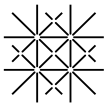

University of Basel

Secure and History-Aware Peer-to-Peer Set Synchronization on Android

Bachelors Project

Faculty of Science of the University of Basel
Department of Mathematics and Computer Science
Computer Networks Research Group
https://cn.dmi.unibas.ch/

Examiner: Prof. Dr. Christian Tschudin
Supervisor: Claudio Marxer, MSc.

David Seger
david.seger@stud.unibas.ch
17-054-693

February 19, 2021

Abstract

Hier könnte ihre Werbung stehen

Table of Contents

1 Introduction

In an infrastructure less environment, such as a peer-to-peer(p2p) network, there is no regulating body that manages the synchronization of content between devices. While this type of network has advantages such as scalability and reliability, it does require different protocols than the more commonly used server based architecture. One of the challenges posed by decentralized networks is the synchronization of content across the participants, especially considering that in a peer-to-peer environment, data can take different paths to arrive at its destination, contrary to the single-distributer concept of server/client networks. As efficiency is key in the growth of new technologies, the need for a fast and secure commmunication protocol between peers arises.

1.1 Append Only Set Synchronization

Many usages of network communication can be broken down to a set synchronization problem, meaning we have to synchronize a dataset across multiple participants, that each might have added things to the set, bringing them all up to the correct state. A subcategory of this problem is the append only set synchronization, this simplifies the problem setting, since in an append only set new elements must always be inserted at the end of a dataset, removing the need to check the whole collection every time a synchronization is to be done. Append only sets can be found in many different network-related uses, for example social media or games.

1.2 History Awareness

Due to the abscence of a central device that regulates all network activity, a decentralized environment needs to self-regulate, effectively transforming every peer into both a server and a client. As every device has to regulate its own connection, the performance of a synchronization can be greatly increased by adding meta-data about past connections with peers, this is called history awareness. While this approach uses more storage space, it has the potential to greatly improve efficiency and therefore user friendliness.

1.3 Security

As there are many risks in any kind of wireless communication, a direct connection between two peers must ensure that the devices are connected to the right partner and that the messages between these partner can not be read or altered by any, potentially malicious, third party.

1.4 The Goal of this Thesis and the General Outline

The aim of this work is to design a protocol allowing participants in a p2p network to efficiently synchronize an append only set, by utilizing history awareness. Furthermore the protocol will be securely implemented in an android app as prove of concept.

This thesis will first take a look at related work that will be the basis for the practical part of the project, mainly the scuttlebutt protocol and the work of Gowthaman Gobalasingam, who built the app that will function as the starting point for the implementation. After this, the protocol and how it came to be will be explained in detail. Next, the implementation of the designed protocol in the android app will be discussed, which will lead into an evaluation of the advantages and disadvantages of the method. Finally, possible future work and usages of the created app will be explored.

2
Related Work

This thesis is largely based on two seperate works, the Scuttlebutt protocol[1] and its functions as the model for an append only set that can be further used in many different ways, such as gaming or building a social media plattform, and the app "SDaTaDirect"[2], built by Gowthaman Gobalasingam as a bachelors project, which will be the base for the practical part of this work. We will now take a closer look at these works and how they played a role in the realization of this project.

2.1 The Scuttlebutt Protocol

2.1.1 What is Scuttlebutt

Scuttlebutt[3] is a decentralized social media plattform, it largely follows the idea of other social media sites, people can post content on their feeds, follow other peoples feeds and interact with them. What differentiates Scuttlebutt is the p2p approach they chose, since there is no central server participants connect to, people need to either have a direct link to peers, in the form of their scuttlebutt ID, or discover other peers in a "Pub", another type of feed where people can subscribe to, making their feed visible for the other subscribers of this pub, allowing people to discover new peers and growing their p2p network. Pubs are always online and their ID, to subscribe to a pub, is distributed through other means, mainly through posting it on the conventional internet.

