ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

SEMESTER PROJECT

CISC Cothority Identity Skipchain SSH Interface

January 17, 2017

Responsible:
Prof. Bryan Ford

Author:
Andrea Caforio

Supervisor:
Linus Gasser

Decentralized and Distributed Systems Laboratory
École Polytechnique Fédérale de Lausanne

DEDIS

I. Introduction

Identification has become a crucial part of the digital world especially in the Internet where authenticity is one of the most important safety measures. Nowadays it is hard to keep track of all different passwords and public keys that we use to authenticate ourselves to different services. In the worst case the same key is used for several services.

In order to solve this issue, keys must be regularly updated and rotated, which is cumbersome when multiple devices come into play and basically unattainable without the help of specialized software.

The Cothority project at DEDIS fills this gap using an approach that resembles the Blockchain mechanism that lies at the heart of the Bitcoin project. In the case of the Cothority the Blockchain is a double-linked list, called Skipchain, that is able to store arbitrary data. The backward links between the blocks preserve the order of the list and consist of cryptographic hashes of the preceding block. A forward link is only added when a newly proposed block is accepted by a majority of participants. Those entities are called managers and are usually user devices like Laptops or Smart-phones The details on voting on a change in the Skipchain and verifying the integrity of the entire data structure are describe later in the text.

For a user this means that he should be able to rotate a key for security reasons, introduce new data to the Skipchain as well as delete obsolete data. The data we are working with in this project are public/private key pairs in particular secure shell (SSH) keys that are widely-used for remote logins and authentication on websites like Github, however this is a merely self-imposed restriction. As mentioned before the Skipchain is able to store basically any type of the data which leads to possible use-cases involving PGP keys instead of SSH-keys or even ordinary passwords or software updates.

Whenever a user requests a change, due to key rotation or lost/new devices, he initiates an update to the Cothority notifying the other devices in the Skipchain, which in turn then have to agree on the proposed change by transmitting their digital signatures (voting) of the updated block. If a certain threshold of votes is reached the change is accepted into the Skipchain. This is synonymous to appending a new block to the chain. Like that a potential attacker who gets hold of a device is not able to interfere with the Skipchain in a harmful way without also controlling at least the threshold amount of devices. Clients, sometimes also called followers, like Github can then easily track the Skipchain and are notified upon every change. Furthermore, because the Cothority service is distributed, clients can poll several servers in case one or more go offline or the connection drops, basically making the service very resilient.

By using the Cothority and its Skipchain a user should be able to comfortably manage a unique SSH key for every device he owns, which in return yields a great enhancement in security and corresponds well to the recommended practice on how to handle public/private key pairs in the wild.

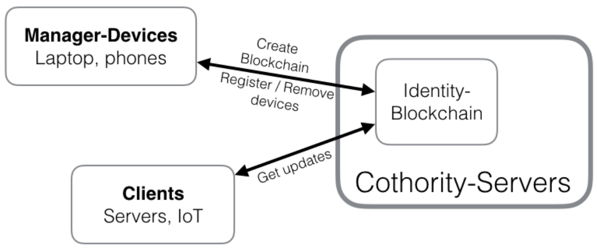


Figure 1: Abstract depiction of the Cothority interacting with devices and clients.

In the subsequent section this report will describe how the analysis of the problem was conducted and what tools for the later implementation of such key-storing application on both client side (Android, Java) and server side (Cothority, Golang) were chosen. Further sections are covering the challenges and problems that arose during the implementation, the strategies chosen to circumvent said issues, a detailed description of the final product as well as its limitations and possible future enhancements.

II. Analysis

The Cothority comprises a large and sophisticated code base almost entirely written in Google's Go programming language. A fully functional network communicates with the help of a protocol that in the early days relied on Protobuf but is wired in the meantime over web sockets. [4, 5] The first step on the road to this semester project was therefore acquiring some picture of the internal workings of the Cothority project with a shy goal of establishing a network connection to a running node. At this early stage of the project it wasn't really clear which framework would be the most suitable. The suggestions went from some multi-platform technologies like Meteor, which would have facilitated the creation of single application that could run on both Android and iPhone as well as in any browser using JavaScript, to a plain Java approach targeted to Android handhelds.[18] However it rapidly became clear that nothing other than the latter was worth further considerations. This was mainly due to the complex way messages are exchanged between a potential device and a node of the Cothority, which involves non-trivial cryptographic measures (i.e. signatures) that turned out to be too cumbersome in order to be handled by languages that don't offer stable support for network communication and low level bit manipulations.

