Cothority Mobile: a mobile application to perform distributed tasks using the cothority-framework

Lucio Romerio

School of Computer and Communication Sciences

Decentralized and Distributed Systems lab

Bachelor Project

Spring 2017

Responsible
Prof. Bryan Ford
EPFL / DEDIS

Supervisor
Linus Gasser
EPFL / DEDIS

Summary

1 Introduction

The main goal of this project, as suggested by its title, is to implement a mobile application to perform distributed tasks with the cothority-framework. Actually, when I started working on this project, there already was an existing android application using the *cisc* service. Anyway there were two main reasons to restart from scratch and to implement a new app:

1. The already existing application used a json-interface to send messages, which has been now substituted by a websocket-interface.

2. Make your application available both for iOS and Android is obviously better than only providing an Android version.

My project was built starting from this two points. There are a couple of cross-platform frameworks which allows you to code the Android and the iOS application both at the same time. The first thing I did was to gather information about those frameworks; then, together with Mr. Gasser, we chose one of them: *PhoneGap*, which I will shortly describe in the next subsection. To use it you have to code in JavaScript, therefore we added another task to this project: to create two libraries. One to use the crypto primitives implemented by the DEDIS Go library in JavaScript, and another one to send/receive the cothority messages through the protobuf protocol in JavaScript.

In the next few pages I will give you a quick overview on *PhoneGap*, as well as on *cisc* – the service which I ended up implementing – and on the two libraries introduced above.

1.1 PhoneGap

PhoneGap [9] is a framework originally developed by *Nitobi*, which was purchased by *Adobe Systems* in 2011 [12]. It allows to build mobile applications using CSS3, HTML5 and JavaScript instead of the native languages (Java for Android and Swift for iOS). Thanks to *PhoneGap Build*[13], CSS, HTML and JavaScript code can be wrapped according to the device operating system. The result is an hybrid application that has access to most of the native APIs functionalities, but in which the layout is rendered via web views (instead of the native UI). *PhoneGap Build* offers three plans, including a free one. The main limitations of this latter are a maximum app size of 50MB and the prohibition to have more than one private application. Both of those limitations are not a problem for this project, thus we decided to use the free plan.

To build an application with *PhoneGap Build* is quite straightforward, you simply have to provide a link to your project repository and launch the build. There is an additional step to do in order to build the iOS version of

your application: to provide a valid certificate for the build. Obviously, if you want to release your application you will need a certificate also for the Android version. You can find information about how to create and register those certificates by looking at the *PhoneGap* tutorial 11.

1.2 Cisc

The official README [3] of the *cisc* service defines this latter as follows.

> *"Cisc uses a personal blockchain handled by the cothority. It can store key/value pairs, and has a special module for managing ssh-public-keys.*
>
> *Based upon skipchains, cisc serves a data-block with different entries that can be handled by a number of devices who propose changes and cryptographically vote to approve or deny those changes. Different data-types exist that will interpret the data-block and offer a service.*
>
> *Besides having devices that can vote on changes, simple followers can download the data-block and get cryptographically signed updates to that data-block to be sure of the authenticity of the new data-block".*

In the quote above, there may be a couple of terms worth an explanation. A blockchain is a form of distributed append-only log, often used in cryptocurrencies. As its name suggests, a blockchain is composed of blocks. Usually those blocks contain some application-specific data as well as the hash of the last block, a timestamp and a nonce. Skipchains combine blockchains with skiplists. Those authenticated data structures allow to traverse in a secure way the timeline in both forward and backward directions, and, thanks to multi-hop links, long distances can also be efficiently traversed. As in blockchains, the backward links are hashes of past blocks; while forward links are added retroactively and consist in cryptographic signatures of future blocks [7].