2.1.2 The Protocol

The scuttlebutt protocol is a flexible communication protocol that can be used for an wide array of applications, one of the first and most common uses is its implementation as a social media plattform as described in 2.1. The following explanations are paraphrased from the Scuttlebutt Protocol Guide[3]:

[1] https://scuttlebutt.nz/docs/protocol/
[2] https://github.com/GowthamanG/SDaTaDirect
[3] https://scuttlebutt.nz/

Every user of a Scuttlebutt based application has an identity, it is generated upon the first registration in a Scuttlebutt network and consists of an Ed25519 key pair, this key pair is non-volatile and identifies an entity in the network. The public key is used as an identifier, the private key is used to sign messages distributed in the network, this is the basis for the security of the protocol. Pubs have the same kind of identity. Every ID is associated with a feed, a set of messages posted by a given entity.

Messages in a given feed form an append only log, they are backwards linked with each other, allowing an easy access to a stream of messages. Messages are signed by the publisher using its own private key. With this, the protocol ensures that no third party can alter the message and claim it is published by the original author. Communication between peers happens over a Remote Procedure Call Protocol (RPC), peers send packages containing an procedure identifier to a remote partner, who then executes the specified procedure, possibly using arguments also contained in the packages, locally. Synchronization of feeds happens after connecting to a peer, a device sends a RPC to its peer, in which it requests all messages newer than the last message it has received in a given feed. The peer answers this request with a stream of messages, beginning at the newest and moving down the message chain until it has forwarded all messages.

Figure 2.1: A RPC request for new messages in a feed[3]

the most common implementation of Scuttlebutt, patchwork[4], follows the policy of requesting and saving messages of all feeds that are up to three hops away, with the first hop being all feeds that the client has actually subscribed to. It shows the user all messages in a two hop distance, and caches the messages of the feeds that are three hops away, as can be seen in 2.2

[4] https://github.com/ssbc/patchwork

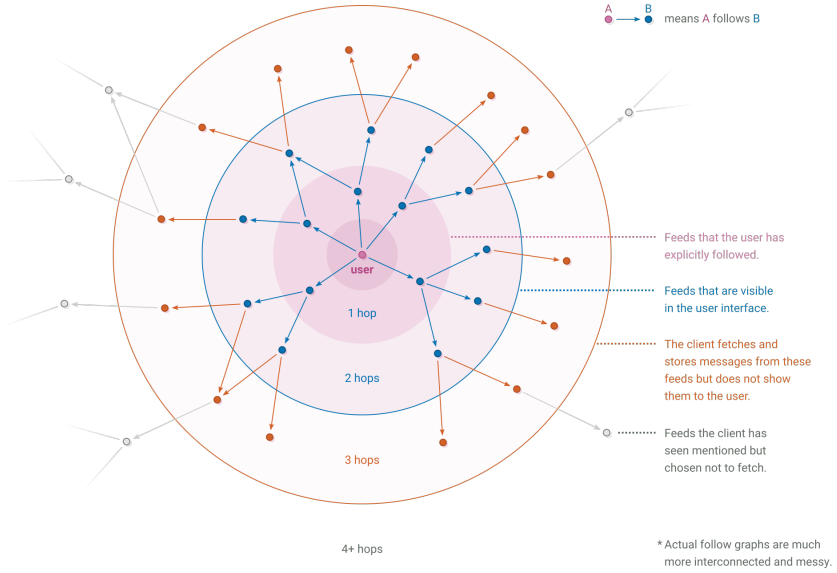

Figure 2.2: Graph showcasing the follow relations and the messages being cashed in Patchwork[3]

This, paired with the fact that Scuttlebutt is not history aware, leads to a large amount of requests, answers and data being tranferred between peers when connecting to each other. While we will adapt the basic concepts of feeds, pubs and messages to our app, we will add metadata concerning past connections and reduce the reach of the protocol implementation to 1 hop, to increase the efficiency of the implementation.

2.2 SDaTaDirect

SDataDirect will be the security basis for implementation. The App was developed by Gowthaman Gobalasingam, it implements a protocol created by him to ensure secure data transfer on the basis of WifiDirect[2].

2.2.1 The Protocol

The app follows a three phase procedure: The users first each scan a QR-Code of the exchange partner, these QR codes include the devices wifi mac address, a symmetrical AES key, and the public key of a generated RSA key pair. These keys are the cryptographic basis of the protocol, they are newly generated for all peers a device wishes to connect to, and saved in an internal database. At the end of phase 1, both parties agree on one of the generated symmetrical AES keys for further communication. In phase 2, the two devices connect via Bluetooth, over which they exchange the signature of the AES key pair, to verify that the devises are actually connected to the right peer. In phase 3 the wifi direct connection gets started, the devices connect to each other, and the client can send the host a file, which is encrypted using the symmetrical key and signed using the private key. The file is only encrypted once the signature has been verified using the previously exchanged

public key. Using the combination of AES and RSA keys, as well as the verification pf the peer in phase 2, ensures that it is impossible for any kind of man in the middle attack, be it reading the exchanged file or manipulating it in any way.

