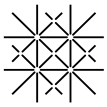


University of Basel

A Feed Bundle Protocol for Scuttlebutt

Bachelor Thesis

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
Computer Networks
http://cn.dmi.unibas.ch/

Examiner: Prof. Dr. Christian Tschudin
Supervisor: Prof. Dr. Christian Tschudin

Jannik Jaberg
jannik.jaberg@unibas.ch
2017-054-370

02.07.2020

Acknowledgments

I would like to thank Prof. Dr. Christian Tschudin for giving me the opportunity to work with him on this thesis. He has supported me during the entire process of planning and developing this thesis and has given me valuable and constructive suggestions and guidance, especially in light of this unique COVID-crisis time. In addition, I would like to thank Christopher Scherb and Claudio Marxer for supporting my work with essential feedback. Finally, I want to express my gratitude to my whole family and friends for supporting me in so many ways during the creation of this thesis.

Abstract

Aspiring new technologies emerge every day, one of which is Secure Scuttlebutt (SSB). Secure Scuttlebutt is a peer-to-peer communication protocol based on ID-centric append-only logs (Tarr et al. [4]).[1] The aim of this thesis is to take the mechanics from Secure Scuttlebutt and bring them to a more commercial environment by introducing new intermediary service providers (ISP) which offer connectivity to servers. Having a contract with such an ISP makes the initial onboarding much easier than in SBB.
By splitting up the ID-centric feeds into feed pairs for every connection, information on the specific dialogs gets bundled and stored independently. Since this is the smallest abstraction, it allows an additional form of bundling by multiplexing log entries together into larger feeds. Therefore the challenge of the immense replication work done by SSB is approached differently.

[1] Quelle

Table of Contents

1 Introduction

The world is constantly changing and so is the internet. At this very moment, a revolution in networking research is taking shape. This movement is leading away from well-known, proven practices and measurements of the centralized web and strives for novelty: distribution. The direction is away from centralised servers and classical routing and moving towards routing into a new peer-to-peer-driven, distributed and decentralized web. Secure Scuttlebutt is exactly one of these new developments, which captivate with refreshingly different approaches to solving common networking problems. Yet they are still in development and have a future that is anything but sure.

1.1 Secure Scuttlebutt

Secure Scuttlebutt (SSB), invented and created by Dominic Tarr in 2014[2], is a gossip peer-to-peer communication protocol (Tarr et al. [4]). The term scuttlebutt is slang for "water-cooler" gossip used by sailors and boatsmen. Coincidentally his motivation to develop such a protocol was an unreliable internet connection on his sailboat and the result was his own offline-friendly secure gossip protocol for social networking. [3]

Differing from other technologies, Secure Scuttlebutt does not offer a self-explanatory out of the box onboarding principle. In other software, the user typically receives suggestions for content (e.g. Instagram) or connectivity and management are built into the software (e.g. default gateway DHCP). In SSB, the user has to connect manually to a pub via an invite code, which they must obtain on a channel other than SSB.[4]

1.2 Motivation

However, it is problematic for new users to connect to the SSB world, hence a very interesting and intriguing problem has presented itself. SSB is a promising, innovative, new technologie

[2] Initial commit github
[3] P2P-Event Basel
[4] Invide Code - https://ssbc.github.io/scuttlebutt-protocol-guide/

that has a great deal of potential. At the moment, it is still in an experimental state and used primarily in pilot projects where the technology is connected to existing domains (social network, git, databases etc.)[5] This thesis would like to explore its potential in a more commercial manner and environment.

1.3 Goal

This thesis explores the role of intermediary "connectivity providers" which sell connectivity e.g. to Google or Facebook, through a prototype implementation of a Feed Bundling Protocol. It is based on SSB, but also differs in several concepts. Introducing these intermediary participants, where you are connected on start up, will make the onboarding easier, since they will hold all the information to create new connections. In plain English: *It's a guy who knows another guy who can help.* With "feed-pairs", which are described later in this thesis, the ID-centric information gathered into one single feed from SSB is split into parts. This results in less data in each dialog between two participants and allows bundling.

1.4 Outline

First, a more detailed description of SSB, with a focus on the concepts and problems connected to the Feed Bundle Protocol, will be given. Then, the newly created and adapted concepts, as well as the architectural idea of the FBP with respect to SSB, will be presented. Subsequently We will take a closer look at the implemented code and how it has been solved. This evaluation covers previously solved issues, as well as newly generated problems with the approach and how they might possibly be solved. Finally, the thesis will present a conclusion and highlight future challenges discovered during the process.

[5] Quelle

2
Related Work

In this chapter we examine some of the key features which were taken into account when creating the Feed Bundle Protocol. Before taking a closer at the baseline for the protocol, which consists of parts from the Secure Scuttlebutt technology [3] and the Remote Procedure Call Protocol, we will quickly jump into the blockchain and its properties, everybody who reads this thesis will have heard about it by now.

2.1 Blockchain

The blockchain is well known as the foundation of the bitcoin. It has received an extensive attention in recent years (Zheng et al. [5]). But what is it that makes the blockchain so impressive and desired? The blockchain is described by Zheng et al. [5] as an immutable ledger which allows transactions to take place in a decentralised manner. Exactly these key properties we also find in Secure Scuttlebutt.

2.2 Secure Scuttlebutt

Having a rather well known append-only log like the blockchain as a foundation and rather well know by the broad mass makes the jump into the the universe of Secure Scuttlebutt much easier. Secure Scuttlebutt is a novel peer-to-peer event-sharing protocol and architecture for social apps.[6] The aim of this section is to give a very high level overview about SSB, its ideas and properties, since the they are not quite easy to understand.