The next hurdle came in the form of the Protobuf serialization protocol used to push messages into the network. Since it is already integrated as a crucial part of inter-node communication it seemed natural to use this interface also for exchanging messages with an Android client. It actually turned out to be a harder problem than choosing a fitting platform. The way the Cothority implements the protocol buffers does not allow for communication with anything other than a node in the network. This meant that a substantial amount of the code base needed to be rewritten to make such a thing possible. Clearly, this was a dead end an other solutions had to be found. In the end, we settled for a more crafty but still simpler way by using the ever more popular JSON objects instead of Protobuf. By gluing a little proxy message receiver onto the existing network stack of the Cothority enabled for the first time to send and receive messages emitted from an Android device. On the handheld side things looked far more comfortable. Google's Android developer suite has reached such a maturity that a application almost programs itself, coupled with the unbeatable richness of the Java environment fast progress could be made.[10] Still some aspects of the application needed some special attention. For example handling network connections required a robust and error-free handling of threads and processes inside the Java run-time such that a user is completely oblivious to anything that could block the graphical user interface. Further, because every communication is strictly initiated by the Android device and then acknowledged by a Cothority node, the HTTP-protocol had to be sketched out very precisely. Luckily, there were already some stubs for a potential future API inside the Cothority code base, which made the design of said protocol a breeze. This all made the creation of some simple proof-of-concept application possible within some few weeks. Nevertheless, the application as well as the Cothority hadn't been doing anything productive up until this point. The simple exchange of serialized JSON messages was still a far cry from an application that could verify a Skipchain issued by the Cothority and manage SSH-keys with the help of it.

The next section is therefore dedicated to the implementation details of the various parts of the system that went on the be a working application fulfilling the duties mentioned before.

III. Implementation

The implementation details for both server side and Android client are discussed in separate subsections followed by a little detour about the involved cryptography parts in particular the signature of block and the important verification of a complete Skipchain..

A. Server

As previously mentioned the Cothority is extended by a basic HTTP-server whose only job is to intercept incoming request on a different port than the sockets used for inter-node communication are listening on. Since Golang offers a rich environment for concurrency and parallel computing its standard library naturally includes fully functional HTTP-server implementation that can be set up with one single asynchronous call.

```
go func() {
  http.ListenAndServe(addr, server)
}()
```

Subsequently, functions can be attached to the server interface that listens to one specific path in the URL. Those helper functions then perform the actual work by receiving the raw string in JSON format and parsing its entries into intermediate structures. [9] This is necessary because Go's JSON implementation is not able to serialized complex data structure like hashes or public keys. In order to navigate around this issue all message fields which are not primitive types like strings or integers are encoded into base64 before being passed over the network. Now when the server receives such a message it reassembles its original content and converts the intermediate structures into their proper equivalent that can be understood by the interface associated to the Cothority. For example, performing a vote on an updated block consists of sending a serialized JSON object containing the user's signature of the block to the Cothority, which then would look similar to the following code extract.

```
ProposeVote {
  ID: "YXNkZmdoa2xtbG..."
  Signer:  "user1"
  Signature: "MzJycWtlZmFqbz..."
}
```

Here the ID is a 32 byte base64 representation that identifies a specific Skipchain handled by the Cothority. More precisely it's the hash of the very first block, also known as genesis block, of the chain. The signer field marks the name of the user device that emitted the message along with his signature of the block in question. also in base64 representation. We will see shorty how exactly a user needs to perform a signature in order to cast a vote. In case the received message is not corrupt and contains correct data the server will answer with a generic 200 OK response code back to the Android device. Errors are treated similarly by sending standard 5** error codes which then trigger appropriate measure within the Android application. [23] Special care must be taken when handling the public key strings in a message. Here a standard base64 decoder is not sufficient enough anymore to recreate a key. Fortunately, DEDIS has an own cryptographic library entirely written in Go that is able to tackle such tasks.[3] As soon as a successfully converted message is passed to the interface the node that was contacted by the device spreads the message among the other nodes of the Cothority. Because an Android device can only contact a single node the response from the Cothority must be sent from the same node that was contacted by the device. How exactly a Cothority is exchanging messages remains completely hidden from the user and cannot be influenced by him.

Whenever a user wants to add a new device to the Skipchain he scans a QR-code [6] spawned by a node onto a terminal screen or scans it from a device that is already connected to an existing Skipchain. By doing so he receives not only the address and port of this particular node but also the ID of the Skipchain that he wants to join. For the moment those encodings are printed onto a terminal screen but in a further more user-friendly way they could also be issued by a web interface. For creating and printing those codes a separate library has been created for this project. [20]

To conclude this section about the server, some arguments why exactly the HTTP protocol was chosen over other maybe more flexible alternatives likes web sockets. The main reason is that all the communication that is necessary between a node and an Android device boils down to a purely ACK/SYNACK handshake type of interaction where the handheld opens a connection to a node, sends his requests and then waits for the response. This is what HTTP does best. Furthermore there is no need to establish a persistent connection for the exact same reason. In short, there was no urge to dabble in more experimental network features.

B. Android client

The implementation of the handheld application was significantly more involved than the HTTP/JSON extension to the Cothority. Not only because of the lack of experience in creating Android applications but also in terms of the amount of code that had to be written.

After sketching out some possible arrangements of the user interface and testing their looks on a real device it was clear that coding the GUI wasn't where the focus should lie. Instead it should be kept so simple that it would not interfere with any other internal part of the application. It boiled down to having a few very basic Activities (windows) that are powered by little code. Those activities are described in section five of this report. The application itself is meant to preserve and store data thus enabling the user to close the program without affecting its current state. In other words, the current configuration of a Skipchain needs to be stored on the disk. Amongst Android developers sqlite file based database is the de facto standard when it comes to storing persistent information but since this project only requires some marginal amount of data to be kept on the disk there was a simpler solution.[17]

Shared preferences are normally used to store user settings. It's nothing more than a key-value storage where an unique key can be mapped to either a string or to one of Java's primitive values.[16] In our case that meant that by using said shared preferences the address and port as well as the Skipchain configuration and the cryptographic key pair could be saved over multiple sessions and up until the users decides to update them or joins a new Skipchain.