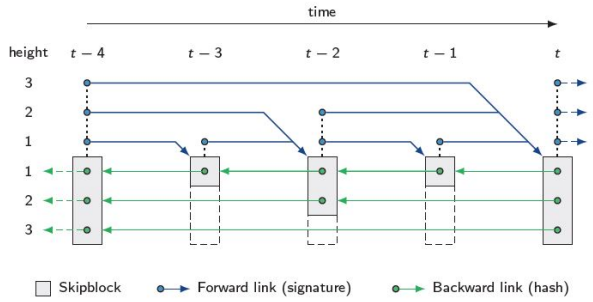

Figure 1: Skipchain structure [7]

On top of those structures, DEDIS has developed a collective authority (cothority) which basically consists on a set of servers that provides a series of services – like *cisc* – in a collectively and decentralized way. From now on those servers will be called "conodes".

In order to fully understand what explained in the next sections, a more detailed but still general overview of the *cisc* service is necessary. As you now know, *cisc* is a key-manager tool; its more relevant functionalities, for what concerns this project, are listed below.

1. Add your device to an access-control-list.
2. Proposing to add, remove or update a key/value pair to this access-control-list.
3. Define the threshold of the votes that a proposition has to reach in order to be accepted.
4. Vote an existing proposition.

Every conode is provided with a config file in which are stored the threshold, a collection containing all the devices on the skipchain and another collection containing the key/value pairs. Note that usually, the values stored on this latter are ssh-public-keys.

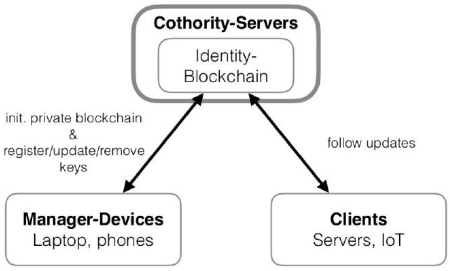

Figure 2: Cothority Architecture [6]

1.3 Libraries

The two libraries were already introduced and their goals should be clear enough; what I want to explain here is just how we decided to proceed for their implementation. During this semester there also was a master student doing a project in the DEDIS laboratory. He also needed those two libraries for his project, therefore we decided to implement them together. I have to say that he had a better JavaScript knowledge than me, thus he coordinated the implementation of those libraries. Nevertheless he gave me a lot of useful advices, not only for what concerns the libraries but also for JavaScript in general.

2 Architecture

In the next two sections I will try to describe all what bounded with the development of Cothority Mobile, the application I implemented. Doing

this, I will follow the logical order of the process which, from my point of view, also reflects the best approach to build this mobile application.

1. Extend the two libraries, Cothority Protobuf and CryptoJS, to support the needs of the *cisc* service.
2. Setup a *Phonegap* project.
3. Design and implement the database of the application, after analyzing its needs.
4. Implement all the *cisc* features one by one.

Unfortunately, for a couple of reasons, I was unable to always follow this logical order, but it still makes sense to follow it while describing my project.

This section, Architecture, includes the first two points, as well as the design of the database. Then, in the next section, I'll proceed explaining the implementation of Cothority Mobile itself more in detail.

2.1 Cothority Protobuf

As already outlined, DEDIS conodes are using the Protobuf protocol in order to communicate. In particular there are a series of messages used by the *cisc* service; the ones used by Cothority Mobile are the following: *ConfigUpdate*, *ConfigUpdateReply*, *ProposeSend*, *ProposeUpdate*, *ProposeUpdateReply* and *ProposeVote*. Some of those messages contain one or more complex fields, which are special structures like *Config*, *Device* or *SchnorrSign*.

I do not think explaining the above structures in detail would be useful, thus I will rather focus on explaining the general idea behind the implementation of one of them. Anyway you can find their definition by looking at the library source code [2], as well as at the DEDIS repository on github [4].

Cothority Protobuf has three "core" files: *root.js* takes care of wrapping and exporting all the messages defined inside the library, *index.js* defines a couple of helper methods that are then used in *cothority-protobuf.js* to define the library API, which basically consists in a series of functions to encode/decode the previously defined messages. Given the structure above, the implementation of a single message was the following.