2.2.1 Append-Only Log

2.2.2 Onboarding

Onboarding in Secure Scuttlebutt differs from the

[6] Tschudin Paper

2.3 Remote Procedure Call

Remote procedure calls, as the name implies, are based on procedure calls but extended to provide for transfer of control and data across a communication network. There are two participants in the simplest manner, the caller and the callee. The caller wants to invoke a procedure with given parameters. The callee is the instance, which actually proceeds with the data and returns the result of that specific request. If an RPC is invoked, the the caller's environment is supended and all the information needed for the call transmitted is through the network and received by the callee, where the actual procedure is executed with these exact parameters. The benefit of such an RPC-protocol is that the interfaces are designed in a way that third parties only write the procedures and call exactly these procedures in the callers environment (Birrell and Nelson [1]) This leads to a very promising basic version of such an RPC-protocol for this thesis, since it allows the caller to invoke the procedure in the caller's own environment, but the procedure is actually performend by a callee, which returns the result back to the caller.Birrell and Nelson [1]

3

Concepts and Architecture

As described in the chapter Related Work, SSB is an ID-centric single feed driven environment, where onboarding is challenging. This prerequisite changes from the beginning. The basic idea of the Feed Bundle Protocol is to split up this ID-centric environment into replicated feed-pairs, where two participants hold at least one of such a pair. This pair contains the whole dialog between two identities which have a contract with each other. Similar to the tin can phone from your childhood, where you had two cans connected for every friend you want to communicate with. By introducing intermediary service providers, the onboarding happens at contract signing. Clients have a possibility to connect to new servers via this ISP and create a new feed-pair for each server, which is replicated over the same ISP. Since this approach means an enormous amount of feed replications between ISP and server, these feeds are bundled again.

3.1 Tin Can Analogy

This system as described may seem difficult to understand, but the description can be simplified. Think of it as a tin can phone from your childhood, but where you have two cans on each side. You need this two-can set up for each friend with whom you want to communicate. You start with a connection to your best friend, the one you trust the most. You talk into the “talk can” and that can ’saves’ everything you say to it. From your other can, the “listen can”, you are only able to hear things from your friend, but this can also saves everything your friend says. By labeling the cans with our names in the format ”talker-listener”, we distinguish them. The You-Friend can is therefore you ”talk can” and your friends ”listen can”, the Friend-You can the other way around. This is one replicated feed-pair, having both cans on both sides.

Having this, the dialog needs a way or language to express expectations or requests from one side to communicate with each other, where you can declare what you want from your friend, which leads us to the simplified RPC protocol. After a while, it gets boring only talking to this one friend. Luckily, your friend is the coolest kid in school and knows everyone and even tells you about everyone he knows. Then you ask your friend if he could introduce you to his other friends, since you are tired of only talking to your friend. This introduction

process is real, human social behavior, not face to face, but via tin can phone.
After your friend has introduced you to one of his other friends and this other friend decides to be friends with you, you and your new friend start to build a new tin can phone, with the “talk can” and “listen can” tins. Due to the fact that you are too far away from each other, you cannot just have a cord from one to the other, so your best friend allows you to route the cord through his house. This corresponds to the replication of the feeds over an intermediary connectivity provider.
Having this, the dialog needs a way or language to express expectations or requests from either side to communicate with each other, where you can declare what you want from your friend. Leading to the simplified RPC protocol.
There is another problem, however: you are not the only one. After a while, your best friend has so many connections running through his house from all his friends who want to talk to their other friends, that here is an enormous number of strings going to that other friend. Your friend decides to combine all these strings into one and send all messages through this one, single bundled connection, which contains all the information about which tin can it needs to come out of at the end. He multiplexes. Given that little story, we can derive concepts and architecture for the tin can phones of the future.

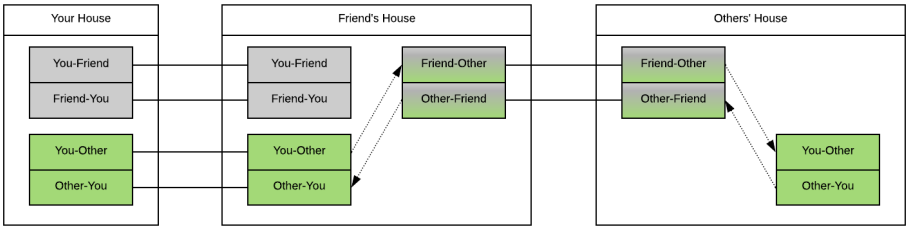


Figure 3.1: Tin can analogy illustrated.

3.2 Contracts

Now we can see the foundation of the friendship. The friendship between you and your best friend and the friendship between your best friend and his other friend. By the same token, it is the business contracts between the nodes that provide the foundation for the entire connectivity, protocol and bundling. These contracts are the most important building blocks of the whole thesis since they define the behaviour of the replicated feed-pairs, onboarding mechanics and bundling.

3.2.1 Contract Values

To build the tin can phone, three basic identifiers are needed. First of all, you have to trust each other. This corresponds to the concept of a legal contract between the two parties. Next you need to know the names to label the phones, so you know who you are talking to. These are the public keys. Since everybody in your house can use the tin phone, you also need some sort of code so that your friend knows that it is you who is sending the message

and not your mother. These are the private keys. Having that, you need your two tin cans, the feed-pair and two wires, the replication process.
If you do not know your friend's address, you do not know where to put that wire, so you need some sort of address as well. The address, as the name is well chosen, refers to the IP-address, since this is the most commonly known address of a computer. Last but not least, to distinguish all the cans, you label them corresponding to the "talker-listener" format. Therefore, a contract consists of a public key, a private key, a feed-ID and the peers public key, feed-ID and location. Since a contract has been established, we need to know what happens in the tin cans and the wire. This leads to the replicated feeds.
Eventually Picture of identities with contract

3.3 Replicated Feeds

The following statements can be extracted from the Scuttlebutt Protocol Guide: "A feed is a signed append-only sequence of messages. Each identity has exactly one feed."[7] and "Messages from feeds 3 hops out are replicated to help keep them available for others."[8] - So what do these properties mean?

