block/unblock event is handled, something will be displayed. For example, an orange direct edge appears from a node following another, and the same edge colored in red will express a block. Unfollow and unblock events display the same edges, but with a red cross on the node that is being unblocked/unfollowed.

Several classes were created to represent edges, crosses and the content triangles, each one as a custom element.

To change colors and weights in a dynamic way, we created two style classes that implemented the *EdgeStyleOGL2D* interface. At the beginning, we wanted to use *DefaultEdgeStyle2D*, which then turned out to be used in Repast Simphony 2.7, but deprecated since the 2.8 version. These classes call methods from *DisplayManager* to get the proper values of the objects they represent, and since we kept having problems in making Repast use them instead of the *DefaultEditedStyleData2D* counterparts, we had to manually edit the */Ds2Assignment1.rs/repast.simphony.action.display_1.xml* file to make it work. Also, edges needed a class implementing an *EdgeCreator* to instantiate them.

For crosses, triangles and the nodes themselves the situation was easier, because their colors and sizes were not dynamic, so to describe their properties the *DefaultEditedStyleData2D* was enough. Style files for these objects were stored in the */Ds2Assignment2.rs/styles/* folder.

Nodes are represented as small blue circles, under which there are labels identifying them.

Speaking of parameters, only DISPLAY_SIZE affects the visual part of the simulation, by re-sizing the space dimensions.

VI. Parameters

A complete list of parameters follows:

- OUTPUT_PATH The path at which the logs are written. A folder with as name the current time in ISO8601 is automatically created to contain all logs. The default value is "results".
- CONSOLE_OUTPUT If true the logs are written also in stdout. The default values is true but for long simulations it is recommended to set it to false as it gives a small performance bump.
- PROTOCOL_TYPE The protocol to run on the machine. It can be "Open gossip" or "Transitive interest".
- MEAN_CREATE It is the average time between the creation of two nodes. By using the value 0, the creation of nodes can be disabled obtaining, when also MEAN_KILL and MEAN_GO_OFFLINE are 0, a static network. By default it is 0 seconds
- VAR_CREATE Standard deviation of the time between the creation of two nodes using a normal distribution. By default is 0 seconds.
- MEAN_KILL The Average time between the removal of a two nodes. By default is 0 seconds
- VAR_KILL The Standard deviation of the time between the removal of two nodes. By default it is 0 seconds.
- MEAN_EVENT It is the mean time between the creation of two events. By default it is 3 seconds
- VAR_EVENT The standard deviation of the time between the creation of two events. By default it is 0.5 seconds.
- MEAN_FOLLOW The mean time between the creation of two random follow events. By default it is 100 seconds. The follow event is sent to a random peer, which randomly chooses to start following someone who it does not follow yet.
- VAR_FOLLOW The standard deviation of time between the creation of two follow events. By default it is 5 seconds.
- MEAN_UNFOLLOW The mean time between the creation of two unfollow events. By default it is 150 seconds.
- VAR_UNFOLLOW The standard deviation of the time between the creation of two unfollow events. By default it is 5 seconds.
- MEAN_BLOCK The mean time between the creation of two block events. By default it is 250 seconds.
- VAR_BLOCK The standard deviation of the time between the creation of two block events. By default it is 10 seconds
- MEAN_UNBLOCK The average time between the creation of two unblock events. By default it is 300 seconds.
- VAR_UNBLOCK The standard deviation of the time between the creation of two unblock events. By default it is 10 seconds.
- MEAN_GOOFFLINE The average time between two GoOffline events. By default it is 0 seconds(disabled).
- VAR_GOOFFLINE The standard deviation of the time between between two GoOffline events. By default it is 0 seconds(disabled).
- MEAN_RECREATE_DELAY The average time after which a temporarily offline node gets back online (Uniform distribution). By default is is 0 seconds (disabled).
- VAR_RECREATE_DELAY The standard deviation of time after which a temporarily offline node gets back online. By default it is 0 seconds (disabled).
- INITIAL_NODE_SIZE The number of nodes created at time 0. By default is is 25 nodes.
- MAX_NODE_SIZE The maximum number of nodes allowed in the network. It only blocks new nodes from being created only after 0 so it doesn't influence INITIAL_NODE_SIZE. By default it is 100 nodes.
- MEAN_IDENTITIES_PER_NODE The average number of identities per node (Gamma distribution[12]). This number cannot be ≤ 1. By default it is 2 identities.