Apart from signing and verifying, which will be discussed in the following section, the actual network features with the HTTP core are the most interesting part of the application. The big challenge here lies in the fact that Android mandates the use of concurrency when integrating more complex functionalities but also offers several techniques to handle them without much hassle. In more detail, every Android application runs in a single thread on the JVM. This thread essentially powers everything related with the graphical user interface. If now computationally heavy tasks are run in the same thread the application will in the best case just stall the screen but more often simply crash. This is were asynchronous tasks are a valuable support.[12] Similar to Futures in Scala or Promises in JavaScript asynchronous tasks allow for spawning new threads within the Java run-time hence decoupling the GUI from unrelated computations. For the application it meant that for every new HTTP request a fresh task would be dispatched. By registering observers the Activity is notified when the server responds and can then update its GUI before terminating the task. More precisely, every message type is encapsulated in a class and passed to the starting HTTP task by the Activity. A message class carries a private class representing the JSON message. This private inner class can be serialized a passed to the network. On the other hand incoming messages can be parsed into such private inner class for further usage because the message objects correspond exactly to the intermediate structures on the server side. All of this is achieved by Google's GSON library that is capable to even convert complex objects into JSON strings and vice versa. [15]

For the sake of completion a listing of the entire communication protocol used to exchange messages between a Cothority node and an Android device.

```
ConfigUpdate {
  ID, String
}
```

This is the basic polling message only containing the ID of the Skipchain encoded as a base64 string. If there exists a Skipchain withing the Cothority corresponding to the given identifier the node will respond with a Config message.

```
Config {
  Threshold, Integer
  Device, Map<String, String>
  Data, Map<String, String>
}
```

A configuration contains next to the Skipchain threshold also dictionaries (maps) with all the current devices attached to the chain as well as their public keys and SSH keys. This data structure is essential later when it comes to performing signatures upon a proposal, since a proposal is nothing more than a disguised Config message. But before we are able to vote on changes we have to join an existing Skipchain by sending a ProposeSend message. The same is done when performing an update.

```
ProposeSend {
  ID, String
  Propose, Config
}
```

Due to the fact that the ProposeSend message carries a Config fields every request of such type is always preceded by a ConfigUpdate message yielding the current configuration. When there is pending proposal the current configuration differs from the proposed one, thus to check if some change has been introduced to the Skipchain we need a separate message type to request a proposal configuration.

```
ProposeUpdate {
  ID, String
}
```

It may also be possible that there is no proposal pending. In such a case the Cothority will respond with an empty string to a ProposeUpdate message. After all no proposition would ever be accepted into the Skipchain if there wasn't a message representing a vote.

```
ProposeVote {
  ID, String
  Signer, String
  Signature, String
}
```

In case there is a pending proposed configuration from himself or another device a user can vote on it by signing the proposed configuration structure with his public key. The specific nature of the signature is described in the following section that discusses the entire verification process. The Cothority keeps track of how many votes have already been cast on a proposal and appends the new block to the Blockchain as soon as the threshold is reached. At this point a ProposeUpdate message is met by an empty string from the Cothority.

Additionally, the information that is used to generate the QR-code is also kept in a JSON object, whose string representation can be retrieved from the encoding. The remaining messages all involve the verification of a Blockchain and are explained in the following section.

C. *Verification*

Being able to verify a given Skipchain is crucial for the integrity of the data handled by the Cothority. One imagines a Man-in-the-middle attack where a hoodlum could tamper with the Skipchain and modify its data. Further, the Cothority doesn't want to allow votes from unknown sources hence also voting is part of the verification procedure, which commences the first part of this chapter.

As we already know, whenever a device requests a change in the Skipchain it proposes its update to the Cothority which in turn distributes it to their connected devices for voting. However to be very precise not only the user devices perform the votes but also the individual nodes of the Cothority. This procedure is significantly more involved and is explained in all detail in the CoSi paper of the DEDIS laboratory.[2] For the device signatures the Cothority demands a so called Schnorr signature first introduced in 1988 by Claus-Peter Schnorr most prominently used in the Bitcoin project. [25] A Schnorr signature exploits the discrete logarithm problem and bears therefore similarities to other asymmetric signing schemes like RSA. [22] But before any signature can even be created we first need to generate the private and public key pair. For this the Cothority requires the utilization of the Ed25519 public key signature system whose keys are rather small and rapidly generated.[7] The Edwards curve implementation is part of the popular NaCl library written in C, but since Android is based on Java the application deploys an open-source Java port, which does the job just as well.[21]

Having generated the key pair we can finally produce our signature for a proposed block that is then send to the network. When the Cothority witnesses enough valid signatures it accepts the proposed change into the Skipchain.

It follows a description of the underlying mathematics the lead to a successful Schnorr signature and its subsequent verification.