2.2.2 The App

The app was implemented using Kotlin[5], a programming language by JetBrains, it is fully interoperable with Java and has taken center stage in android development, as stated by googles "kotlin-first" commitment[1].

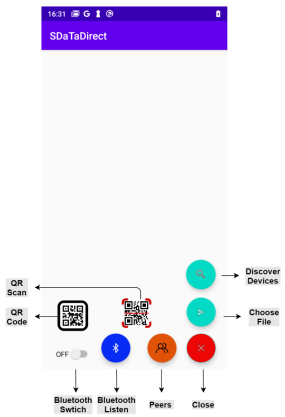

Figure 2.3: The UI of the SDaTaDirect application[2]

The app provides a simple interface to connect with a peer and exchange a file, the exchange is one-way only, and after a file has been exchanged, the connection will be closed. The app will provide the cryptographic basis for the implementation, we will keep the three phase security protocol, but modify the implementation to provide two-way communication, non-aborting connections and the feed/pub functionality as explained in 2.1.2. With a basis for the realization of the theoratical work, we can now go on to have a look at the design of the protocol.

[5] https://kotlinlang.org/

3
Designing the Protocol

To design a history aware set synchronization protocol in the vein of the Scuttlebutt p2p protocol, one of the first things to consider is the data the protocol can work with. Because of the nature of history awareness, we need to define different data objects that can be used to design the protocol.

3.1 Needed Data

To reduce the amount of traffic in the network we will have to save as much information about the different connections as possible, in this step we will try to identify the data needed to keep non-content (read: control messages) packages at a minimum:

1. To only allow communication about feeds that the peers are actually interested in, we have to save information about which peer is following which feeds or pubs.

2. We want to ensure that the saved information, which will dictate the feeds that will be relevant in the protocol, is always up to date. the first history aware element we will have to include is a measure to know if changes have occured in the feed subscriptions since the last synchronization.

3. To filter out any uninteresting feeds, the peers have to know which feeds dont have any new messages in them, for this we need to have a way to save if any new messages arrived in a relevant feed since the last synchronization

To address requirement one, every application implementing this protocol needs to maintain a database where every peer is saved and which feed he is subscribed to. The second requirement can be easily implemented using timestamps, these timestamps have to be attached to every peer in our database, signaling the last synchronization, as well as to every feed, where we update it every time we discover a new one, when we subscribe or when we unsubscribe from a feed. The last thing we need requirement three, this can be achieved by assigning message sequence numbers to every message published in a feed, and by saving, after every synchronization, what the last message sequence number of a feed was that we have transmitted to the partner.

With these data requirements identified we can move on to designing a protocol, we constructed it with 4 general phases in mind, which will be explained further in the following sections

3.2 Design Goals of the Protocol

There were some goals we kept in mind while developing the protocol, that are worthy mentioning here.

During the creation we decided on a "pull instead of push" approach, this can be seen in phase 3, no messages get exchanged if the partner doesn't ask for them first, this was planned this way to keep it more flexible for different kind of usages, for example if multiple devices are currently connected to device X, device X can actively chose from which of its peers it wants to receive the updates of a specific feed from, one obvious reason being if one of the devices has more messages in the feed than the others.

the phases of the protocol are largely kept independent from each other, while a complete set synchronization is performed when two devices go through all of the steps, it is possible to trigger the phases independently, depending on what the application that implements it, needs. To give an example, if a chess application running over a p2p network uses this protocol, it would not need phase 1 and 4 of the protcol. We designed it in such a way that it supports the same structures as can be found in Scuttlebutt, but it is a protocol that can be used for any kind of append only set synchronization.