What can be derived from this information? Signing ensures that you can trust that this message was created by one specific identity or more specifically, the encryption of plain text with the sender's private key to a cipher text. The crypto text can be deciphered with the sender's public key.[9] Therefore, this guarantees integrity. Append-only means this sequence can not be forged. So there is no possible way to modify or delete any entries that were appended at any time.[10] This append-only property is realized with a mechanism which references the previously generated message.[11] The ID-centric architecture described means exactly one identity (key) is mapped to exactly one feed, where every single bit of information an identity creates and "consumes"[12] in the SSB universe is stored in this feed. There is a lot more going on in the SSB feed and protocol, but an adapted simplified version with three fields*(Figure link and simplify even more)* is more than sufficient for this thesis. These properties underline the trust by guaranteeing completeness and validity of the information read in a feed. We can, however, see the sticking point in the replication of ID-centric feeds. Since the replication of the SSB protocol always replicates the whole feed with all information to all peers three hops away from a single identity, there is great amount of replication work done. This causes latency and long scuttling time (feed update).[13] By splitting the feeds into smaller ones and having the effective communication between two parties bundled in the feed pairs mentioned previously, we try to bypass this bottleneck. As a result, we have a diagram like this: For the sake of clarity, only the situation between the

[7] https://scuttlebot.io/more/protocols/secure-scuttlebutt.html
[8] https://scuttlebot.io/more/protocols/secure-scuttlebutt.html
[9] quelle
[10] Feeds - https://ssbc.github.io/scuttlebutt-protocol-guide/
[11] Feeds - https://ssbc.github.io/scuttlebutt-protocol-guide/
[12] describe better
[13] Quelle

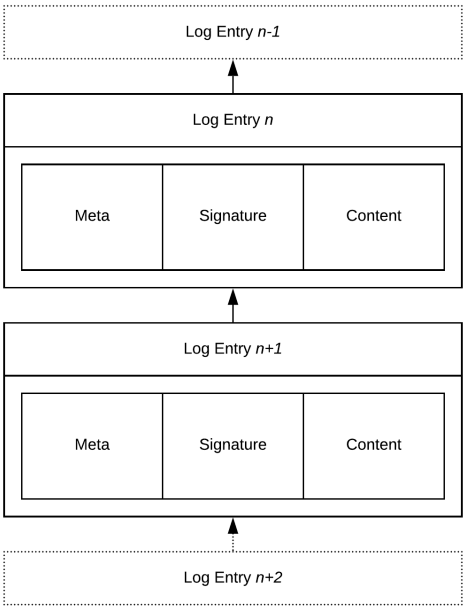


Figure 3.2: A schematic simplified feed with the fields meta, signature and content.

client and the ISP is shown.

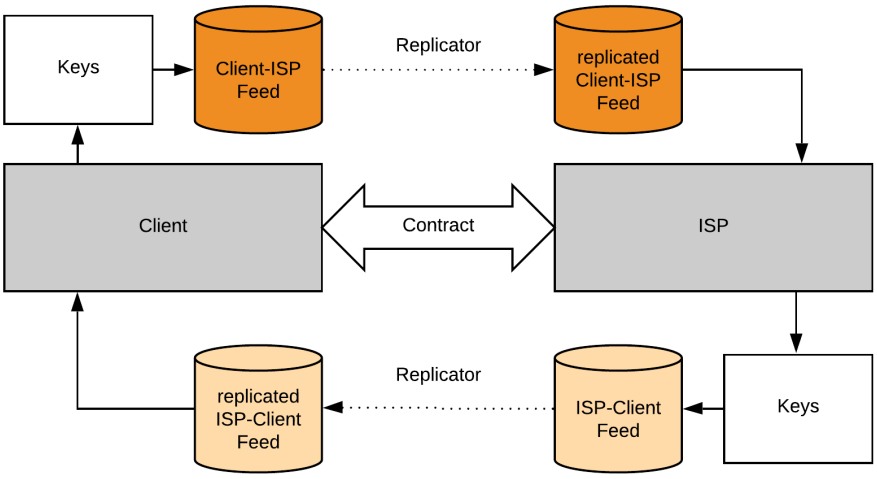


Figure 3.3: Feed-pair between client and ISP.

The replicator or the replication process has it first appearance. As a concept, there is some sort of replicator instance or procedure that replicates the feeds to the corresponding address or location. Let's take a closer look at the implementation part.
Having this setup, the next step is to have a possibility to communicate, so the client can request information from the ISP's real database.

3.4 Remote Procedure Call

As explained in the section Related Work, Remote Procedure Calls are a useful paradigm for providing communication across a network between programs (Birrell and Nelson [1]). This faces many challenges, but these are not particularly relevant to this stage in the development of the Feed Bundle Protocol. Therefore, the RPC used in this section is a very simplified version.

The idea is to have a caller, in our case the client, and a callee, the ISP or server. Having this kind of request–response protocol, an RPC-request is initiated by the caller, which sends a request to a callee in order to execute a specified procedure with given attributes. However, "sending" is not the appropriate terminology for an environment with the replicated feeds. The RPC-request is written in the Caller-Callee feed, which is replicated to the callee. When the replication is complete, the callee is notfied of the feed change and can read the request. After processing the request, the result is written in the Callee-Caller feed and the caller can use the result as needed. In our case, these specified procedures are called services. By having only one such service e.g. the echo-service, which simply echoes the attributes back to the caller, it is ensured the RPC-protocol works as defined. The introducing and detrucing mechanics, described in the next section, can also be summarized in such services, where the caller makes, for example, an RPC-request to the introduce-service with the necessary parameters.

3.5 Introducing and Detrucing

3.5.1 Introducing

Recapping the tin can phone story: The idea of introducing is to get in touch with a new friend, to whom your best friend introduces you. You and your new friend create a new tin can phone. Since the cord is only long enough to reach your best friend, he connects you to your newly acquired friend. Therefore, the general idea of introducing in the context of the feed-bundling protocol, is onboarding to a new server over your ISP. This approach differs from the common publish and subscribe (pub-sub) architecture. Where the server has no choice to decline a client in the pub-sub model, this is the foundation of the introduce-detruce model, by simulating real, human behaviour. A more detailed description: A client writes an RPC-request for the introduce service of his ISP in the Client-ISP feed. This request needs an attribute which specifies the server which the client wants to be introduced to. The ISP invokes the introduce service with this given parameter, which now makes an RPC-request with information about the client and the fact that it wants to introduce itself to the server. By writing in the ISP-Server feed, the server detects the change and has the choice to either accept or decline the introduce inquiry. If the Server accepts the introduction, it will directly create the Server-Client feed which is replicated to the ISP. Afterwards, it sends a confirmation or acceptance back to the ISP to fulfill the request.