Let G be a base point of the twisted Edwards curve, Ed25519, and x, k elements of the integer multiplicative group modulo q, where q is a prime. Calculating $X = G^x$ yields the public key hence x is our private key. To actually sign a message we proceed with $r = G^k$ and $e = H(M||r)$, where M represent our message, H is a hash and $||$ denotes the concatenation operation. Computationally, those values are kept as bit strings. Now performing $s = k - xe$ leaves us the signature pair (s, e).

The verification step is then just the reciprocate of the previous calculations. After receiving the message M and the signature pair (s, e) we calculate $r' = G^s X^e$ followed by $e' = H(M||r')$. If $e' = e$ the verification is successful.

For the sake of completeness we proof the correctness by setting $r' = G^s X^e = G^{k-xe} G^{xe} = G^k = r$ therefore it follows that $e' = e$.

More insight into Schnorr Signatures can be found in the original paper from 1988.[1]

After having discussed the internals of the voting procedure we can now turn our attention to the actual verification of an entire Skipchain. In order to do that we first need to establish the remaining JSON structures used to transfer a Skipchain from a node to a device. A verification always starts with an UpdateChain message.

```
GetUpdateChain {
  ID, String
}
```

The GetUpdateChain message is congruent to the ConfigUpdate and ProposeUpdate messages we have seen in the previous section. The response however is differs significantly.

```
UpdateChain {
  Update, Skipblock[]
}
```

The UpdateChain response contains a list of Skipblocks that make up the Skipchain. A Skipblock itself is another message type.

```
Skipblock {
  SkipblockFix, SkipblockFix
  Hash, String
  Signature, String
  Message, String
}
```

A Skipblock contains next to its identifier also an aggregate signature of all the nodes in the Cothority and the message which is effectively a hash of the block. Further there is an additional data structure SkipblockFix that contains more information about the particular block. The SkipblockFix fields contains another object with more data concerning the block.

```
SkipblockFix {
  Index, Integer
  Height, Integer
  MaximumHeight, Integer
  BaseHeight, Integer
  BackLinkIds, String[]
  VerifierID, String
  ParentBlockID, String
  Aggregate, String
  AggregateResp, String
  Data, String
}
```

Here we mainly care about the list of back link identifiers and the aggregate public key of the Cothority. Also it important to stress that the order of fields is crucial for the later calculation of the block's hash. At this point we are ready to talk about the three methods that are used to check the integrity of an entire Skipchain on an Android device.

1) Back Link Soundness: Naturally, the first check that should be conducted, is to make sure that the doubly linked list of Skipblocks is well-aligned. Namely, does every block contain the hash (back link) of the previous block. The back link to the preceding block is always at the head of the BackLinkID list inside the SkipblockFix structure.

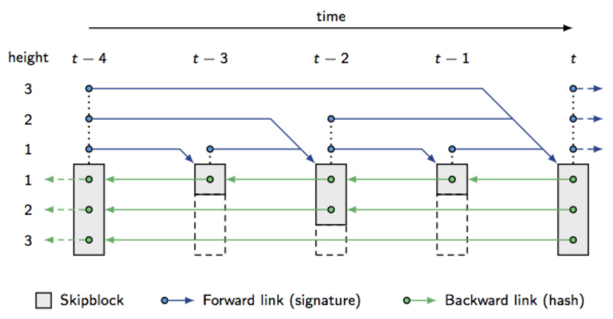


Figure 2: Time line of a Skipchain

2) Block Hash Correctness: In the next step we need to check that each individual block's identifier corresponds to the hash of its SkipblockFix. Here we need to pay attention that every field of the SkipblockFix is added at the right position and with the right byte size to the SHA256 hash. In other words the first fours fields are little endian eight byte integers and all the following fields are binary blobs which first need to be decoded out of the base64 encoding described in section two. At the end the two hashes of the block's identifier and the Skipblock need to be identical.

3) Signature Soundness: Finally, all that is left to be able to confirm the integrity of the Skipchain is to check the signature of each block. As seen above every Skipblock comes with a signature and its SkipblockFix contains the aggregate public key of all the nodes in the Cothority. By using the before mentioned EdDSA library we can easily check if the signature of the message corresponds to the aggregate public key.

IV. Application

In this section we are going to take a look at the overall Android application that implements everything discussed in the previous sections. The final release candidate for this semester project comprises five activities (screens) that make up the user experience. In a broad sense the application implements the following state machine in which each state represents an activity and each transition is triggered by a user interaction.