3.3 History Aware Set Synchronization Protocol

3.3.1 Phase One: Updating the Peer

The first phase is used to update the peers database, since we are working in a history aware environment, we dont need to exchange a whole list of all feeds that we are subscribed to, every time we connect to each other. There are two possible messages that can be sent in this phase, "subscribed" and "unsubscribed", the two devices simply check in their database when the last synchronization between them was, and send a short update to the peer for every feed in which a change happened after the last synchronization. In case a new feed was discovered by one of the peers, and the partner is not yet aware of this feed, the update message will trigger a probe, asking for details on the feed. In 3.1 we can see a hypothetical interaction between two devices in phase 1.
The protocol coordinates itself using flags to signal the partner that they are done sending their data and that they are ready to receive the data of the partner, the same flag is used to move on to the next phase.

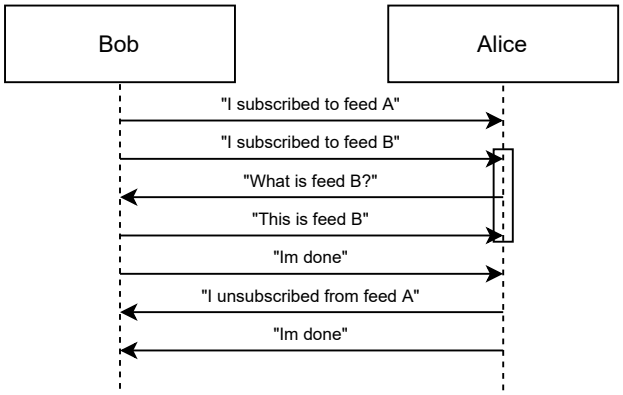

Figure 3.1: An example of how phase one could look like for Bob and Alice

3.3.2 Phase Two: Notify the Peer of the Available Information

In phase one we ensured that the further communication of the protocol is only limited to feeds that are relevant to this synchronization. The filtering goes a step further in phase two, the two devices go through the list of relevant feeds for the partner, and compare the sequence number of the newest available message in the feed with the sequence number of the last message that they have sent this particular peer in this particular feed, as we have defined in requirement three in 3.1. If the sequence number in our local version of the feed is greater than the sequence number of the last transmitted message, we send a package containing the sequence number of the newest available message. If there are no new messages in the local feed, it will be discarded and not further considered in phase 3. In 3.2 we can see a hypothetical interaction between two devices in phase 2.

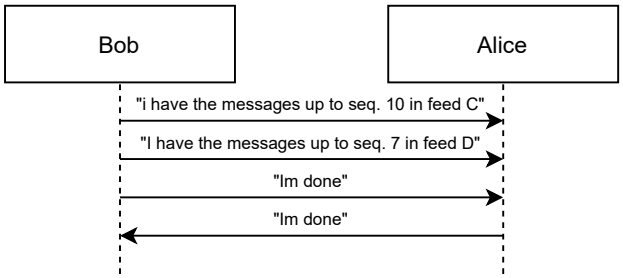

Figure 3.2: An example of how phase two could look like for Bob and Alice

Again, the phase is coordinated with flags that give the control to the other device or trigger the start of the next phase.

3.3.3 Phase three: Requesting the Messages Missing in our Local Feeds

The devices save a list of all the feeds and sequence numbers they have received in phase two, now, in phase three, they internally compare the newest available sequence number of the peer with the newest message the device has in its local version of the feed. If the remote device has content that this device has yet to receive, it sends out a request to its partner, containing the sequence number of the newest message that the device has saved in the feed, the remote device will answer this request by sending all messages with a newer sequence

number than the one that it received to the device. When both devices went through all of the feeds they have saved from phase 2, they will both have synchronized all of the feeds theyre interested in. With filtering out feeds not eligable for synchronization in phase one and two, we hope to reduce the network communication compared to the simpler approach of Scuttlebutt, in which the devices ask for new messages in every feed. A simple example about how phase 3 could look for two devices can be seen in 3.3.

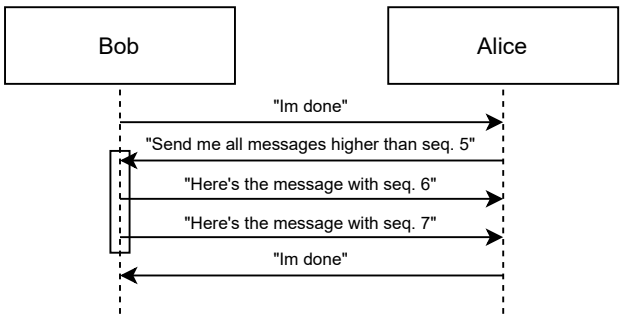

Figure 3.3: An example of how phase three could look like for Bob and Alice

The communication control flags are once again included to guide the protocol along. The end of phase three automatically triggers phase 4.