The server can also decline the introducing attempt, which results in a rejection, followed by no contract or some sort of empty contract. Either way, the ISP gets the result and fulfills the initial RPC request. The client now gets their result. Depending on the state of acceptance or rejection, the Client-Server feed, replicated to the ISP in accordance with the

contract,is successfully established. If the client wants to use a service from server, it only writes the request in the corresponding feed and the procedure is the same as described in the RPC Section, whereby the feed is just simply forwarded over the ISP. An important distinction: only the client can introduce itself. The server has no knowledge of clients and also no way to acquire knowledge of clients, so only the client can ask the server for a contract.

3.5.2 Detrucing

Detrucing as a newly invented word in this thesis, since normally after you introduce yourself to a person, there is no way to make this "un-happen". It acts the same as an unfollow in a pub-sub domain or as terminating the contract. But contrary to the introduce procedure, both parties of the contract can detruce. Either the client or the server can send an RPC-request to the ISP service to detruce, which is propagated to the opposite end descibed above in the Introducing section and results in the termination of the whole contract. The result of this action is deleting keys and feeds. There is no way to decline a detruce service request. An important note: after detrucing from either side, the client can yet again introduce itself to the server and the server has the possibility to accept or decline the introduction again. A new diagram of the network can be derived using these descriptions.

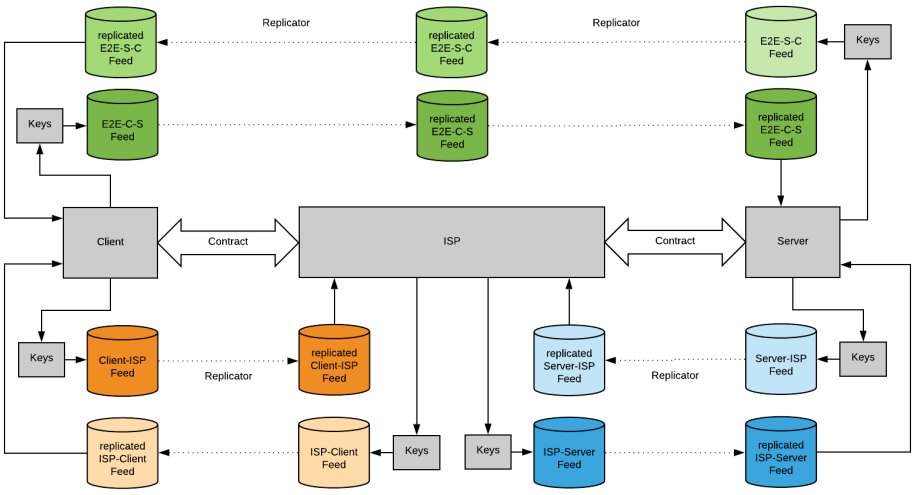


Figure 3.4: State after accepted introduce-request from Client to Server. Clearly seen the green End-to-End feed-pair is replicated over the mediating ISP.

3.6 Bundling

Taking a look at the real-world problem again, the ISP will have a random number of clients, many of whom want to communicate with the same server. Instead of repeating each End-to-End feed-pair over the ISP to the server, the new requests will be sent through a single feed pair between the ISP and the server. This should reduce the amount of replication work enormously.

3.6.1 Adapted Introducing and Detrucing

The introducing and detrucing idea remains the same, whereby the replication process is different. After a server accepts a client, the server generates the entire feed pair: Client-Server and Server-Client feed. But instead of replicating to the ISP, nothing happens. To close the introduce request, the server sends the contract to the ISP and the ISP generates the feed there, which holds data from the server to the client: the Server-Client feed. It is the same feed as in the server, but it is not replicated over the general replication instance from server to ISP, it is replicated from the ISP to the client. Finally, the client receives the result and generates the feed, which contains the data from the client to the server: the Client- Server feed. This feed is also replicated the normal way to the ISP. So, the feed-pair for client-server communication is normally replicated between client and ISP, whereas between ISP and server a new way of replication is given.

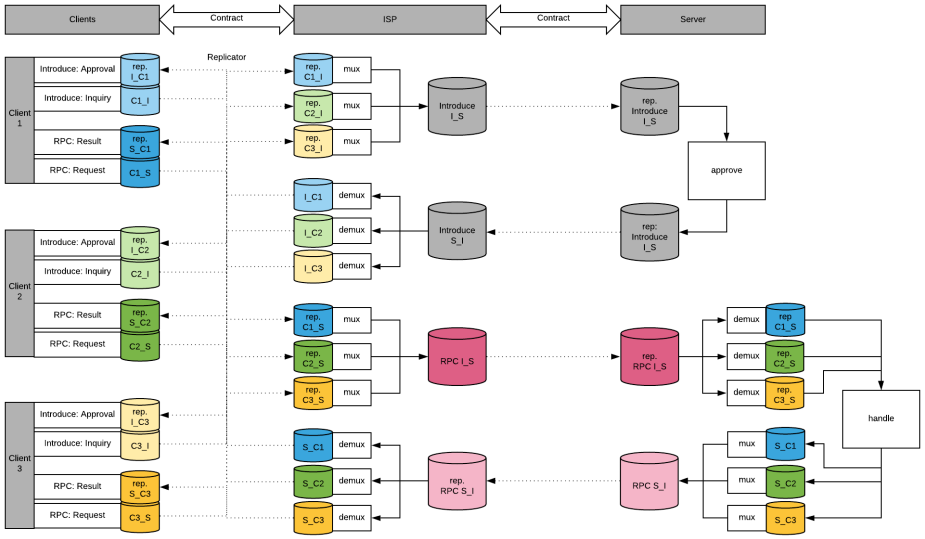


Figure 3.5: multiplexing

3.6.2 Multiplexing and Demultiplexing

As far as the communication between a client and a server is concerned, requests from the client are transferred to the ISP the same way as before. Instead of just forwarding the updated feed, by replicating to the server, the ISP detects new log entries and multiplexes

these into a new log entry. More specifically, the ISP generates a log entry signed by itself, containing the whole log entry signed by the client. This log entry is written to the ISP-Server feed and replicated. The server detects the change on the ISP-Server feed and takes this log entry. The multiplexed log entry, which belongs into a client-server feed, is extracted and appended to the Client-Server feed. This step is called demultiplexing. At this point, the situation is the same as before. A change in the Client-Server feed is given and the request is executed. The result is written to the Server-Client feed and not replicated again over the ISP directly to the client. From there, the whole story is repeated. The log entry is multiplexed once again in the Server-ISP feed and when it arrives at the ISP, it is demultiplexed and appended to the Server-Client feed. From there, it is replicated to the client and the client receives its result for the request.