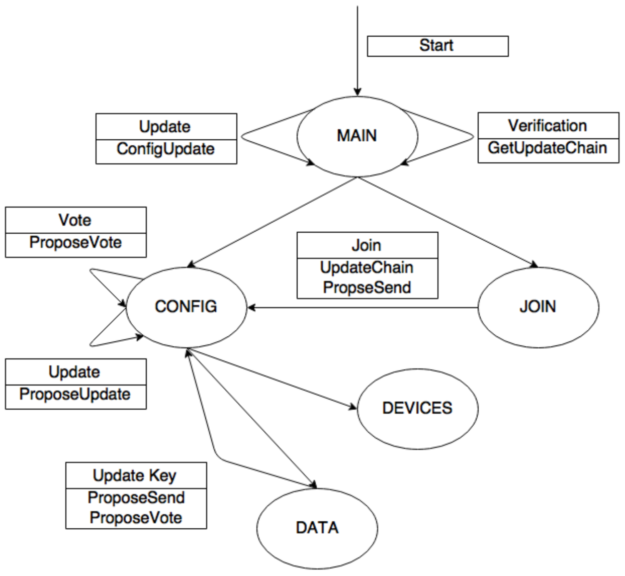


Figure 3: State machine of activity flow

A. Main Activity

The main activity marks the entry point to the application. It offers the possibility to start up the QR-code scanner to join an existing Skipchain, which triggers a propose send, propose vote cascade to introduce the device to the Cothority. Further the network status can be checked by sending requesting a configuration update from the node. In case of a functional connection the hash of the Skipchain identity as well as the host address and port numbers are used to encode and draw a QR-code onto to screen. This code can then be used by other devices to join the Skipchain instead of scanning it from a terminal screen. This is mainly done to increase user-friendliness by visually indicating the successful establishment of a connection. Last but not least the main activity is also used to engage the Skipchain verification process, which for the moment has to be started manually by the user and is not automatically performed after every network transaction. It is also important to note that the main activity is able to remember the Skipchain state for an ongoing connection after the application has been closed and reopened again. Like that no information is ever lost and a simple configuration update request brings the application to the most recent state.

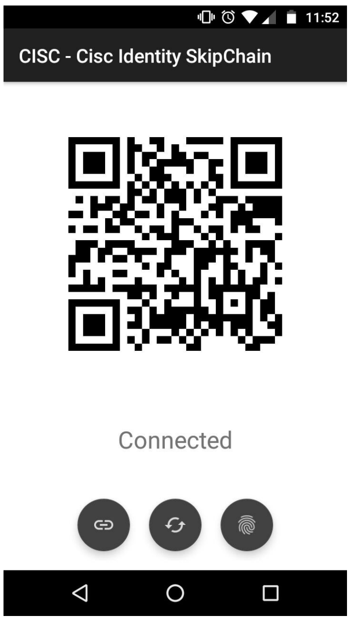
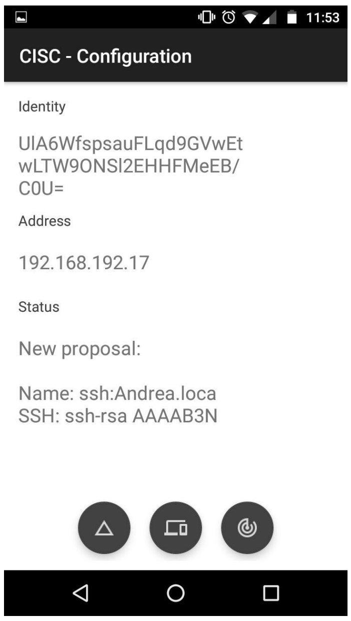


Figure 4: Screenshot main activity



Identity

UlA6WfspsauFLqd9GVwEt
wLTW9ONSl2EHHFMeEB/
C0U=

Address

192.168.192.17

Status

New proposal:

Name: ssh:Andrea.loca
SSH: ssh-rsa AAAAB3N

Figure 5: Screenshot configuration activity

B. Join Activity

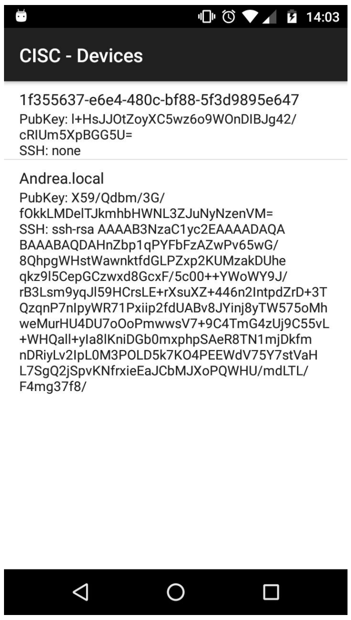
Technically, the QR-scanner isn't a proper activity since it does not load any GUI, instead it only fires up the camera, which then calls an external library whenever it comes across a QR-code. After a successful join proposition the join activity starts the configuration activity.

C. Configuration Activity

The core of the application is found in the configuration activity, which is opened by clicking the QR-code in the main activity. From here an user is able to interact with the Cothority. This includes initiating the renewal or adding of a SSH key, which is then is done in a new activity described further down, and navigating to yet another activity to that will display all the devices currently connected to the Skipchain. However most importantly, the configuration activity is able to fetch the newest proposed block and to display the main information about the proposal. A user can then vote on it by simply pressing the text box containing the proposal string. This clickable text field is disabled when after a successful vote or when no proposal is pending. The same text box is also used to indicate the current status of the Skipchain. Those states range from 'Threshold not reached', when a proposed block is waiting to be accepted into the Skipchain, to 'Skipchain up to date', when the Cothority is not handing any change requests. An user will probably spend most of his time in the configuration activity.