[12] An interactive jupyter notebook to more easily set this parameter can be found in `./scripts/notebook/gammaInt.ipynb`

VAR_IDENTITIES_PER_NODE The standard deviation of number of identities per node. This number cannot be 0. By default it is 5 identities.

MIN_FOLLOWED_PER_NODE The minimum number of follows assigned at each node during initialization (using a uniform distribution). By default 15.

MAX_FOLLOWED_PER_NODE The maximum number of follows assigned at each node during initialization (using a uniform distribution). By default 30.

STOP_TIME Time at which to automatically stop the simulation. By default it is 5000 seconds.

PROPAGATION_SPEED Speed of propagation of the message. By default it is 1198837 screen/second. This value can be considered as having of a 250m x 250m square in which the information is propagated at the speed of light. For the transitive interest variant given its different use case we decided to use a bigger square of 3500km x 3500km (roughly the width of Europe) which gives us a propagation speed of 85.63121 (always using the speed of light as the simulated propagation speed).

PROCESSING_DELAY The delay needed to process and then transmit a response. By default it is 0.01 seconds.

UPDATE_INTERVAL Interval after which a new update is triggered. By default it is 30 seconds.

DISPLAY_SIZE Size of the display (Changes the size of the UI elements).

Disabling functionalities: For the parameters MEAN_CREATE, MEAN_KILL, MEAN_EVENT, MEAN_FOLLOW, MEAN_UNFOLLOW, MEAN_BLOCK, MEAN_UNBLOCK, MEAN_GOOFFLINE the value 0 can be used to disable that functionality. We can notice that by putting to 0 MEAN_CREATE, MEAN_GOOFFLINE and MEAN_KILL we obtain a static network.

Generation using normal distribution: The parameters for create, kill, follow, unfollow, block, unblock and goOffline use a normal distribution. By doing this we can achieve an uniform rate[13] of events while allowing them to be not perfectly periodic, which would be unrealistic .

VII. Analysis

In order to perform the analysis, we collected logging data from different modules of the program, specifically from `Oracle`, `SSBProtocol`, `SSBTransitiveInterest`, `Store`, `RealApplication`. This data allows us to know when the various events happen, and when application content is generated and spread to the peers. A specific `Logger` class was used for this reason, in order to easily write and categorise logging data in a folder under the path specified by the parameters.

This data is already divided into different output files depending on the node on which an event is logged, and can be easily filtered to obtain only the information needed for the specific analysis. The filtering operations were done in bash, and the further processing of the output in Python.

[13]As an example of this see the jupyter notebook `./scripts/notebooks/normalDelay.ipynb`

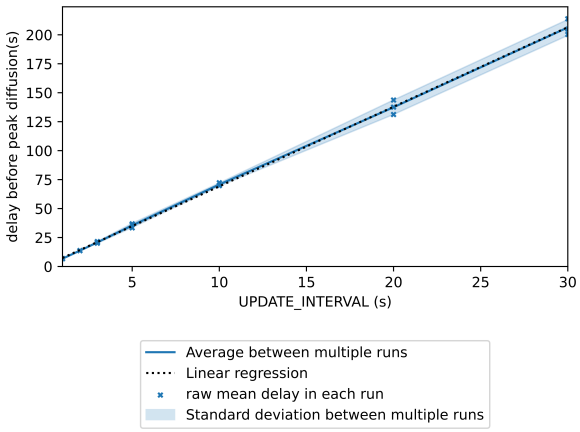

Figure 4. Diffusion delay vs UPDATE_INTERVAL

A. Delay vs update time

This analysis was performed on the Open Gossip version of the protocol, where generated application content reaches all the other peers.