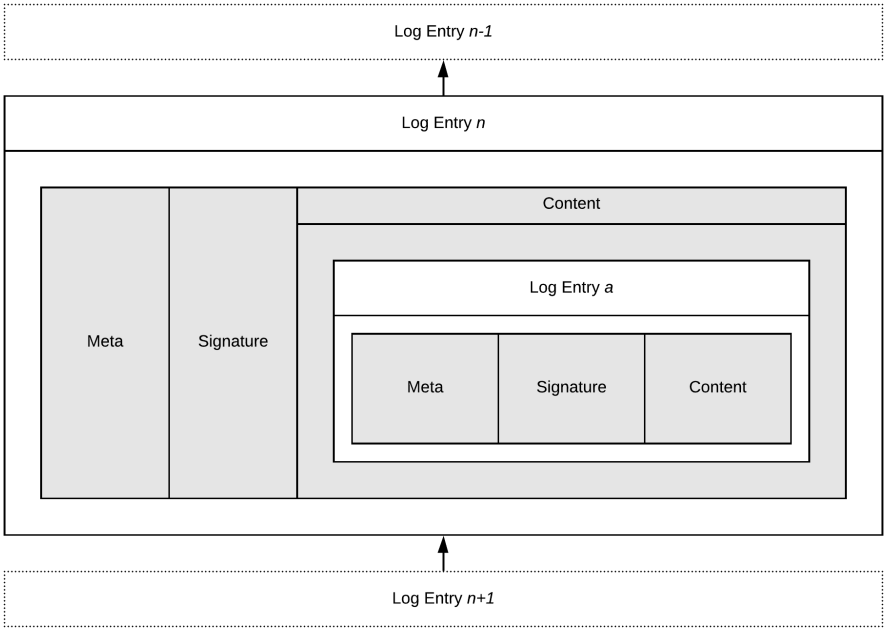


Figure 3.6: A log entry inside another log entry.

3.7 Outlook

Having all these concepts and architecture, we see the whole process is simplified on a single central ISP. In the real world, this is not the case, but as proof of concept it is more than sufficient. In the next steps, the system must be expanded by splitting this same internet service provider into a network of internet connectivity providers, which act as effective connectivity stations. An additional architecture must be developed where clients and servers can connect to these connectivity nodes. Adding more ISPs necessarily means that more internet service provider companies will be needed. The dynamic of contracts between ISPs will be explored, but more in the Future Work section.

4

Implementation

How a prototype implementation could be realised by applying the above outlined concepts and architectures will be discussed in this section. This implementation or more accurately, pseudo code, will give a broad overview of what can be derived translated from the concept into the software, by discussing the key elements. All these descriptions are so far known 'best practice', but do not tackle all software-architectural concerns of the FBP as long as the concept and consense is not violated.

4.1 Contracts

Implementing the contracts in a prototype implementation is significantly easier than it would be for a real-world application. As mentioned above, it consists of the knowledge of each other and where they are located.

Client-Contract	value	ISP-contract	value
actual public key:	cli001	actual public key:	isp001
actual private key:	******	actual private key:	******
Client-ISP feed ID:	cli001_isp001	ISP-Client feed ID:	isp001_cli001
ISP public key	isp001	Client public key:	cli001
ISP-Client feed ID	isp001_cli001	Client-ISP feed ID:	cli001_isp001
ISP location:	131.152.158.21:5000	Client location:	131.152.211.12:5000

In this table we can see a basic contract with all the information needed. This contract can be broken down even more, since the feed-IDs are just appended public keys. In the public and private key fields a first abstraction to the real-world application is made, the public and private keys would most likely be some 256-bits long integer key pairs for example Curve25519[14] . This would result in a 64 digit long hexadecimal representation. The terms here act as simplification and easier distinguishable keys. Having this setup for example with a Curve25519 key pair, it is best practice to store them in a secrets or key file. Therefore the most basic contract can look like this:

[14] Quelle

Client-Contract	value	ISP-contract	value
key file:	cli001.key	key file:	isp001.key
ISP public key	isp001	Client public key:	cli001
ISP location:	./isp001/	Client location:	./cli001/

This even more simplified base, which runs on a localhost over the filesystem, can be stored in any suitable file type and the rest of the contract, such as the feed-IDs can be built by the program, if the rules are defined over the entire network. These contracts must be stored somewhere in the software, but the implementation can be freely chosen.

4.2 Replicated Feeds

This implementation was developed by Prof. Dr. Christian Tschudin. It is a very simplified version of an append-only log in the .pcap format, generated from a Curve25519 key pair. Every log entry is signed by some private key, which leads to integrity, but not security. The entire security aspect was omitted from this thesis.

4.2.1 Structure

The Feed is a list of log entries. Each log entry consists of three main parts: meta data, the signature and the content. The meta data holds the information about the current log entry, such as the feed-ID of its feed, its sequence number, which is the internal position of the log entry in the feed, a hash reference to the log entry before and its own hash value of the content for the next log entry to reference. Next comes the signature, which signs the meta data. The content part is what is actually put into the log entry. Since all the information is stored in the cbor2[15] format and held in a pcap file, the result is a binary array which holds important properties useful for the bundling. Either a new log entry can be written with a key or an existing log entry can be appended to the binary array without validation. This mechanism is a key feature for the bundling aspect. *BILD*

4.2.2 Replication

The replication mechanism is invoked after each write operation to a feed. Generally speaking, this could be realised easily with TCP or UDP in a real network. Since this basic implementation has no connection to the internet, it replicates feeds in the filesystem. This was solved by simply copying the feed to the corresponding folder given in the contract.

4.3 RPC

Having the contract and replicated feeds, the type of RPC-protocol has its turn. To communicate between two participants, four general methods are needed as listed below. By having a simple serialisable data structure for requests and results, it is easy to incorporate

[15] Quelle

them into the feeds. Requests call services, which use the given attributes and produce a result, as advanced in the original RPC-protocol by Birrell and Nelson [1].