D. Devices Activity

As mentioned before the devices activity displays a complete list of all devices with their associated public and SSH keys that are attached to the Skipchain. This is a purely static activity that offers no interactivity with the user or the network. Also since SSH keys can get quite long they may be shortened to a certain amount of characters similar to the way Git shortens its commit hashes.[8]



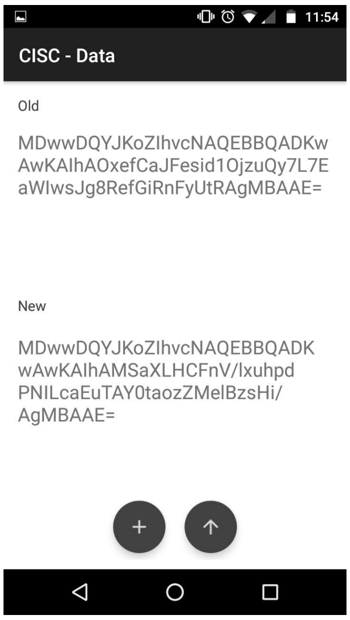
1f355637-e6e4-480c-bf88-5f3d9895e647
PubKey: l+HsJJOtZoyXC5wz6o9WOnDIBJg42/
cRIUm5XpBGG5U=
SSH: none

Andrea.local
PubKey: X59/Qdbm/3G/
fOkkLMDelTJkmhbHWNL3ZJuNyNzenVM=
SSH: ssh-rsa AAAAB3NzaC1yc2EAAAADAQA
BAAABAQDAHnZbp1qPYFbFzAZwPv65wG/
8QhpgWHstWawnktfdGLPZxp2KUMzakDUhe
qkz9l5CepGCzwxd8GcxF/5c00++YWoWY9J/
rB3Lsm9yqJl59HCrsLE+rXsuXZ+446n2IntpdZrD+3T
QzqnP7nIpyWR71Pxiip2fdUABv8JYinj8yTW575oMh
weMurHU4DU7oOoPmwwsV7+9C4TmG4zUj9C55vL
+WHQall+yla8lKniDGb0mxphpSAeR8TN1mjDkfm
nDRiyLv2IpL0M3POLD5k7KO4PEEWdV75Y7stVaH
L7SgQ2jSpvKNfrxieEaJCbMJXoPQWHU/mdLTL/
F4mg37f8/

Figure 6: Screenshot devices activity

E. Data Activity

The last activity in the application is concerned with the creation of new SSH key pairs, which may be switched out indefinitely. The activity then also allows a user to propose the newly created key to the Cothority. After this action the application returns to the configuration activity and waits for the proposal to be accepted.



CISC - Data

Old

MDwwDQYJKoZIhvcNAQEBBQADKw
AwKAIhAOxefCaJFesid1OjzuQy7L7E
aWIwsJg8RefGiRnFyUtRAgMBAAE=

New

MDwwDQYJKoZIhvcNAQEBBQADK
wAwKAIhAMSaXLHCFnV/lxuhpd
PNILcaEuTAY0taozZMelBzsHi/
AgMBAAE=

Figure 7: Screenshot data activity

V. Limitations

lEven though this application is ready to be deployed into a circle of users who are willing to manage their SSH keys within a Cothority Skipchain it remains a mere prototype of what it could potentially do. We are going to dig into the future improvements in the following section but first we need to establish a clear picture of what the application is lacking and what technical limitations it encountered during its development.

A. Background Updates

At the moment the application only synchronizes with the Cothority after a user interaction meaning the user is not notified about a change until he specifically asks for it. This is of course not a state of the art technique in a era where push notifications are basically part of every other application but since a lot of development time went into hacking the Cothority server to accept messages from external devices, it was simply not feasible to tackle this feature in the given time.

B. Update Rejection

Currently it is not possible to reject a proposal of an updated block. This means that if a proposition is not acceptable by a user it will stall the Skipchain state until the Cothority decides to remove the proposed block from the server. This can be a source of frustration since the user won't be able to apply change to the Skipchain until the proposition has vanished.

C. Simultaneous Requests

While voting on a proposal is great, it does not imply that there will always just a single pending change in Skipchain at a particular moment in time. Especially when the number of devices attached to a single chain with potential different users behind them we have assume that several requests can be proposed at the same time thus voting on a selection of proposal is an important part. The application as it is now does not support said feature.

D. User Interface

Although functional, the user interface has still a long way to go until it reaches some reasonable state of maturity. Especially with regards to user-friendly design al lot more could be done. The relatively narrow screen of tablet or smartphone demands that the user interface is self-explanatory, which is not really the case for our application. If a better user interface is not feasible a hand-written guide to the application can always ease the use.

E. Efficient SSH key creation

This one was mentioned earlier and is quite a source of concern. For now the application only supports the creation of 512 bit SSH key pair even though Cothority demands them to of length 2048 bits. Already the creation of a small 512 bit key pair introduces a serious amount of computational time to the application and anything bigger will just let the application crash after a few seconds of running. Clearly the Java standard library for creation cryptographic key has its shortcomings at this point. A possible solution to this problem could be to execute an external binary outside of the Java virtual machine run time.

VI. Future

Having discussed the application's limitations we can now move on to future improvements. Because there is no rule that forbids a proof-of-concept to become something that can actually be deployed, there are lot of feature that may or may not be added in the future development of this application. It is even possible that this project marks the basis of another semester project. This section gives a little insight into how the author imagines the implementation of some of the issues we saw in the previous section.