1. Analyze and understand the structure of the original message.
2. Define additional structures, if any is needed.
3. Define the message itself.
4. Add to *cothority-protobuf.js* some methods to decode and/or encode the new message.

One final note on the Cothority Protobuf: during the last weeks of the semester its structure has been changed to use *.proto* files. I was not involved into this change, thus I am not going to explain this library structure further.

2.2 CryptoJS

This library uses GopherJS [5] in order to compile Go code to pure JavaScript code. The key idea is to take the cryptographic functions included in the DEDIS Go library, to adapt them to be compatible with JavaScript but still coding in Go, and finally to transpile them. This library is provided with some common primitives that allow – for example – to create an EdDSA key pair, to extract public keys and aggregate them, as well as computing sha256 and sha512 hashes. Beside those basic functions, there are some more specific ones that may be bounded to one of the services provided by DEDIS conodes.

I think an interesting thing here would be to give an overview of the *cisc* specific functions implementation, which are namely: *HashConfig* and *SchnorrSignature*. Both of those functions are hardly inspired to their respective ones in the original Go library, this is something that holds in general for the whole CryptoJS. As already outlined, the biggest challenge was bounded to compatibility. For example, considering *HashConfig*, the original function takes a *Config* – which represents the access-control-list – as an argument, while here all what we have is a *JavaScript Object*. As a consequence we need to get all fields from the object one by one and, at the same time, cast them to the correct type. Apart from that, the function is basically an exact copy of the existing one. The same holds for *SchnorrSignature*, which computes a Schnorr signature from a given key pair and a message: apart from the type of the arguments and from the return value, the function is identical to the original one.

```
"threshold": 2,
"device": {
        "icsil1-conode1": {
                "point": /*public key*/
        }
},
"data": {
        "ssh:icsil1-conode1:test1": /*ssh key*/
}
```

Figure 3: simplified JSON representation of a Config

All the functions defined in the library are then wrapped into a *JavaScript Object*: cryptoJS. Thus, once compiled with GopherJS and correctly imported into your project, the function of this library can be used with a syntax similar to *'cryptoJS.myCryptoFunction(args)'*.

2.3 Project Setup

There are two different ways to setup a *PhoneGap* project: through the *Desktop Application* or the *CLI*. I chose the second one and used *npm* to install the *CLI*, then the setup was quite straightforward, given the official tutorial [8]. What you need to know about the structure of the project is that it is almost identical to a simple website; in addition there is a

config.xml file which controls options related to the device operating system. The next step was to include the two libraries described above. Here I have to underline the fact that a pure *PhoneGap* project can import external libraries/JavaScript files only through the script tag[1].

After importing the two libraries, I created a simple sidebar to simplify the debug and to make all functionalities easily accessible to the user. In figure 4 you see the final version of the sidebar, which includes all the functionalities that we will soon analyze.

Another thing I did before starting the development of the mobile application itself, was to include the *phonegap-plugin-barcodescanner*[1] into the project. In fact the ability of scanning a qr-code is quite important for the *cisc* service.

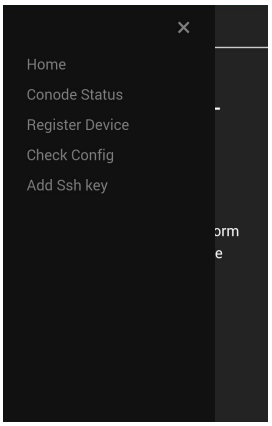

Figure 4: Cothority Mobile sidebar

2.4 Database

As you should now know, the *cisc* service allows the user to add a device to an access-control-list by giving his public key, and then to store multiple SSH keys on it. Obviously you are not storing the private keys into the skipchain, but you have to store them somewhere: this justifies the need of a local database for the application.