3.3.4 Phase Four: Synchronizing the Timestamps

In the last phase of the protocol we simply want to make sure that no discrepancy between the last synchronization timestamp occurs between the two devices, so the initiator of the synchronization sends out the timestamp to be saved as the last synchronization time. This is a simple one package operation, as can be seen in 3.4.

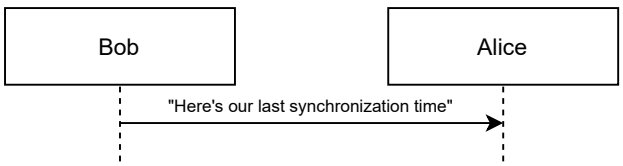

Figure 3.4: An example of how phase four could look like for Bob and Alice

After the timestamp synchronization, the protocol is done.

3.3.5 The full protocol

If we put all of the phases together, we arrive at the final product of our design efforts, in 3.5 we can see the full protocol that will be the basis of the practical portion of this work.

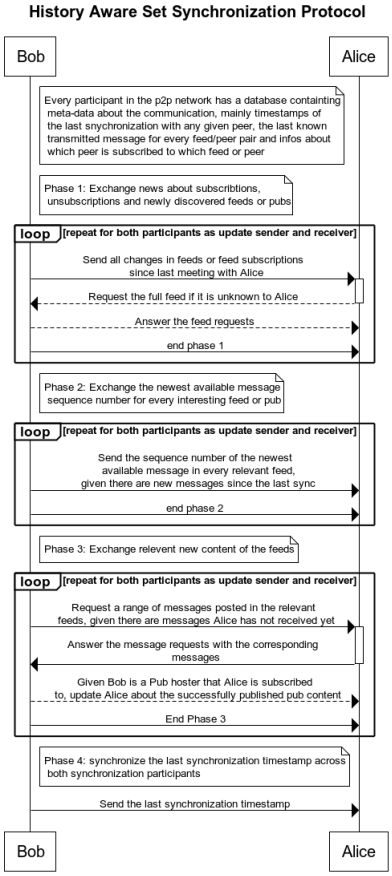

Figure 3.5: The final design of the set synchronization protocol

4
Implementation

The practical part of this work aims to implement the previously constructed history aware p2p set synchronization protocol in a secure way, it will be a standalone app based on SDaTaDirect. To showcase the synchronization it will provide a similar functionality as discussed in 2.1, it will provide the same system of feeds and pubs and the possibility to dynamically add messages to feeds. The goal will be to allow a fast synchronization of these feeds and messages between two devices.

4.1 Basic Functionality

Every device has an associated feed, it gets automatically created and stored in the datbase when the app is started for the first time. In this feed the device can publish messages directly through the app. A user can also create one or more pubs, feeds that simply republish all messages in the private feeds of all devices subscribed to the pub, to subscribe to a pub, a device has to directly connect itself to the pub owner when it wants to subscribe to a pub, after that the pub messages will travel through the p2p network just like any other message in a private feed. A user can not only subscribe to a pub, but also follow private feeds of devices. The app is designed as a proof of concept for the set synchronization protocol, so the functionality of the sets themselves is kept at a minimum. To synchronize one device with another device, and thus forward all messages it has published into the network, the user simply has to securely connect to the partner through the SDataDirect protocol, as described in 2.2, upon a connection the two devices automatically start the synchronization, without any further user input. The synchronization is done securely by encrypting all packages with the established, peer dependent, AES key pair and signing it with the public key of the sender, as proposed by Gowthaman Gobalasingam in the SDaTaDiret protocol.

4.2 Configuring the Database

The SDaTaDirect app already has a database set up, but because its only use was to save the previously connected peers, the data model remained quite simple, with only one table

that includes all needed data. The data is organized in a Room[6] database. To support the history awareness of the synchronization in a peer-to-peer environment, every device needs to save the state of the last synchronization with any peer locally, as discussed in 3.1. We created tables to support the feeds and message functionalites, extended the peer table to include the history aware elements and added a way to save which peer is interested in which feeds. in 4.1 we can see the design of the database in the form of an ER diagram. One important difference to the original SDaTaApp is the persistent public key, the app checks on start-time if it already posseses a public key, identifying it in the network, if not it will create a key and store it in the database. In the original design, a new public key was created for every new peer.