A. Exploit Android Services

A future feature has to include the integration of Android services.[11] These services permit to have background tasks running transparently in the background and across Activity life cycles. It would then be rather simple to attach a service to a sending mechanism that emits a configuration update request after a certain amount of time. Additionally this would mean that the current asynchronous task construct can be avoided by moving all the network related code into a service. Such an application would gain total independence from the user by polling automatically proposals and updates. Furthermore this could potentially be integrated alongside a mock server, which would certainly be easier than attaching a mocking facility onto an already existing code base.

B. Web Sockets as Communication Protocol

This is a large task and may make up an entire semester project itself. It is however crucial on the way to a full deployment of the application. With the latest introduction of web sockets to the Cothority, which are able to directly transmit binary data, the application's communication layer could be rewritten in such a way that it seamlessly integrates with the web sockets. Also with regards to the previously mentioned background services.

C. PGP Keys

In connection with another ongoing semester project at DEDIS that involves PGP. It would be an interesting addition to the current SSH keys. This means that a stable PGP Java library needs to be found that can safely be integrated into the project. Again this can be further extended to arbitrary data types that can be stored in with key/value pairs.

D. Secure communication

The entire hassle around authenticity remains shallow if the data is still sent in clear over the network. Currently, all the JSON messages are sent in clear text and thus are an easy target for potential eavesdropping attacks. Even though no private keys are ever emitted it is still possible to harvest the public SSH keys of the devices and reuse them for malicious activities With regards to future data types this requirement could the pivot to rather security relevant data.

The integration of an TLS layer would circumvent the mentioned problem but has to hacked on top of the already existing communication layer, which is often reason for concern. Both Java and Golang provide stable libraries to engage TLS connections.[19, 14]

VII. Knowledge gained

The actual purpose of every semester project is frankly the knowledge that is acquired during those few weeks. Therefore the following section is entirely dedicated all things the author learned and had to learn while developing the CISC Android application. We are going to split this section in two subsections discussing the practical as well as the theoretical spike in knowledge.

A. Practical

Creating an Android application means to write a lot of code but before the Cothority server part had to receive the notorious JSON interface. Since the knowledge of both the Cothority code base and the Go programming language was virtually zero at the start of this project a huge amount of code had first to be read before a single line could have been written in the first place. In other words the first two or three weeks of the initial phase in the summer were solely dedicated to reading and understanding the enormous Cothority code base alongside some relaxation tasks to acquire to needed skills to write reasonable code in Go.

At some point enough confidence was built up and the work on the JSON interface began. First it was a far cry from being a successful mission. Basically any attempt of assembling a working HTTP interceptor failed spectacularly. It was clear that one introductory course in computer networks was not sufficient to be able to write such a structure. It meant to hit the books again to read up on some basic HTTP implementations and as it turned out Go has a strong networking support built into its standard library. The consequence of this struggle was a first working JSON interface on the server side. In the end that script was rewritten several times but it was a first step in the right direction and it taught a great deal about the importance of understanding a problem before it is tackled. Furthermore it was first personal occurrence of the fact that code is really more read than written. A point every professor is iterating countless times when it comes to programming. It was the leading conclusion of this phase that stuck until the end of the project. Every code that was to be written should be lean and clean such that it can comfortably be read by others.

When it came to coding the Android application the course in software engineering had only just begun. Still it was important to apply some basic concepts and design patterns to the Java code. Especially in fields like asynchronous callbacks and communication design. It was crucial to realize that even though the code became larger with every week that overall oversight was never lost thanks to sticking to those basic principles of software engineering. However in the end every piece of code seems to be endlessly improvable but no matter how many times one rewrites a piece the uncomfortable feeling that something just does not look right never perishes thus finding the equilibrium between useful refactoring and a waste of time is a challenging task.

Writing an project from scratch yielded a valuable introduction to the various testing technologies available to probe different parts of an application. Regular unit tests are helpful when it comes to test internal program routines but they lack the ability to thoroughly work with graphical interfaces and networking. With the help of special libraries dedicated to ease work with graphical component, it is possible to test a GUI in the style of a unit test, however examining network communication strictly requires a mocking mechanism that imitates a server locally. [13] This unfortunately had to be learned the hard way after trying for weeks to test the communication protocol without a mocking interface, which in the end resulted in the absence of said mock server. This deficiency shows that tests have to be planned in parallel to the actual application and cannot be easily added to already existing code. This observation marks the corner stone of Test Driven Development, to which this semester project was a gentle introduction. [24]

All in all, the most important skill acquired was to organize yourself in such a way that heaving such a more or less large project alone was not bound to fail from the beginning. To put it in even more controversial terms, it became evident that precisely planning every single step of the project counted more than the actual coding in the end. Still it wasn't a lone wolf project after all, integrating the excellent support from my mentor helped to overcome many boundaries especially during the server phase and when trying to wrap my head around cryptographic parts of the application. In summary, the author was able to dabble in a lot of different technologies and apply them to build a working application.

B. Theoretical

In terms of theory the most important gains were made in getting some reasonable understanding of the nuts and bolts of an HTTP stack. Especially when it later came to integrating this knowledge on the Cothority server side. This ranges from the simple handshake protocol until the emission of correct error code in case of a failure also with respect to handling network tasks asynchronously. Finding the correct encoding for the various messages was another point that had to be learned and which was unfortunately something that was taken for granted prior to this project.