We wanted to understand how the update interval influences the delay, (wrt. the creation of the id) with which ids are received.

To do this we varied the parameter UPDATE_INTERVAL using as values 1, 2, 3, 5, 10, 20, 30 in a static network of 125 nodes. We think this can be a useful analysis as it can give us insights on how to configure the protocol so to use the least amount of bandwidth given the maximum delay that our application is able to tolerate.

To do this we obtained from the logs the delay with which every id sent reached its maximum diffusion and averaged all the delays (so we averaged the delay of each single id together). The ids considered, were then filtered to discard all the ids generated in the last 150s. This was needed as these last events didn't have enough time to completely propagate through the network and would thus skew the results away from the actual result. Ids were generated every 3 seconds in average, as indicated by the default value of the corresponding parameter. To make sure that these values were not given by "noise" due the random processes we decided to do 3 runs for each UPDATE_INTERVAL values.

In the graph (fig. 4) we can see that the behaviour is linear both when the update interval is shorter than the event generation rate and when it is longer and both have the same slope. We then used linear regression on the average between the runs to obtain the slope (6.869) meaning that with our configuration the network needed on average 6.869 rounds to distribute the message to all nodes.

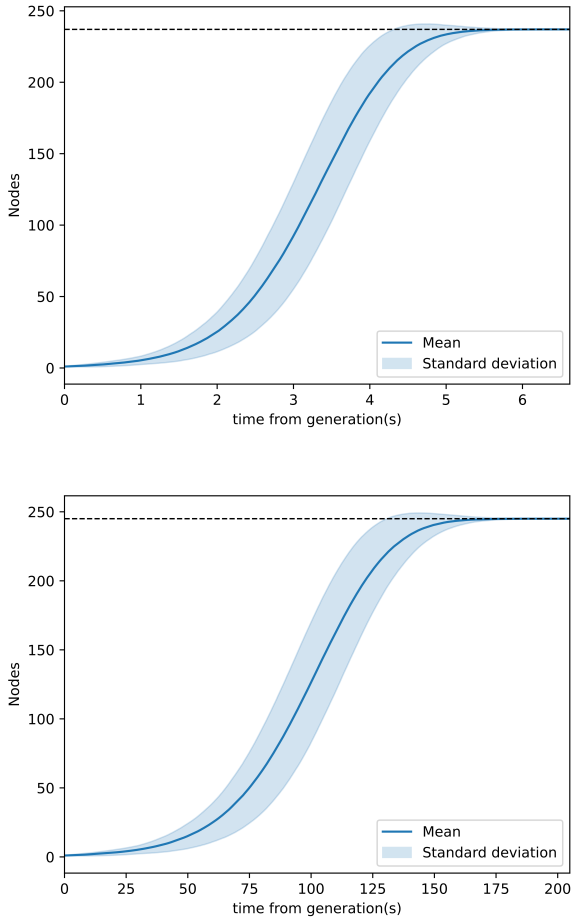

Figure 5. Nodes reached vs time from generation for UPDATE_INTERVAL 1 and 30

We can also see in fig. 5 that no real difference (other that the scale) in the behaviour of the distribution between the two extremes used (1 update per second and 1 update per 30 seconds).

B. Delay vs number of nodes

Another important point for us was understanding how this version of the protocol (open gossip) scales based on the number of nodes (and protocols) in the network.
To do this we varied the number of initial nodes, using the values 25, 50, 75, 125, 250, 500, 750 , on a static network with a UPDATE_INTERVAL of 30 (the default). In this test we used the same method used in the previous test. As we can see in fig. 6, increasing the number of nodes (and protocols) in the network causes a small increase in the time needed for a message to reach all the nodes but the increase gets smaller and smaller with a higher number of nodes. In particular, we can see that the time increase of the mean between 500 and 750 is very small, 2.173s, especially considering the variance of the standard deviation between the different runs (5.027s for 500 nodes and 3.407s for 750). As the number of protocols

