Cross-Platform Mobile Application for the Cothority

Vincent Petri & Cedric Maire

School of Computer and Communication Sciences

Decentralized and Distributed Systems Lab

Semester Project

January 2018

Responsible
Prof. Bryan Ford
EPFL / DeDiS

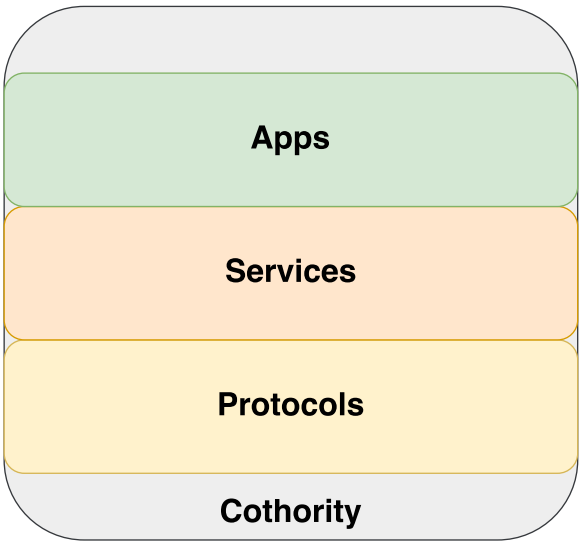
Supervisor
Linus Gasser
EPFL / DeDiS

Acknowledgements

We would like to express our special thanks to Linus Gasser who gave us the opportunity to do this very interesting project related to the collective authority (Cothority) framework developed by the DeDiS laboratory at EPFL. We would like to thank him for the valuable help he gave us whenever we needed and for the knowledge he provided to us throughout the semester. Secondly, we would also like to thank our parents and friends who helped us through the hard times of finalising the project within the limited time frame.

Abstract[1]

The Cothority[2] framework has been developed and maintained by the DeDiS laboratory at EPFL. This project provides a framework for developing, analysing, and deploying decentralised and distributed cryptographic protocols. A set of servers that runs these protocols and communicates among each other is referred to as a collective authority, or cothority, and the individual servers are called cothority servers or conodes. A cothority that executes decentralised protocols could be used for collective signing, threshold signing, or the generation of public-randomness, to name only a few options. The highest level of abstraction can be created by protocols like the collective signature (CoSi) protocol, the random numbers (RandHound) protocol, or the communication (Messaging) protocol used by the conodes to exchange data. Then come the services, which rely on these protocols. As of this writing, there exist several services: the Status service to enquire into the status of a conode, the CoSi service for collective signing, the Guard service that allows for the distributed encryption and decryption of passwords, the SkipChain service for storing arbitrary data on a permissioned blockchain, and the Identity service for distributed key/value pair storage. Applications (also called "apps") run on top of these services, including Status, CoSi, Guard, collective identity skipchains (CISC), and proof-of-personhood (PoP). In this project report, we only concentrate on the last two, CISC and PoP.



Cothority Framework

[1] https://github.com/dedis/cothority/wiki
[2] https://github.com/dedis/cothority

CISC App

// TODO: INSERT

PoP App[3]

Anonymity on the internet is often a trade-off with accountability. Users want to be as anonymous as possible without losing rights and opportunities. This desire is in contrast with the needs of many online service providers who require this accountability to be able to provide users with a secure and high-quality experience. Captcha is one of the most frequently used methods to block non-human beings from accessing information. However, on one side, programs have become better and better at solving the Captcha queries, and on the other side, even human beings are occasionally unable to correctly decode a Captcha. The PoP app tries to remedy to this problem by providing so-called PoP Tokens, which can be considered to be a one-time captcha. These tokens are comparable to completely anonymous ID cards. The PoP Token proves that its holder is a human being who visited a specified place at a specified time, though it does so without revealing to which specific person the token refers.

[3] `https://github.com/dedis/cothority/wiki/PoP`

Contents

Chapter 1

Introduction

As CISC and PoP apps are completely implemented and functional only in the backend, it is difficult for casual users to use these technologies without investing ample time in the hands-on process of creating and running one or multiple conodes. The main purpose of this semester project is to lift this restriction and bring these technologies to the availability of the casual and non-technical user. In the past, in order to store data in an identity skipchain or hold a PoP party, it was necessary to use the command line interface (CLI), which is quite impractical and difficult to use. Providing the end user with an easy way to access the Cothority framework functionalities is a crucial part in the process of bringing this new technology into wide public use.