4.4 Data Structure

A suitable data structure or format is a dictionary or a JSON-String, having keys that reference a field, as well as being serialisable. In this structure, an ID must be given to distinguish repeated reqests, as well as map the results to their request, a type has to be set to distinguish request and result, further the service to be called is needed as well as the attributes or the result of the call. This results in a minimal set of keys for request and result:

```
request = {'ID':0, 'type':'request', 'service':'echo',
           'attributes':['An␣echo']}
result = {'ID':0, 'type':'result', 'result':'An␣echo'}
```

Having the ID as an identifier, the caller can look up the request made and map the result to the call.

4.5 Services

Services are the procedures called by the caller and executed by the callee. In the feed bundle protocol there are some key services which have to be considered:

- echo - Tt just returns the given attributes.
- get_service_catalog - The caller needs a list of all services the caller has available.
- introduce - This service passes a request to the server specified in the attributes and introduces the caller(client) to it and signs a contract.
- detruce - This closes an previously established contract and ereases all information built on it.
- get_servers - A list of servers is needed to call the introduce service. Since the caller has no knowledge of servers, this is essential.

4.6 API

As a disclaimer, these API methods correspond only to the logical aspect of the pseudo code. There are many ways to implement them in differing software-architecturial styles, the implementation only shows what is minimally needed and executed.

4.6.1 Send Request

```
def send_request(destination: Feed, service: str, attributes: list):
    request = build_request(ID=get_next_ID(),type='request', service, attributes)
    destination.write(request)
```

```
    destination.replicator.replicate() #neede if write() does not replicate

    wait_for_resolution(request) #keep track of pending requests
```

The send_request method needs to have knowledge of the servie to be invoke and its attributes, as well as a destination. As described above, it is not a "sending" in the old-fashioned way, the destination corresponds to the Caller-Callee feed where the caller is the source and the Callee is the destination. Given these parameters, a request can be formed and written to the destination feed, whereby the feed must be replicated afterwards.

4.6.2 Read Request

```
def read_request(request: dict):
    ID, service, attributes = extract(request)
    result = invoke(service=service, param=attributes)
    return request, result
```

This method needs to follow on a detected feed change. This can be realised either by a global polling mechanism, which invokes the read_request method, or directly in the method by listening to a feed. It takes the request, extracts its contents and invokes the specified service and waits for its result. Afterwards, it either returns the result or passes it directly to the send_result method.

4.6.3 Send Result

```
def send_result(request:dict, result):
    result = build_result(ID=request['ID'], type='result', result=result)
    destination = request.get_source()
    destination.write(result)
    destination.replicator.replicate()
```

This method is very similar to the send_request method. It uses the previously executed request to gain information about the ID, forms the result and writes it to the Callee-Caller feed.

4.6.4 Read Result

```
def read_result():
    response = feed.listen()
    ID, result = extract(response)
    return ID, result
```

This method also to follows on a detected feed change, takes the result and closes the request. Due to the fact that quite some time has elapsed since the send_request method, there is also a threading issue. Whether the send_request method is blocking or not depends on the overlay of the FBP and cannot be decided in general, although it is blocking in the implementation of Birrell and Nelson [1]. Therefore, it needs to be taken into account, but not considered at this API level.

4.7 Bundling

Leading to the bundling key element, the other components were laid out and specifically designed for this. Two approaches to bundling can be made. One consists of a single feed-pair between ISP and the server where all the communication happens, the other involves two feed-pairs where one acts only for request from the ISP to the server and the other only

for multiplexed client requests. In this section, the first approach is discussed, since they do not differ significantly.

4.7.1 Introducing and Detrucing

As seen in the RPC implementation, a client calls the introduce-service with the server it wants to be introduced to. At this point, the ISP holds a request from the client. The ISP does an RPC-request by itself to the server as followed:
Client request:

```
client_request = {'ID':0, 'type':'request', 'service':'introduce',
                  'attributes':'ser001'}
```

ISP request:

```
isp_request = {'ID':0, 'client_request_ID':0, 'type':'request',
               'service':'introduce', 'attributes':{'public_key':'cli001'}}
```

In this request, everything the server needs to know is given. After creating the entire feed-pair and setting up the contract, the result is returned to the ISP.

```
isp_result = {'ID':0, 'client_request_ID':0, 'type':'result',
              'result':{'public_key':'ser001',
                        'client_server_feed_ID':'cli001_ser001',
                        'server_client_feed_ID':'ser001_cli001'
                        }
             }
```

With this information, the ISP can build the Server-Client feed and replicate it, since the location of the client is already known. Afterwards, the client receives their result on the initial introduce-request and can build the Client-Server feed and replicate it to the ISP.

```
client_result = {'ID':0, 'type':'result',
                 'result':{'public_key':'ser001',
                           'client_server_feed_ID':'cli001_ser001',
                           'server_client_feed_ID':'ser001_cli001'
                          }
                 }
```

This implementation corresponds to a client, ISP and server, which do not know how the other lables their feeds, nor over which public key the connection will be handled. This lays the base for several ISP nodes discussed in the Future Work Section.

4.7.2 Multiplexing

Having now an introduced client we need to have a look at the communication between it and the server. As previously mentioned, there is no direct feed replication between the client and the server over the ISP. Instead the requests are taken from the Client-Server feed at the ISP and multiplexed into the ISP-Server feed. Afterwards, this same request is demultiplexed from the replication of the ISP-Server feed in the server and transferred to

the Client-Server feed representation in the server.
In order to achieve this, a new type is accepted and a new field is added in the RPC-data structure: the mux-type and the meta-field. Here an abstraction of the structure of a 'mux-request':

```
mux_request = {'ID':1, 'type':'mux',
               'meta':{'feed\_ID':'cli001_ser001'},
               'request':{'ID':1, 'type':'request', 'service':'echo',
                          'attributes':'hello␣server'}
              }
```

The mux type ensures that the log entry is not designed for the read result method of the RPC and all additional information needed by the server is given in the meta field. Any server reading this knows in which feed the request belongs and writes it into this feed and performs the requested service form there. The result is as followes,d written in the Server-ISP feed:

```
mux_result = {'ID':1, 'type':'mux',
              'meta':{'feed_ID':'ser001_cli001'},
              'result':{'ID':1, 'type':'result', 'result':'hello␣server'}
             }
```

Again, the meta data describes how to handle this specific mux-result in the ISP. If we approach the problem like this, one particular inconsistency occurs. When writing in feeds with this approach, the signing process is violated and the integrity broken. Hence the replicated feed offers a solution for this. Instead of extracting the request, and then rewriting it, the whole log entry is put as the value of the key request or result and at demultiplexing stage appended bytewise to the corresponding feed. *Picture* Clearly seen is the full log entry w of the Client-Server feed. By putting w directly in the log entry v of the ISP-Server feed no information gets lost or changed. This ensures integrity and replicates the feed exactly as it is: bytewise over the ISP to the server.