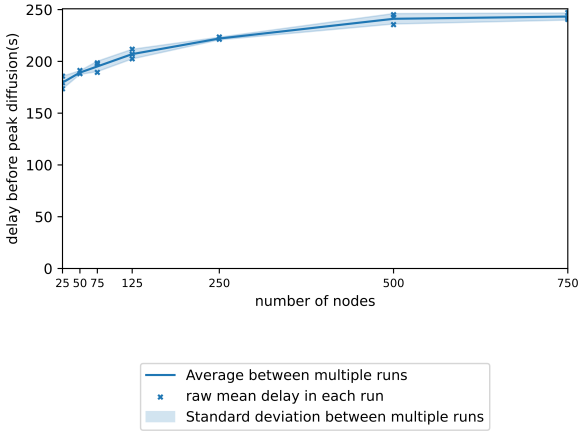

Figure 6. Diffusion delay vs number of nodes

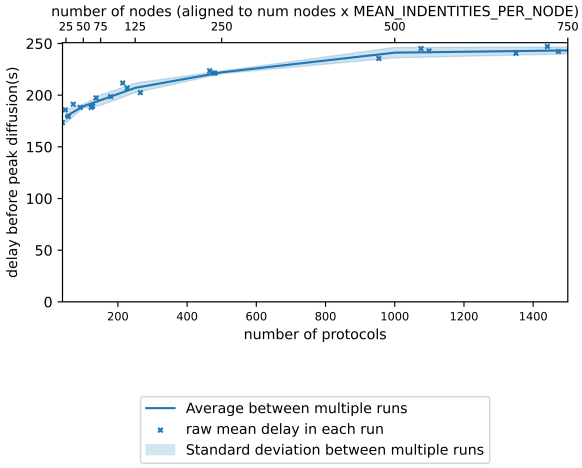

Figure 7. Diffusion delay vs number of protocols

assigned to the node is a random process, and this could introduce some noise, in the test we also show the diffusion delay vs the number of protocols in fig. 7. As we can see the raw data still follows (even if a bit more loosely) the curve drawn by grouping the different runs by the number of nodes. This test showed us that the protocol can scale with a high number of nodes with minimal impact on the network.

C. Diffusion time in Transitive Interest Graph

The following analysis were performed on the Transitive Interest version of the protocol.

We have considered the diffusion time of application data, the ids, from when it is generated to when it arrives to the identities that follow the generator of this content. In this subsection, we only consider the followers that the generator has at the moment of the creation of the content. Another analysis that can be done is that of considering the time that a new follower takes in order to catch up with the old content.

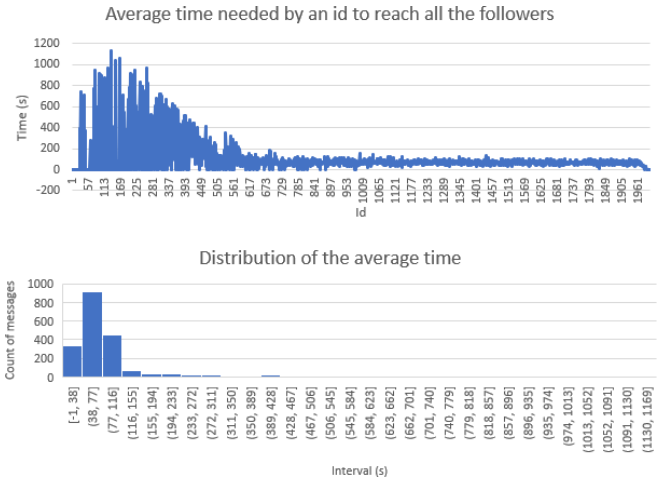

Figure 8. Average time needed by every application data to reach the followers, and its distribution, in the case of no initial followers.