Almost everyone nowadays owns a smartphone, be it an Android device or an iPhone, most people have one in their pocket. The idea behind this project is to create a cross-platform mobile application for the Cothority (CPMAC), and thus make the functionalities as user friendly as possible so that this technology is accessible to almost everyone. Starting with a simple proof of concept (PoC) for CISC and PoP as a mobile application, the final aim is to build this app such that further technologies in the Cothority framework are easily implementable and extensible. The JavaScript (JS) language has been chosen for this purpose, not only due to its popularity and simplicity, but also because it allows us the ability to write the entire application logic in a single language and compile it to run on both desired platforms, Android and iOS. With only few tweaks and changes (due to libraries only available to NativeScript), the core logic of CPMAC could even be used to run web apps or desktop applications, as there are many frameworks that enable users to compile for these systems by writing the logic in JS. The framework we chose is called NativeScript[1]. The reasons behind this choice are simple. First, NativeScript makes it possible to have a real

[1] `https://www.nativescript.org`

native app on Android and iOS without it running in so-called web views. Secondly, since the user interface (UI) is described in the XML format, it is cross-compatible i.e., the app does not have to be redesigned for each platform. Lastly, NativeScript is highly extensible with the use of NPM[2] packages or even native Gradle[3] and CocoaPods[4] libraries for Android and iOS, respectively. This last reason provides zero-day support for native APIs.

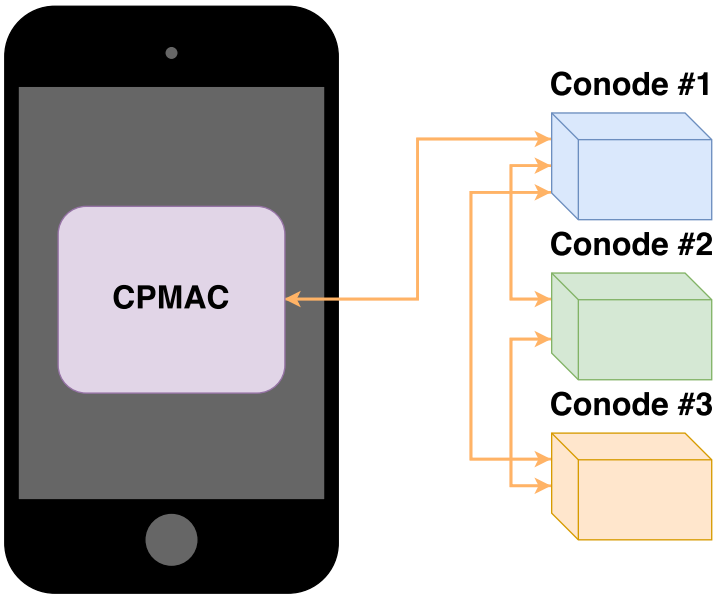


Figure 1.1: Inter-conode and CPMAC-conode communication.

[2] https://www.npmjs.com
[3] https://gradle.org
[4] https://cocoapods.org

Chapter 2

Background

2.1 Cothority

The Cothority framework is composed of multiple protocols, services, and apps. At its current stage of development, CPMAC only supports the CISC and PoP apps, but it is intended to progressively feature and integrate more apps. We now present these two apps in more depth.

2.1.1 CISC

// TODO: INSERT

2.1.2 PoP

The PoP app of the cothority framework is used to generate and verify proof of personhood, which indicates that a user is a human being. Personhood is proven through stating that a specific person was at a precise location at a particular time and thus that this user is not a bot or any other kind of human-simulating program. Before continuing, we define some important terms:

Key Pair
: In cryptography a key pair is composed of a public and private key, the public key is shared with anyone as opposed to the private key which has to remain secret to the owner of the key pair. Typically a message is encrypted and/or signed with the private key and can then be decrypted and/or verified with the public key.

Conode Linking
: Before being able to exchange data with a cothority node, one has to link itself to it, this is done by registering the public key on the conode after having shown that one has access to it (typically by reading a short PIN in the server logs).

PoP Party
A PoP party is a gathering of people wanting to prove that they are human beings by showing everyone that they are able to come to a specific location at a specific time.

Organizer (Org)
An organizer is someone hosting a PoP party by providing a conode. Since there are multiple organizers, their conodes form a cothority network and thus host the party in a distributed and decentralized manner.

Attendee (Att)
An attendee is someone present at a PoP party without providing a conode and thus only attending it for the sake of a PoP token (an organizer can at the same time be an attendee).

PoP Party Configuration/Description (Config/Desc)
The configuration or description of a PoP party defines all the required properties. It should include the name, the date and time, the location and a list of all the hosting conodes (which is commonly refered to as a roster) for the PoP party.

Attendee Registration
The attendee registration is the process of registering all the public keys of the attendees on each of the organizers conodes, all the organizers have to do it separately for their own server.