5 Evaluation

Evaluating a system, which takes baby steps into a completely new environment, was quiet challenging. Since most of the work consisted of coming up with the concepts and architecture, the implementation is somewhat chaotic and often not best practice oriented as in the Implementation chapter. Nevertheless, the code has been tested and important conclusions were drawn.

5.1 Testing Environment

As seen in the implementation section, the client can call services via requests from the ISP, as well as from the server, after introducing itself. Given this, the main testing aspect was to see that the ISP and servers can keep up with the work load and distribute the results back to their origins. The test was conducted as follows:
First, the implementation was tested manually by testing each functionality on its own. Second, the automated test was conducted. At the beginning, there were three nodes involved, one client, one ISP and one server. After initialising all nodes, the client went into a loop, where it first introduced itself to the server. The server automatically accepted this request and the feed-pair was created. After this, a random number was created between 5 and 20, which was the amount of service requests sent to the now connected server. To mimic a human interaction, delays of between the 1.0 and 4.0 seconds were used at first, randomly distributed with one decimal place between the RPC-requests. This was the basic evaluation of functionality, performance, reliability and correctness.
Unfortunately, after approximately 50 requests, either the ISP or the server had an exception on the cbor2 library[16] . Something with the bytestream seemed to be broken. Any other solution than just ignoring this exception and leaving this specific log entry hanging could not be found. This resulted in an open, unresolved request which the client waited for.
In a next step, after ignoring inconsistencies in the log entries, the system was tested to its limits, by having delays lowest at 0.1 seconds, adding more and more clients and up to a

[16] Quelle

thousand iterations per client. My personal machine nearly broke down during the last test with a delay of 0.1 seconds, 5 clients and a combined one thousand requests per client. The tests lead to interesting results.

5.2 Results

5.2.1 Functionality

The functionality was tested manually on different Linux distributions[17][18][19] where as on Windows and Mac OS, serious problems occured. The filesystem poller (Watchdog[20]) could not detect any changes on feeds, even when they were made. As far as concerned and tested on Linux distributions, the functionality is complete, the whole process of requesting services from the ISP, as well as introducing to different servers, is given and works as intended. This is the case only when the user strictly follows the documentation on how the system must be used. The user experience is not particulary intuitive and wrong use can corrupt the system. This as found in a very early stage of the project, but by iterating over it, these issues can be found and eliminated.

5.2.2 Performance

The performance begins with an astonishing speed, with nearly no latency between request and response, but collapses over time. This fact was already known at the implementation point. To distinguish between already handled and completed requests, the system cycles through the whole feed every time a change is detected. This problem can be solved by indexing the feed and saving the previously reached position. This factor has been left out intentionally to concentrate on the underlying concepts and architecture of the big picture.

5.2.3 Reliability and Correctness

As already indicated in the section Testing Environment, some *undefined* or *undiscovered* fault between this implementation and the cbor2 library caused some ignore requests. Having this information, the tradeoff between the reliability and correctness of the feed-bundle protocol is obvious. Having the underlaying simplified RPC protocol, the fact that request do not get a response violates the consensus given by Birrell and Nelson [1]. If it is not exactly specified, this does not line out a huge problem, but in order to achieve a fully functional system, some sort of error detection must be given. This error detection must find such "log entry loss" and resend the requests. Consequential, all these findings generalise one big problem.

[17] Ubuntu - 18.04.4 LTS, Python 3.6.9
[18] Ubuntu - 19.10, Python 3.7.5
[19] Arch - 5.7.6, Python 3.8.3
[20] Quelle

5.2.4 General Collaboration of Components

Due to the fact that this completely new technology was developed in just a few months, with a focus on the general aspects, a patchwork of different libraries, common technologies and new technologies, which are not coordinated with each other, is the result. To generally improve the very broadly open architecture and concepts, procedures(???) need to be tightened, with more rules and less freedom, resulting in a more specific implementation where the components are written to function optimally with each other. The claim in this thesis that the onboarding experience is made easier has definitely been shown. After establishing a contract with an ISP on start-up of the software the client is directly connected to this ISP and indirectly connected to the servers the ISP holds contracts with. However, the hypothesis that by splitting the ID-centric feeds from Secure Scuttlebutt into feed-pairs bundled by the Feed Bundle Protocol, the load on the wire can be reduced, can not yet be judged. The Feed Bundle Protocol is in a preliminary stage, with not enough experience in its implementation, to achieve that kind of judgement.

6
Conclusion and Future Work

6.1 Conclusion

The goal of this project was to introduce new intermediary service providers and replace the ID-centric append-only log from Secure Scuttlebutt Protocol with a feed-pairs, which hold information of the dialog of two identities in the Feed Bundle Protocol. This extension or modification allows a much easier onboarding experience, since the client is indirectly connected to all the ISP's servers after signing a contract with an ISP. With the new introduce-detruce architecture, clients can connect and disconnect to new servers in a simple manner. Within this process, new feed pairs are created which bundle all the information for that specific connection. To ease the load on the wire and the process of replicating every feed through the ISP directly to the server, requests are multiplexed into a single feed pair between the ISP and the Server. Whether this approach is more promising than the ID-centric architecture of Secure Scuttlebutt can not yet be determined. Since this system is so new and has been developed completely from scratch, there are many ways to improve it, one of which is to use the feeds properties primarily to define the state of doneness inside a feed and its single log entries. There are still many avenues to be explored and important key features to be added in order to produce a reliable Client-ISP-Server Network, some of which are discussed in the next section.