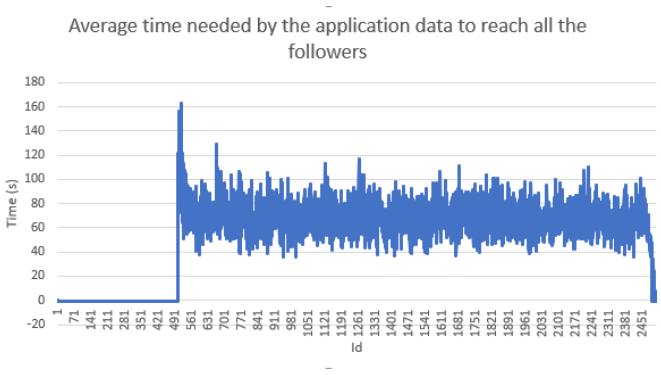

Figure 9. Average time needed by every application data to reach the followers, when some following relationships are given at the beginning.

For time constraints, we were not able to perform this analysis that we had in mind.

1) Small network: During the implementation, we have first of all initiated the network with nodes which do not follow anyone, and so they only start following after a `FollowEvent` arrives from the simulation. In this case, content is not replicated at the beginning, because there are none or few followers when it is generated. We then improved the implementation such that a part of the social network of follow relationships was already initialized, and content could be exchanged since the very beginning. This also simulates a more natural behaviour, in which a user which connects to a social network for the first time already adds some contacts because they know them personally. We have analysed the difference in the diffusion time of content in both cases.

The network used for this initial analysis contained 25 nodes, with 47/48 total identities on these nodes. The parameters were the default ones, a part from the following: the MEAN_FOLLOW was set to 15 seconds, and VAR_FOLLOW to 5, in both simulations; the network was static, with no UNFOLLOW or BLOCK events. These needed a more detailed analysis, for which the time given was not enough. The propagation speed considered here was 1198837 screen/second, simulating a small square of 250x250m, typical of a LAN.

Case 1: No initial followers: In this case, the protocols do not know about each other and do not start to replicate the content of their store until they start following each other.

In fig. 8 you can notice on top the plot of the average time needed by every id to reach all the followers, and, below, the histogram that summarizes the distribution of these times. We notice that the ids are ordered in time, so the high diffusion times at the beginning are due to the fact that there are few followers, and many identities remain “isolated”, and their data takes also more than 1000 seconds to reach the interested entities. This is due to the fact that few protocols replicate the stores, since the transitive interest graph is quite empty.

After a high peak in the diffusion time at around 150 ids generated (which, given a mean of generation events of 3 seconds, corresponds to 450 seconds from the beginning of the simulation), the average time starts to decrease and stabilizes after around 600 ids in the range 30 - 100 seconds. At this time, which corresponds to 1800 seconds since the beginning, given a mean of ’follow’ events happening every 15 seconds, there were in average 1800/15=120 follow events, which means that the network started to be more connected.

The distribution of these diffusion times resembles a Rayleigh distribution, and we can see how most of the data is concentrated in the initial range, from 0 to 116 seconds, meaning that most of the news propagate in up to 4 update intervals of 30 seconds. There is nevertheless a long tail, which accounts for the initial events that needed a long time to arrive to every follower.

In any case, we think that the network converges quite fast in the sense that, starting from a random topology of 47 identities, where no one follows anyone at the beginning, and content is yet generated, it is sufficient that everyone starts following 2 or 3 other identities (120 follows/47 participants) in order to start replicating the content faster.

Case 2: With initial followers: In this simulation, each peer was set to follow between 7 and 15 other identities since the beginning, and then dynamically follow someone based on the `FollowEvent` arriving from the simulation.

Figures 9 and 10 show the results of the same script used for the graphs in fig. 8. A major difference that can be seen is the presence of messages with a diffusion time of 0 (flat line in 9 and first bar in 10). These are only due to the fact that the initial followers are initiated via `FollowEvents` that generate application content of type `UserActions`, which was also exchanged during the update step. All these events happen simultaneously at time 0, so their diffusion, which happens afterwards, is not considered.