Final Statement
A final statement of a PoP party is generated by the cothority network composed of the hosting conodes, it is composed of a PoP party configuration, all the public keys of the attendees, the collective signature generated by the hosting conodes and a boolean to state if this PoP party has been merged with another one.

PoP Token
A PoP token is the final token proving that the holder is a human being and attended this particular PoP party, it is composed of a final statement and a key pair.

First, all of the party organisers must agree on a date, time, and location. Once these specifications are set, all of the attendees (including the organisers) meet at the right location, date, and time. Each organiser has to link with a conode, complete the PoP configuration, and register it on his or her own conode; the organisers then receive the identification of this PoP party, which is henceforth used as a reference to uniquely identify it. Once every organiser registers the configuration and the PoP party is over, each

one registers the list of all of the public keys of the attendees. During this step, the organisers have to ensure that each attendee has registered one and only one public key; this is a crucial step because otherwise an attendee could generate as PoP tokens for every public key that he or she registered. This would then contradict the important criterion that each human being is unique and thus should only receive one PoP token for each attended party. Once every organiser registered all of the attendees, the cothority composed of their conodes finalises the party by generating the final statement containing all relevant information, which includes a collective signature. This final statement is then sent to the attendees so that they can generate their PoP token by linking this final statement to their key pair.

2.2 Technologies

Throughout this project many different technologies were used, we are now quickly presenting the main ones in a few words.

2.2.1 Elliptic Curve Cryptography[1]

This public-key cryptographic system using elliptic curves (EC) has been independently suggested by Neal Koblitz and Victor S. Miller in 1985. EC algorithms only entered wide use in 2004 to 2005. The major difference with prior cryptographic protocols (for example DSA or RSA), that were defined over multiplicative groups of finite fields, is that EC cryptography uses point addition instead of modular exponentiation. This results in faster computation.

2.2.2 Schnorr Signature[2]

The schnorr signature algorithm is considered as one of the simplest scheme being provably secure in a random oracle model to produce digital signatures. In addition to that, the schnorr algorithm is efficient and produces short signatures.

2.2.3 Protocol Buffers[3]

Developed by Google, this data interchange format is a language- and platform-neutral structured data serializer. Protobuf allows to define data structures once and then easily write to or read from data streams. In this particular project it is used to encode JS objects and decode byte streams received from the conodes and thus ease the process of data exchange.

[1] `https://en.wikipedia.org/wiki/Elliptic-curve_cryptography`
[2] `https://en.wikipedia.org/wiki/Schnorr_signature`
[3] `https://en.wikipedia.org/wiki/Protocol_Buffers`

2.2.4 Websocket[4]

This communication protocol uses a single TCP connection to send data back and forth while keeping the connection open, thus facilitates real-time data transfers from and to the server. It is the used protocol for data exchange with the conodes.

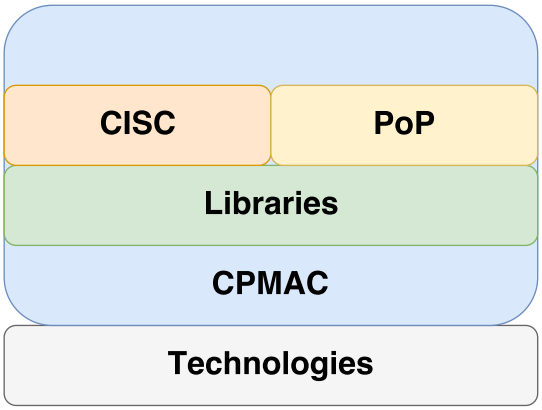


Figure 2.1: CPMAC Structure

[4] https://en.wikipedia.org/wiki/WebSocket

Chapter 3

Design

In this chapter, we begin by presenting the main structure of the app, and then we further detail the different objects and libraries that have been implemented throughout the project.

The app itself uses a side drawer component to navigate between the main parts, such as the home screen, CISC, PoP, and settings. Each these drawers then splits in multiple tabs if needed in order to separate sub-sections of each component. In each tab, only the functionalities available to this sub-section of the component can be found. The following list shows the main structure of the app, such that each main point is a drawer and each sub-point is a tab.

- Home
  The home screen is used to display the conodes of the app user, he can add and remove conodes, fetch their statuses and display them as a QR code.
- PoP
  - PoP
    Used to display data shared between attendees and organizers, it shows the list of all the fetched final statements and generated PoP token.
  - Org
    Contains all the functionalities needed by the organizers of a PoP party.
- CISC // TODO: INSERT
- Settings

  - User
    The settings of the user include key pair generation and displaying, you can also completeley reset all the data linked to the user of the app itself.
  - CISC // TODO: INSERT
  - PoP
    In this tab you can reset the global PoP data and the data linked to the organizer.