Figure 4.1: the final ER diagram of the underlying database of the app

4.3 Extending the GUI

While no changes had to be made to the way devices connect to each other in the app, and therefore no additional interfaces had to be created to initialize the synchronization, the GUI needed to be changed to allow for dynamic content creation, interaction and viewing of the feeds the app has discovered.

The user can visit its own feed after starting the app on the feed view, as can be seen in 4.2.

[6] https://developer.android.com/jetpack/androidx/releases/room

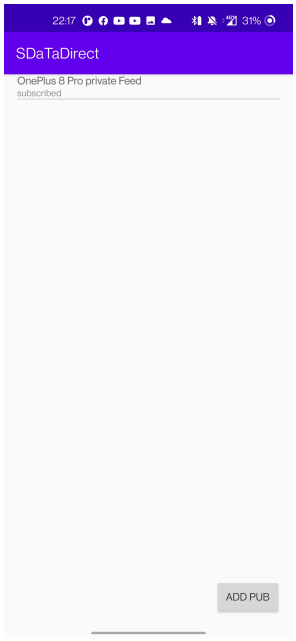

Figure 4.2: The view of the feeds after the first start, with no discovered feeds or created pubs

When the user clicks on a feed, the app opens up a view of all the messages in the feed that this device has received, if the user is allowed to publish messages in the feed (if the feed belongs to this device), a small text box allows the user to create messages that will be posted in the feed. A user can create as many pubs as they want, in which any messages, published in the private feed of the device owner, automatically will be reproduced. In 4.3 we can see the dialog to create a new peer, and in 4.4 you can see the reproduced messages that were originally published in the creators private feed, which can be seen in 4.5

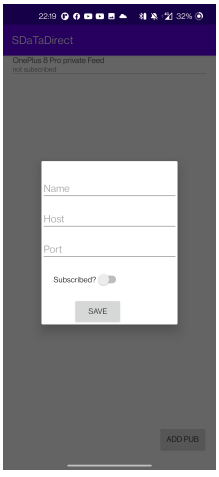

Figure 4.3: Creating a new pub inside the app

Figure 4.4: the view inside of the newly created pub, all private messages of every subscribed device will be published here

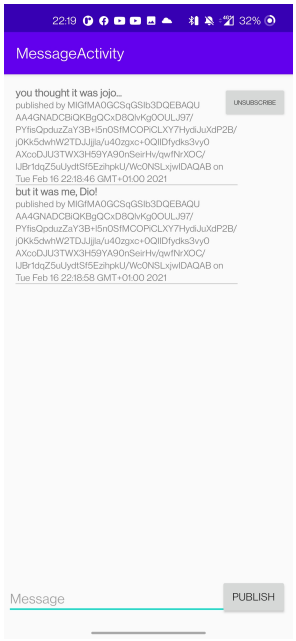

Figure 4.5: The interface of a feed in which the user is allowed to publish messages

4.4 Adjusting the Communication Functionalities

The app was set up for a one-time-only data exchange, it opens a TCP socket where the client can send data to the host, after the data has successfully been transmitted, the socket closes again and the app has to be restarded to send another file. As the protocol requires two-way communication, we had to adapt the communication, the host creates a new socket for every incoming connection, with which it connects to the new client, allowing messages to go both ways. The sockets get organized in a static connection manager, from which every connected device can be reached from. A message buffer catches all messages from all sockets and conitnuosly verifies them by their signature and forwards it to the interpreter. All packages contain a payload and a signature, this all gets converted to byte arrays that are transmitted using a kotlin data output stream.