We can see that the majority of the average times stays in the interval of 40 to 100 seconds, which is not very different with respect to the previous case, after the stabilization of the network. The distribution instead resembles a Gaussian, which has its peak in the interval [62.2, 78], corresponding to two update intervals.

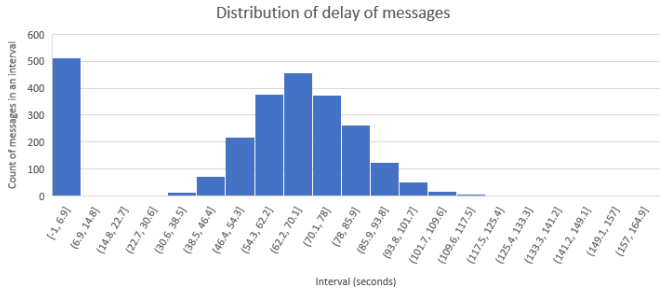

Figure 10. Distribution of the time needed to reach all the followers, when some following relationships are given at the beginning.

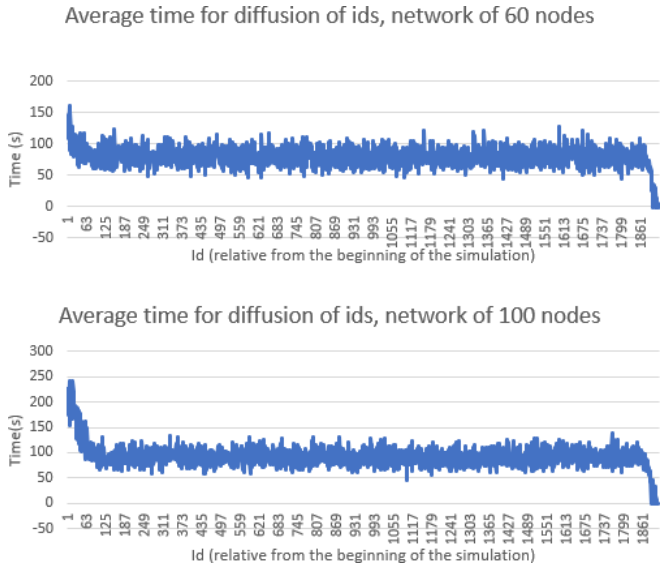

Figure 11. Average time needed by every application data to reach the followers, in a network of 60 (top) and 100 (bottom) nodes.

2) Bigger network: The two graphs in fig. 11 were drawn from the analysis performed on two static networks with respectively 60 and 100 nodes, with some followers given at the beginning. The number of total identities, so also protocol instances, is not equal to 60 (respectively 100), but more, as described at the beginning with the total identities per node. For the 60 nodes, the total identities were 93, while on the 100 nodes there were 183 identities. All the parameters were set to the default value, apart from MEAN_FOLLOW set to 20, VAR_FOLLOW to 5, INITIAL NODE SIZE to 60 (respectively 100), MAX_NODE_SIZE to 120, PROPAGATION_SPEED to 85.63121 (to simulate a network in a square of 3500x3500km) and default random seed to 860,511,038.

For visualization purposes, the ids created at time 0, containing the initial followers, are not shown.

Both graphs show a similar distribution of the diffusion times, that behave, as in the case of the network of 25 nodes, as Gaussian noise, centered at different means. In the network with 60 nodes, most of the averages stay above 50 seconds, and in the network of 100 nodes we see another increase, that indicates that the bigger the network, the more time it will be needed for the news to propagate.

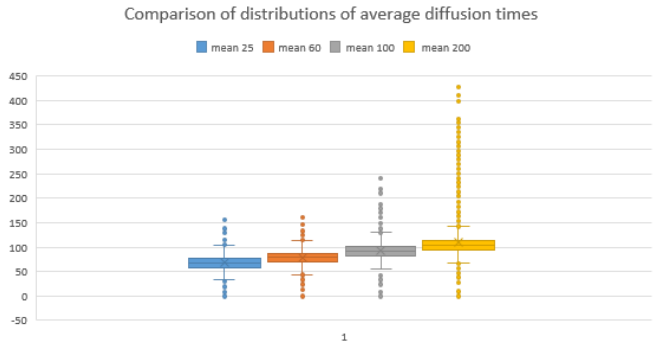

Figure 12. Comparison of the distributions of the delays with which ids arrive at the followers, for networks of different size.