In order to easily represent the information, manage the data, and use the corresponding functionalities for all of these components, we chose to create multiple JS classes that are true singletons (as far as JS allows). Singletons permitted us to always work on the exact same object independently from the location we require for the class; since we had to store data locally, we had to load the saved states into memory for each of these classes. Because of the singleton design pattern, this loading must only be executed once, either at the start-up of the app or the first time the class is required, and all subsequent calls use this pre-loaded object. On the other hand, since we work with singletons, we were not able to create relationships between them; for example, the class that represents the organiser of a PoP party does not extend the main PoP class (each subclass instance would recreate their own parent class instance). The classes that have been implemented are listed below:

- User.js
  Contains all the data and logic that is global and belongs to the user of the app. At this state the user class manages the key pair and the roster displayed in the home screen.
- Cisc.js // TODO: INSERT
- PoP.js
  Contains everything that is common to organizers and attendees, it manages the list of final statements and PoP tokens belonging to the user.
  - Att.js
    This class is only a skeleton for now and not used at all. It is a placeholder for future implementations that are specific to the attendees of a PoP party.
  - Org.js
    Representing the organizer of a PoP party. This class manages the current linked conode, PoP configuration, registered attendees and ID of the PoP description.

In addition to these objects, we wrote some libraries so that we can easily manipulate any kind of data, be it local or exchanged with conodes. The main libraries are presented as follows:

- Convert.js
  Library for converting data types but also to parse all kind of stored data.

- Crypto.js
  Everything related to cryptography which includes but is not limited to key pair generation, message signing and verification or EC points aggregation.

- Helper.js
  All kind of helper functions that may be needed in several different places but do not belong to any other library.

- Net.js
  Contains all the methods related to communication over the internet, may it be over websockets or HTTP requests.

- protobuf/
  Creation, encoding, decoding and more using protocol buffers to efficiently use the same object structure as the CISC and PoP implementations in Go[1] from the DeDiS lab.

The classes and libraries in combination provide any required logic to execute the CISC and PoP apps locally but also to perform any communication needed with the conodes. We chose to implement these in such a way that it not only can be taken out as a whole and used as a library with as few tweaks and changes as possible for any other JS project related to the Cothority framework, but also so that it is easily extensible for other Cothority apps that will one day be implemented. We decided to call this aggregation of libraries "DeDjS". Throughout the entire project, any data like keys, IDs, or EC points were handled as unsigned byte arrays (Uint8Array in JS) but shown to the user in the base64[2] format (unless explicitly stated otherwise). This choice combines an easy way of handling the data and an easy way for the user to read the data on the screen. Moreover, since JS is an un-typed language, we decided to implement DeDjS so that it is as type safe as possible. This choice of implementation was made to prevent simple errors and return meaningful exception messages when a required type is incorrect.

[1] `https://golang.org`
[2] `https://en.wikipedia.org/wiki/Base64`

Much of the logic in DeDjS requires writing and reading data to and from the disk or sending messages over the internet in order to use the functionalities of Cothority. These tasks are slow, generate delays, and even have a relatively high probability of failing. One of the challenges was to find a way to execute all of these tasks in an asynchronous way so as to not block the main thread of the app and handle exceptions in a simple and clean manner. The solution to this problem was that, since ECMAScript[3] 2015, it is possible to use promises[4] in JS. Thus, we decided to implement DeDjS so that it uses promises anytime an asynchronous task is needed. In this way, the main thread of the app is never blocked, and the exceptions can easily be handled by the simple syntax provided by JS.

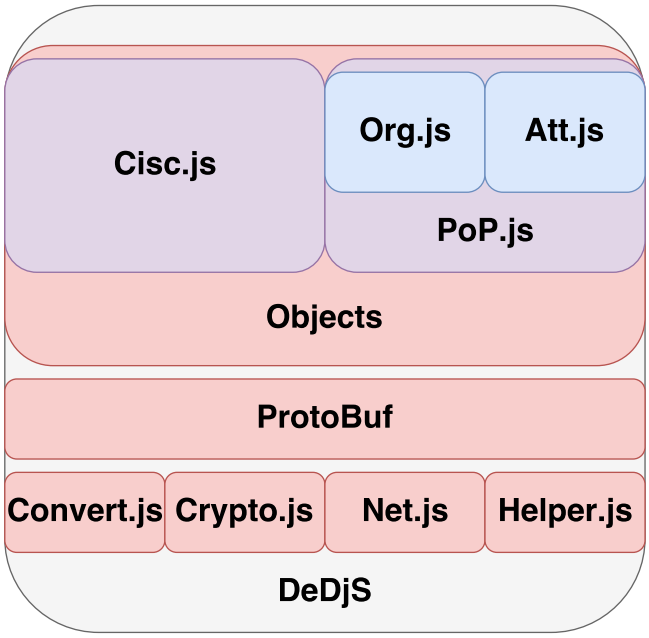


Figure 3.1: DeDjS Library Structure

[3] https://en.wikipedia.org/wiki/ECMAScript
[4] https://en.wikipedia.org/wiki/Futures_and_promises