4.5 Adapting the Protocol as a Kotlin Script

As previously mentioned, the protocol was implemented as a RFC protocol, There are predefined commands that are transmitted that cause an action to be made in the partners device. Together with any additional arguments that need to be passed to the function, the method call is packed into a JSON object and transformed into a byte array, which then will be forwarded to the communication manager for encryption and signing. An overview of all available method calls can be seen in 4.1

The protocol is mainly implemented in the *SetSynchronization.kt* file, the script is started after every new connection, and can be called programatically at any time by providing the target, the role of the invoking device (master or slave), and the starting point of the protocol (e.g. phase one to four). The file contains all methods that can be called remotely, it also contains a static Hashmap that collects all the feeds where the partner has new

Method Call String	Arguments	Description
SEND_FEED_UPDATE	feedKey: String, Subscribed: Boolean	Inform the partner of a subscribtion change or newly discovered feed since the last synchronization
INQUIRE_FEED_DETAILS	feedKey: String	Inquire the details of a feed in case the device hasn't discovered it yet
ANSWER_FEED_QUERY	feedKey: String, type: String, host: String, port: String, subscribed: Boolean, owner: String	Send all required details to save a new feed, the owner String is a public key of another device, the type can be either Pub or Private
END_PHASE_ONE	*none*	End the first phase of the protocol, automatically triggers phase two
SEND_LAST_SEQ_NR	feedKey: String, lastSeq: Long	Inform the partner, if a new message has been published in the feed since the last synchronization, what the sequence number of the newest available message is
END_PHASE_TWO	*none*	End the second phase of the protocol, automatically triggers phase three
SEND_RANGE_MESSAGE_REQUEST	feedKey: String, lowerLimit: Long	Request all messages in a feed that are newer than the lower limit sequence number provided
SEND_MESSAGE	sequenceNr: Long, feedKey: String, content: String, publisher: String, signature: String, timestamp: Long	Provide a message in a feed that the partner has not received yet, the content variable is a byte array encoded to string using the UTF_8 Charset, publisher is a public key of a device, the signature can be used to verify that the message has not been altered
SEND_PUB_UPDATE	sequenceNr: Long, feedKey: String, content: String, publisher: String, signature: String, timestamp: Long	If one of the messages received during the sync is a message that has to be published in a pub hosted by the receiver, and if the current synchronization partner is subscribed to this pub, inform him of the new message that has been published in the pub
END_PHASE_THREE	*none*	End the third phase of the protocol, automatically triggers the synchronization of the lastSync variable
SEND_LAST_SYNC	lastSync: Long	Synchronize the lastSync variable across both devices, this method will only be used by the initiator of the synchronization

Table 4.1: An overview of the available methods that can be used in this implementation of the protocol

messages, this collection gets cycled through in phase three, and for every feed where the device is not up to date, a message range request is sent. The methods largely update or exchange data objects from the database according to the protocol, one thing that had to be changed from the general set synchronization protocol to the specific use of a Scuttlebutt like environment is the special handling of Pubs, since we still have to send updates about feeds that the partner is potentially not interested in, if the feed owner is subscribed to a pub hosted by the partner.

Bibliography

[1] Android Developers. Android's kotlin-first approach. URL https://developer.android.com/kotlin/first.

[2] Gowthaman Gobalasingam. An infrastructure-less and secure localpeer-to-peer data exchange protocol. Master's thesis, University of Basel, 2020.

[3] Scuttlebutt. Scuttlebutt protocol guide. URL https://ssbc.github.io/scuttlebutt-protocol-guide/.

A

Appendix

Declaration on Scientific Integrity
Erklärung zur wissenschaftlichen Redlichkeit

includes Declaration on Plagiarism and Fraud
beinhaltet Erklärung zu Plagiat und Betrug

Author — Autor
David Seger

Matriculation number — Matrikelnummer
17-054-693

Title of work — Titel der Arbeit
Secure and History-Aware Peer-to-Peer Set Synchronization on Android

Type of work — Typ der Arbeit
Bachelors Project

Declaration — Erklärung
I hereby declare that this submission is my own work and that I have fully acknowledged the assistance received in completing this work and that it contains no material that has not been formally acknowledged. I have mentioned all source materials used and have cited these in accordance with recognised scientific rules.

Hiermit erkläre ich, dass mir bei der Abfassung dieser Arbeit nur die darin angegebene Hilfe zuteil wurde und dass ich sie nur mit den in der Arbeit angegebenen Hilfsmitteln verfasst habe. Ich habe sämtliche verwendeten Quellen erwähnt und gemäss anerkannten wissenschaftlichen Regeln zitiert.

Basel, February 19, 2021

Signature — Unterschrift