The introduction to the Blockchain technology not only sparked an new interest that is certainly going to live on but also made it much clearer how non-trivial data structure are handled over a distributed system of nodes. This meant that some superficial understanding of the underlying cryptographic measures had to be obtained as well. This includes the usage of elliptic curves for key pair generations as well as the technical procedures necessary to perform a digital signature on a specific data structure and why certain signature are more useful for a particular application then others. Furthermore signing and verifying digital objects also implies the usage of hash functions whose importance to the whole process was an eye-opener. With the addition of courses in information theory and abstract algebra this project absolutely paved the way to a broader understanding of how cryptography is handled in a real world project.

In terms of coding getting to know the inner workings of the Android runtime will certainly pay off in the future not only when it comes to Android related projects but in general with everything tied to the Java Virtual Machine.

VIII. Bonus section: qrgo

Although not a part of the initial project specification the QR-code encoder turned out to be a valuable tool for the entire application. It not only simplified testing but also yields a decent user experience when it comes to joining a Skipchain.

The decision to actually code an encoder from scratch in Go fell pretty early in the project initially as a little side project to learn enough of the language to be able to write the JSON-HTTP interceptor for the Cothority. As the state of the encoder progressed it became more and more clear that the available QR-code specification could be translated into Go code almost without any complicated modifications with the exception of the Reed-Solomon which is handled by an external library.

In the end it was an intriguing experience when the first device could join an existing Cothority solely via scanning a simple QR-code that has been printed onto a terminal screen.

The source code for the QR-encoder can be found here:

github.com/qantik/qrgo

IX. Installation guide

Everyone gritty enough to give the CISC Android application a try is required to have access to a running Cothority network, which sports a specific branch containing the JSON interface. It can be found under the following link. This includes the installation of the Go programming language because the project needs to be built from scratch.

github.com/dedis/cothority/tree/android_client_586
golang.org/doc/install

To start a simple Cothority with three nodes locally one can execute the shell script in the inside the project directory.

app/cisc/setup_cothority.sh

The actual Android application is available either in source code form or as a packaged APK on its online repository.

github.com/qantik/CISC

Apart from SDK 10 (Gingerbread) and the permission to use the camera nothing else is required to run the application. Naturally, the entire code base is open source and contributions or positive stimuli are more than welcomed.

References

[1] D. Bernstein. *Multi-user Schnorr security, revisited*, 2015. ed25519.cr.yp.to/multischnorr-20151012.pdf.

[2] DEDIS. *CoSi White Paper*, 2015. arxiv.org/abs/1503.08768.

[3] DEDIS. *DEDIS Crypto Repository*, 2016. github.com/dedis/crypto.

[4] DEDIS. *DEDIS Protobuf Repository*, 2016. github.com/dedis/protobuf.

[5] DEDIS. *Web Socket Pull Request Cothority*, 2016. github.com/dedis/cothority/pull/668.

[6] Denso. *QR-code Specification*, 2000. www.qrcode.com/en/about/standards.html.

[7] D. B. et.al. *High-speed high-security signatures*, 2011. ed25519.cr.yp.to/ed25519-20110926.pdf.

[8] Git-SCM. *Hash Descriptions in Git*, 2016. git-scm.com/book/en/v2/Git-Tools-Revision-Selection.

[9] Google. *Introduction to Go Structs*, 2015. golang.org/doc/effective_go.html.

[10] Google. *Android Developer Hub*, 2016. developer.android.com/studio/index.html.

[11] Google. *Android Background Services Documentation*, 2016. developer.android.com/guide/components/services.html.

[12] Google. *Android Async Task Documentation*, 2016. developer.android.com/reference/android/os/AsyncTask.html.

[13] Google. *Espresso Android GUI Testing*, 2016. google.github.io/android-testing-support-library/docs/espresso/.

[14] Google. *Go TLS Library*, 2016. golang.org/pkg/crypto/tls/.

[15] Google. *GSON User Guide*, 2016. github.com/google/gson/blob/master/UserGuide.md.

[16] Google. *Android Shared Preferences Documentation*, 2016. developer.android.com/training/basics/data-storage/shared-preferences.html.

[17] D. R. Hipp. *sqlite Main Page*, 2000. sqlite.org.

[18] Meteor. *Meteor.js Introductory Page*, 2017. meteor.com.

[19] Oracle. *Java SSL Sockets Library*, 2016. docs.oracle.com/javase/7/docs/api/javax/net/ssl/SSLSocket.html.

[20] qantik. *qrgo Repository*, 2016. github.com/qantik/qrgo.

[21] str4d. *Edwards Curve Implementation Java*, 2016. github.com/str4d/ed25519-java.

[22] Various. *Wikipedia Schnor r Signature Entry*, 2016. en.wikipedia.org/wiki/Schnorr_signature.

[23] W3. *HTTP1.1 Response Codes Specification*, 1999. www.w3.org/Protocols/rfc2616/rfc2616-sec10.html.

[24] S. Wambler. *Introduction to Test Driven Development (TDD)*, 2013. http://agiledata.org/essays/tdd.html.

[25] P. Wuille. *Schnorr Signature in Bitcoin*, 2016. diyhpl.us/wiki/transcripts/scalingbitcoin/milan/schnorr-signatures/.