Chapter 4

App User

The user of the app is represented by a single class, the User.js object. This singleton handles all that is global to the app or the user and does not belong to CISC or PoP in particular, such as the roster displayed in the home drawer and the key pair managed in the user tab of the settings drawer. After the loading screen of the app, users land in the home drawer. Here, there are two main components: the roster of the user and a button to fetch the status of all of the user's conodes. To add a new server, one can either enter the information manually or scan a QR code with either a JSON[1] or TOML[2] description of the node. After being added to the roster, the status of all of the servers in the roster is fetched and can be re-fetched at any time with the corresponding button. To see the status of a conode, simply clicking on one of the conodes in the list opens a new page that contains all of the fetched stats of the conode. On top of this page is a button that permits the user to display a QR code to represent that particular conode. This permits convenient sharing of conodes, including when setting up a PoP party description.

The next functionalities of the user are located in the user tab in the settings drawer. Here, the user has the ability to generate a new key pair at any time, which is then displayed in this tab. Moreover, the user can display the key pair as QR code (with the private key removed). For example, this could be used for easy registration in a PoP party. The last button in the users' settings completely resets any data related to the user, including the roster displayed in the home drawer and the key pair displayed in the settings.

[1] `https://www.json.org`
[2] `https://github.com/toml-lang/toml`

Chapter 5

CISC

// TODO: INSERT

Chapter 6

PoP

As already stated in the introduction, the backend for the PoP app is completely implemented in the Cothority framework, but it requires technical knowledge because one has to use the CLI to handle the functionalities. The first and main goal was to provide a user-friendly interface that is packaged in a cross-platform app for Android and iOS and allows anyone to use the technology provided by PoP. In this section, we describe the implementation of PoP in CPMAC and the process of creating, handling, and finalising a PoP party including the final PoP token creation.

6.1 Implementation and Evaluation

Three main classes form the PoP component: PoP.js, Org.js, and Att.js. The PoP.js object handles all information that is common or shared by the organiser and attendee status of a PoP party, for example the final statements and the PoP token the attendee owns. These are displayed in the PoP tab in the PoP drawer of the app. A final statement can be added by simply scanning a QR code provided by one of the organisers, deleting a final statement, generating a PoP token by linking it with the user's key pair, and revoking PoP tokens to return to the final statement. When revoking a PoP token, the final statement is then restored, so the PoP token could now either be generated again by linking it with a key pair or the final statement can be deleted. This has been implemented so that the user can change the key pair linked to the PoP token if a mistake has been made. The next class is Att.js, which, as mentioned previously, is only a skeleton for the class. At this point of development, this class is empty and acts only as a placeholder for future implementations that are specific to the attendee of a PoP party.

The last class is Org.js. This object is the most complicated one of the three as it has to handle every functionality that an organiser needs to create, handle, and finalise for a PoP party. The first step for an organiser is to link

with a conode. This can be achieved by clicking on the corresponding button in the org tab in the PoP drawer. The organiser can then choose with which conode to link. The suggested conodes are the same ones registered in the home page. By clicking on the wanted server, the organiser is asked to enter a PIN (found in the server logs). If the correct PIN has been entered, the linked conodes name, description, public key, and ID are displayed in the org tab. This way, the conode to which an organiser last linked is always clear (but not a guarantee that the conode is still linked to the organiser). The displayed conode is the one that is contacted anytime a message needs to be sent. The next step is to fill in the configuration of the PoP party. This can be done by clicking on the corresponding button, which opens a new window. Then, it is necessary simply to enter the name, date, time, and location of the party. The last step of the configuration is to provide the roster so that the part can be hosted. Either the organiser can enter each conode manually or the QR codes of the other organisers can simply be scanned, which is the more convenient way to do it. Once the organiser has filled in the configuration, it can be easily shared with other organisers by displaying it as a QR code.

The process of sharing a PoP description could not be achieved using only a QR code, since the amount of data a code can contain is heavily restricted[1]. Because of this restriction, we had to find a work around that permits organisers to share their configuration with other organisers. The current solution we propose is a third-party service called PasteBin[2]. This means that when someone wants to share a configuration and displays the QR code, it is first uploaded to the PasteBin service. The QR code then displays the ID of the paste, and the other person , by scanning this code, then downloads the description from PasteBin. This solution is not ideal and should only be temporary until a better one can be implemented (see future works).