We have tried to see the behaviour of the protocol also for a network of 200 nodes. We have tried to run a 500 nodes network, but the simulation and the computation of the results took too much, so we have decided to discard the results for the analysis. The time to run the simulations and to run the scripts were increasing, so we have stopped at this size.

In fig. 12 a comparison of the means taken by averaging all the values in the series for the network with 25, 60, 100 and 200 nodes is shown. Here, the increase in the average time is more visible, even if it is just a small one.

This result indicates that the increase in delay is quite small when the network size increases also in the case of the implementation for Transitive Interest, which is in line with the results of the analysis performed for Open Gossip, as explained in VII-B.

These analysis show that the number of followers that an identity has since the beginning determines the speed of replication of the content, because of the random nature of picking a peer during the update. Intuitively, if an identity has more followers, and if peers follow more identities, the probability of picking up a node that has content which is interesting for you is higher, leading to a faster replication of the content. We also notice that the physical distance that we simulate does not influence much the propagation of news, because the high speed of Internet connections allows for fast transcontinental transmission. The update time and the number of followers are much more important parameters in determining the diffusion time.

VIII. Open issues and conclusion

We wanted to make some considerations about the limitations of the protocol, that we have encountered during the analysis, and about the open issues that we think it would have been interesting to address but for which we lacked the physical time.

First of all we consider that this is a good protocol since it is application agnostic and quite decoupled from the type of usage that it may serve for. It could be nevertheless further expanded to add more security and trust.

We also think that this protocol is not the best for real time applications, that need to exchange the content directly when

it is generated. Moreover, the random selection of peers during update is not the best in terms of performance, and as we have seen can lead to diffusion times quite high (more than 1100 seconds in the case of transitive interest). Some proposals that we have, and that we wanted to implement, were to chose differently the peers with which to perform the updates, e.g. based on distance or based on the follow relationships, or also to start an update directly when a content is generated, in order to "inject" it into the network as soon as it is created. Another optimization we thought about was to have a different update interval for protocols inside the same machine. A series of updates could even be trigger immediately upon an update so to keep all protocols inside the same node all updated at the same level. This would mean that the delay of the diffusion would depend only on the number of machines and not on the number of protocols. If such an optimization would be made we would expect an improvement similar to the mean amount of protocols inside each node. It would have been interesting to add these improvements, and to study their effect also on the social graph that they would create: would an update that is done only with the friends introduce some cliques? Would it mean that some peers will remain undiscovered? These are questions that are left open and can be addressed in further work.

Other open issues are in the analysis that we wanted to perform. We also wanted to see how the network behaves in presence of offline users. We have added this functionality but we have not used it for many analysis. Moreover, we did not analyse how the follow and block relationships evolve and how does this influence the propagation of data. In fact, we have noticed that in the presence of unfollow and block events, the stores on the nodes are more dynamic: they are removed and added back, and the same content arrives to the peer more times. There were many things to consider when performing an analysis on this type of network, that went beyond the scope of this report.

Also, we think that the totally random nature of our simulation does not completely match the natural social behaviours of users in a social network. For this reason, it would have been interesting to understand how close to social behaviours the simulation is, and to study the graphs of connections that were randomly formed during our analysis, to understand if there were cliques or strongly connected graphs, as there may happen in social networks.

REFERENCES

[1] A.-M. Kermarrec, E. Lavoie, and C. Tschudin, "Gossiping with append-only logs in secure-scuttlebutt."

[2] D. Tarr, "ssb-friends," https://github.com/ssbc/ssb-friends, 2017 (accessed December 29, 2020).