6.2 Future Work

Apart from improving the general system, we have only examined the connection between a single ISP with a single connectivity node with a random number of clients and servers connected. In the real world, however, this is not the case. There are many ISPs on the market which have connectivity stations all over their respective countries. Therefore internet service providers (ISPs) and internet connectivity providers (ICPs) can be separated. Contracts between two ISPs would also be conceivable. In the process of creating the simplified version of the feed bundle protocol, we always kept the big picture in the foreground and decisions were made keeping this in mind. Additionally, a way to combine log entries in the bundling process must be considered to effectively reduce the replication work.

6.3 Combination of Log Entries

Seen in the concept and the implementation, only a single new log entry is multiplexed into the ISP-server feed pair, resulting in a replication after each request. A different approach can be made. We can combine log entries in the multiplexing system, which means that instead of only one log entry, a defined number of new log entries or log entries generated over an elapsed time will be multiplexed to the server. This allows for more efficient replication, since the entire feed gets replicated to the peer every time. Deriving from this the multiplexing feed-pair can be split into an arbitrary number of sub-feed pairs, linked to the priority of the request.

6.4 ISPs and ICPs

As mentioned previously, the ISP was always a single node, with contracts to clients and servers. However, we could also look at the ISP as a company with a network of ICPs where the physical connection between the servers and clients takes place. In simple terms, the client and server were indirectly 'connected' through the ISP. The initial mind map drawn to lay out the concept of this thesis was a peer-to-peer Internet Connectivity Provider (ICP) network where the ISP-Company distributes the feeds internally between the ICPs. In the real world, there is a contract with the ISP, e.g. Swisscom, and this same ISP has connectivity provider stations or nodes which form a network of ICPs. This means that a client has a connection to ICP342 of Swisscom and the server has a contract with ICP903. But both have a contract with Swisscom, which provides internal replication and bundling of feed-pairs to pass information from ICP342 to ICP903. In light of this fact, a new challenge

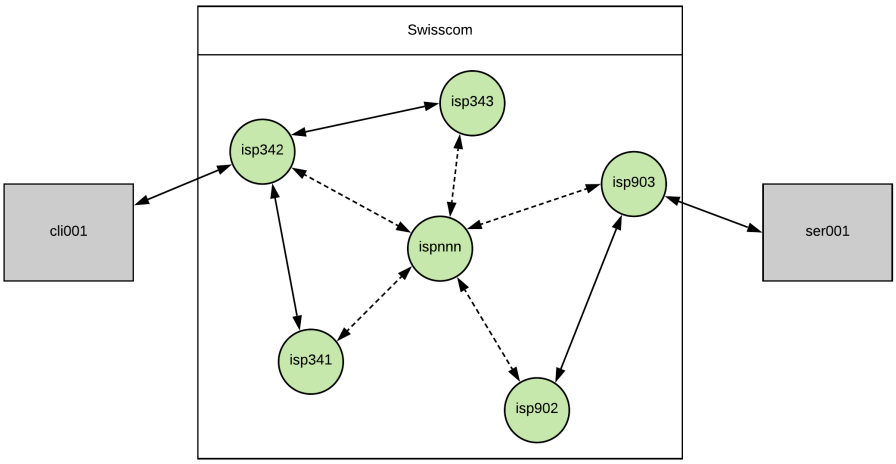


Figure 6.1: A simplified contract network.

emerges. How are the feeds replicated? How will an introduce-request happen? Can a client have several ICPs connected? Either with the same approach as is currently the case, where every ICP node stores a replication of each feed-pair which it routes to the next node, or by appending only the multiplexed log entries to the ICP-ICP feed-pair. Even a hybrid solution

can be considered and explored. Additionally, terminating a contract with one ICP should be possible for a client to change the connectivity provider, for example when traveling from Basel to Zurich. New algorithms need to be developed to handle exactly such use cases.

6.5 Contracts between ISPs

Yet again, we can take this distribution to the next level where ISPs have contracts with other ISPs. This provides a way to bypass the current requirement that each ISP must have a contract with any server in the FBP Universe with which it wants to deal. This has a very important impact on the system however, since new contracts are generated when the business aspect has not yet been defined.

Bibliography

[1] Andrew D Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, 2(1):39–59, 1984.

[2] Secure Scuttlebutt. Scuttlebot (retrieved 30.06.2020), . URL https://scuttlebot.io/docs/social/join-a-pub.html.

[3] Secure Scuttlebutt. Secure Scuttlebutt Protocol Guide (retrieved 30.06.2020), . URL https://ssbc.github.io/scuttlebutt-protocol-guide/.

[4] Dominic Tarr, Erick Lavoie, Aljoscha Meyer, and Christian Tschudin. Secure scuttlebutt: An identity-centric protocol for subjective and decentralized applications. In *Proceedings of the 6th ACM Conference on Information-Centric Networking*, pages 1–11, 2019.

[5] Z. Zheng, S. Xie, H. Dai, X. Chen, and H. Wang. An overview of blockchain technology: Architecture, consensus, and future trends. In *2017 IEEE International Congress on Big Data (BigData Congress)*, pages 557–564, 2017.

University of Basel

Faculty of Science

Declaration on Scientific Integrity
(including a Declaration on Plagiarism and Fraud)

Bachelor's Thesis

Title of Thesis *(Please print in capital letters)*:

A Feed Bundle Protocol for Scuttlebutt

First Name, Surname: Jannik Jaberg
(Please print in capital letters)

Matriculation No.: 2017-054-370

With my signature I 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.

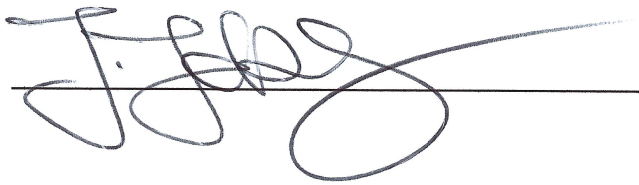
I have mentioned all source materials used and have cited these in accordance with recognised scientific rules.

In addition to this declaration, I am submitting a separate agreement regarding the publication of or public access to this work.

☐ Yes ☐ No

Place, Date: Oberwil BL, 02.07.2020

Signature:



Please enclose a completed and signed copy of this declaration in your Bachelor's or Master's thesis .