The other organisers could fill in the configuration too, but since they must be exact copies (the only exception is the order of the conodes in the roster), the description-sharing idea is highly recommended. Once all of the organisers have entered or imported the PoP party configuration, they have to save it on their respective conodes, which then returns an ID that should be identical for every organiser. After registering, the ID of the current PoP description is displayed in the org tab just under the currently linked conode and is now used for all subsequent actions. The next step is to register all of the attendees of the party. By clicking on the corresponding button in the org tab, a new window is displayed. On this page, the organiser can collect

[1] `https://en.wikipedia.org/wiki/QR_code#Storage`
[2] `https://pastebin.com`

all of the public keys that should get registered. As always, either the public keys are entered by hand or it is possible to simply scan the QR code of the key pair. The order of the attendees is not important; however, organisers have to all register the same list of public keys as those that are not common to every conode are stripped out. Once all public keys are registered locally, they can be sent to the conodes by clicking on the register button (this finalises the PoP party on the respective conode). During this process, all organisers but the last one receives an error message that states that not all other conodes are finalised. Once the last organiser registers the attendees, everyone who received the error message can then return to the org tab and fetch the final statement by clicking on the corresponding button (the last one to finalise the PoP party automatically fetched it). By switching to the PoP tab, it should be possible to see the final statement. Organisers can now share this final statement with the attendees, and everyone can generate their PoP token by clicking on the final statement.

The process of linking to a conode and registering the PoP configuration (i.e., generating the ID of the party) are two crucial steps. During the linking process, the conode stores the public key of the organiser and then only accepts either messages that do not require a signature or signed messages that can be verified using this stored public key. The first signed message sent by the organiser is to register the PoP description. The ID is computed by hashing (in this paragraph, we always talk about SHA-256[3]) this description and the required signature along with the message in the ID signed using the Schnorr algorithm. The hash is computed by concatenating the strings of the name, date, time, and location and then appending the "aggregate" (point addition) of all of the public keys of the conodes that will host the PoP party. This forces the organisers to register exact copies of the party configuration (excluding the order of the conodes, thanks to commutativity). The second and last signed message sent by the organiser is the finalising request, which includes all registered attendees. The required signatures are the hash of the party ID concatenated with all of the public keys of the attendees. If either of the hashes are computed differently or signed using a private key that does not correspond to the public key stored on the conode, these messages are rejected.

6.2 Results

As of now, CPMAC is in its first development phase. The core libraries have been implemented, and currently only basic functionalities of CISC and PoP have been ported from the Cothority framework. For the PoP part, CPMAC enables anyone who wants to host or attend a PoP party to

[3] `https://en.wikipedia.org/wiki/SHA-2`

create, manage, attend, and finalise one. Moreover, it is possible to generate PoP tokens.

All of these functionalities have been implemented to be as user friendly as possible and should be more accessible to the public than the CLI that Cothority provided until now.

6.3 Future Work

As mentioned in the results section, only basic functionalities have been ported for now. However, all of the libraries and objects have been designed with the consideration that CPMAC will be extended by either providing CISC and PoP new functionalities or by even adding complete new Cothority apps like CoSi or Guard. Some possible future works are discussed below.

Replacing PasteBin
The PasteBin service is currently used to share PoP configurations because they include too much data to be contained in a single QR code. This method relies on a third party, and it also exposes data (not sensitive, but still an undesirable situation) on the internet. In addition, it limits the number shares, as PasteBin only allows a certain number of pastes per 24 hours depending on the status of the user who created it (guest, member, pro). A solution that would lift all of these restrictions would be to implement a new fetch functionality directly into the PoP app of the Cothority framework. It is already possible to fetch the final statement from a conode with the knowledge of its ID, and thus the same procedure could be used to fetch the PoP description that corresponds to a certain ID. The new procedure would then require a first organiser to fill in the description, register it on that conode, and provide the others with the ID, and the remaining organisers would simply fetch the description from the first organiser's conode and register it on their own.

PoP Party Merging
It is possible for PoP parties to be merged. This means that people can organise multiple parties around the world and merge the final statements so that they are all considered as one single PoP party. This is useful if the generated PoP tokens should have the same proof value but people have to meet at different places.

Viral PoP Parties
Once a PoP party is finalised, the final statement is registered on the hosting conodes. Moreover, all attendees listed in this statement are trusted people as they own a related PoP token. Attendees can then

host a new PoP party on one of the hosting conodes using their PoP token as authentication (instead of linking to it by providing a PIN). This could ease the process of hosting PoP parties as an attendee without having to set up a conode.

Sign and Verify Services

One of the main purposes of a PoP token is to be able to sign and verify different services. The token allows people to prove that they were at a certain location at a certain date and time, and thus it should provide some rights that were linked to the PoP party. As an example, we use the BeerToken suggested by the DeDiS laboratory. To begin, DeDiS would organise a PoP party and invite all of its members. The goal of the PoP party is to hand out PoP tokens called BeerTokens. A BeerToken would guarantee every attendee a free weekly beer at Satellite[4], the bar of EPFL. In order to make this possible, it would be required to either verify a BeerToken in order to know if the user has already ordered the one free beer of the week, or sign using a BeerToken to claim the weekly beer. All of this could be implemented in CPMAC by extending the core libraries and objects.

[4] `https://satellite.bar`

Chapter 7

Installation and Running of CPMAC

We will now see how to install all required dependencies and how to compile, test and run the app. The following steps are:

1. Installation of Go Language

   To be able to run the cothority framework you'll need the go compiler. Install it by following the official installation guide: `https://golang.org/doc/install` . You also need to set your `GOPATH` environment variable by either following the official guide[1] or by running[2]:

   ```
   $ echo 'export PATH=$PATH:$(go env GOPATH)/bin'
   >> ~/.bash_profile
   ```

2. Cothority Installation and Running of Conodes[3]

   First you'll need the correct version of Cothority and run the conodes to be able to interact with them as you use CPMAC. CPMAC is developped to run against the stable version 1.2 of Cothority. As stated in the `README.md` file on the GitHub page of the cothority framework, you have to use the version installed in `gopkg.in/dedis/cothority.v1`. The source code in this folder corresponds to the branch v1.2 located here: `https://github.com/dedis/cothority/tree/v1.2`. It is crucial that you run CPMAC against this stable version, as Cothority is in heavy developpement many functionalities have already been changed to work differently and the implementation will not be compatible. Clone the repository by running:

   ```
   $ go get -u github.com/dedis/cothority
   ```

[1] `https://golang.org/doc/code.html#GOPATH`
[2] Terminal restart needed.
[3] `https://github.com/dedis/cothority`

Locate and enter the folder `gopkg.in/dedis/cothority.v1` in your `GOPATH`, you can then execute the following command to run three local conodes:

```
$ ./conode/run_conode.sh local 3 5
```

3. Official NativeScript Tutorial[4]

   We recommend to follow the instructions on their official page, as it will always be up-to-date. But here are the main steps:

   (a) NodeJS Installation[5]
   This can be done by downloading the installer on their home page. Always install a long term service (LTS) version as it is the supported version for NativeScript. If you install it using the macOS package manager brew[6], don't forget to manually add the NodeJS path to your bash profile by running[7]:

   ```
   $ echo 'export PATH="/usr/local/opt/node@
   {VERSION}/bin:$PATH"' >> ~/.bash_profile
   ```

   (b) NativeScript CLI Installation[8]
   This can simply be done by running this command:

   ```
   $ npm install -g nativescript
   ```

   If you get EACCES errors, try to run the last command again with administrator rights. If you still get EACCES errors, run the command again with administrator rights and the unsafe permissions parameter of NPM:

   ```
   $ sudo npm install -g --unsafe-perm nativescript
   ```

   Try running the command tns, if the command return no error you may continue.

   (c) Android and iOS Requirements
   For this part of the installation process we redirect you to the official tutorial[9] since it is more complex and depends on your operating system (OS). NativeScript provides scripts for Windows and macOS that will automatically setup most dependencies. We still recommend having a look at the advanced setups they provide to ensure that everything is correctly installed.

[4] `https://docs.nativescript.org/start/quick-setup`

[5] `https://nodejs.org/en/`

[6] `https://brew.sh`

[7] Replace `{VERSION}` by the installed version through brew. A terminal restart is required after executing the command.

[8] `https://www.npmjs.com/package/nativescript`

[9] `https://docs.nativescript.org/start/quick-setup#step-3-install-ios-and-android-requirements`

(d) TNS Doctor

The last step is to check if all requirements are met, this can be done by running: `tns doctor`. If errors are returned, fix them before continuing.

4. Editor

   This step can be skipped if you don't want to contribute to the project. In essence, you could use any editor you'd like. We recommend using Visual Studio Code[10] since this is the officially supported editor and provides an official plugin[11] to integrate with NativeScript.

5. Install, Compile and Run CPMAC

   Before going further make sure you have your conodes and an Android emulator or iOS simulator set up and running(if you want to test or run CPMAC on you smartphone, make sure it is well connected and recognised by your system). First you need to clone the repository by executing:

   ```
   $ git clone https://github.com/dedis/
   student_17_mobile.git
   ```

   After entering the newly created folder `student_17_mobile`, you are able to test and run CPMAC by executing either one of the following commands:

   ```
   $ make clean-test-<platform>
   $ make clean-run-<platform>
   ```

   Where `<platform>` has to be replaced by either `android` or `ios`. This will install and compile all the needed libraries and test or run CPMAC. All subsequent tests and runs can be made by running:

   ```
   $ make test-<platform>
   $ make run-<platform>
   ```

[10] `https://code.visualstudio.com`

[11] `https://marketplace.visualstudio.com/items?itemName=Telerik.nativescript`

Chapter 8

Known Bugs

As it is relatively new, NativeScript still exhibits some unexpected behaviours for which we had to find solutions. We now discuss some known bugs that have been bypassed or are still problematic.

Compressed Files (Android)
Some NPM libraries are shipped with compressed files that usually end with `.gz` and are also included in their non-compressed form[1]. Android interprets these as a same and single file, this means that at compilation time Gradle will throw a `duplicate resources` error. To bypass this bug, we automatically delete the compressed versions[2] before they get compiled. You can find the corresponding commands in the `prepare-hook.sh` bash script.

The BroRand Library[3]
BroRand is a JS library to generate random numbers. CPMAC makes indirectly use of BroRand through the EC library called elliptic[4]. BroRand is not fully supported on the NativeScript framework and throws an error at execution time when trying to generate a random number. We bypassed this by replacing the problematic lines in such a way that it works with NativeScript. You can find the corresponding command in the `prepare-hook.sh` bash script and the modified code in the `brorand-fix/` folder.

WebSocket Bug (iOS)
The websocket library for NativeScript does not work correctly when run on iOS. Currently we didn't find a way to correct this bug in any way. The used library is called `nativescript-websockets`[5].

[1] For example `file.min.js` and `file.min.js.gz`
[2] Not needed at runtime.
[3] `https://www.npmjs.com/package/brorand`
[4] `https://www.npmjs.com/package/elliptic`
[5] `https://www.npmjs.com/package/nativescript-websockets`

Basically, this library wraps native websockets libraries for Android and iOS in JS objects. It seems that the wrapper implemented by `nativescript-websockets` is working properly, but that either the underlying native Objective-C library, which is a slightly modified version of PocketSocket[6], or iOS itself is causing troubles. It seems that either PocketSocket or iOS is resending previous messages by concatenating them with new data. Below are two sent messages and the corresponding received messages by a conode, they are displayes in byte and base64 format to be easily readable. We verified that the sent messages are properly passed down to the native library by the JS wrapper. The first message is an empty pin request, thus only contains the public key of the organizer and is correctly received by the conode. The second message, which now also contains the PIN printed by the conode is not received correctly. The received message is the first message concatenated by six zeros and then the beginning of the second message.

Sent

EiDLgkwNauV7pExX7QEpT3Zdu7z4nxWRnmRXdK9KvZnEfA==
18 32 203 130 76 13 106 229 123 164 76 87 237 1
41 79 118 93 187 188 248 159 21 145 158 100 87
116 175 74 189 153 196 124

Received

EiDLgkwNauV7pExX7QEpT3Zdu7z4nxWRnmRXdK9KvZnEfA==
18 32 203 130 76 13 106 229 123 164 76 87 237 1
41 79 118 93 187 188 248 159 21 145 158 100 87
116 175 74 189 153 196 124

Sent

CgY3NTIzMTMSIMuCTA1q5XukTFftASlPdl27vPifFZGeZFd0r0q9mcR8
10 6 55 53 50 51 49 51 18 32 203 130 76 13 106
229 123 164 76 87 237 1 41 79 118 93 187 188
248 159 21 145 158 100 87 116 175 74 189 153
196 124

Received

EiDLgkwNauV7pExX7QEpT3Zdu7z4nxWRnmRXdK9KvZnEfAAAAAAAAAoG
18 32 203 130 76 13 106 229 123 164 76 87 237 1
41 79 118 93 187 188 248 159 21 145 158 100 87
116 175 74 189 153 196 124 0 0 0 0 0 0 10 6

[6] `https://github.com/NathanaelA/PocketSocket`

As already stated before, we were not able to bypass or fix this bug. Here is a non-exhaustive list of what we already tried but didn't succeed:

- Find another NativeScript library to replace `nativescript-websockets`
- Modify `nativescript-websockets` to try to reset the message buffer and others
- Natively implement websockets by using the same PocketSocket version as `nativescript-websockets`
- Natively implement websockets by using a different native library called SwiftWebSocket[7]

[7] `https://github.com/tidwall/SwiftWebSocket`

List of Figures