We need to store two kinds of public key, thus it seems natural to create two tables, one for each.

conodes *(address TEXT PRIMARY KEY, serverId TEXT NOT NULL, deviceId TEXT NOT NULL, keyPair TEXT NOT NULL, UNIQUE (serverId), UNIQUE (keyPair))*

ssh *(serverAddr TEXT, sshName TEXT, sshKeyPair TEXT NOT NULL, PRIMARY KEY (serverAddr, sshName), UNIQUE (sshKeyPair))*

Each time the device is added to a different access-control-list, a new entry is added to the first table. Such entry contains: the IP address and the ID of the server, respectively in the *address* and *serverId* fields, the unique ID that identifies the device inside the access-control-list as *deviceId*, and of course the *keyPair*. As suggested by Mr. Gasser the IP address of a conode is unique, therefore it's used as primary key.

A single device can store multiple SSH keys into the same access-control-list, as long as it gives a different name to each key, thus the primary key of the second table is composed by *serverAddr* and *sshName*. The third field of the **ssh** table is *sshKeyPair*, the SSH key pair itself.

[1] <script type="text/javascript" src="..." ></script>

For completness, figure 5 shows the ER Schema of the database. In reality the database structure evolved during the development of the application and I had to overhaul it – almost completely – more than once. Obviously the structure described above is the final one.

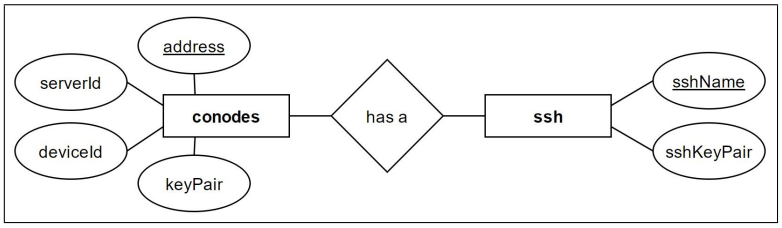

Figure 5: database ER Schema

3 Implementation

Now that you have an overview of the database structure and the setup procedure, we will discuss the implementation of the database and all functionalities of the mobile application.

Note that this application is based on the branch *cisc_qrcode* of the DEDIS repository *cothority* [6].

3.1 Database

After analyzing different possibilities, I decided to use the database API described in the *PhoneGap* documentation, mainly because I wanted to keep the project as vanilla *PhoneGap* as possible.

There is not much to say about the implementation of the database itself, apart from the fact that each call to it is asynchronous. So, ideally, whenever you access the database, you have to wait for the database action to finish before doing anything else. You may know that ES6 introduced promises, which solve this problem, but with pure *PhoneGap* you have to use common JavaScript. Therefore I created a database helper that wraps the official API in a series of functions taking an handler as an argument. Once the database action has been completed, the handler is triggered.

3.2 Conode Status

All DEDIS conodes, when receiving an empty message, send a message[2] containing information about their status. The first functionality I added to the application was, given a valid conode address, sending an empty message to it and showing its current status to the user. The main reason for this feature – which is not strictly bounded to *cisc* – was to test whether the communication between the device and the conodes was working correctly.

[2]more precisely a *StatusResponse*

3.3 WebSocket

The conodes communicate through the ProtoBuf protocol over WebSocket. Thus, while implementing the simple feature described above, I also started coding a JavaScript file that basically became my WebSocket API through the whole project. It includes a "core" function that takes care of using a WebSocket connection in order to send a message, and eventually manage the response. This function is then used by a series of other methods, one for each kind of message, that basically forward their arguments – after adapting them – to the "core" function.

The idea behind WebSocket connections is to reuse the same connection for subsequent messages, instead of a different one for each message. The conodes accept just one kind of message over a given connection, therefore I defined one map for each kind of message. Each map, when indexed with a conode address, returns the existing connection with this conode for this kind of message, if there is one.

As you can guess, sending a message usually implies to wait for a response, which is another asynchronous task. As for the database, I dealt with it by using some handlers.

3.4 Device registration

The procedure to register a device to a conode is composed by two main phases.

1. Scanning the qr-code encoding the conode ID.

2. Creating and sending a proposition.

The html file for this feature contains an element for each phase. The style of those elements is changed programmatically to display them or not, according to the current phase.

Phase one

Through the command *'cisc id qr'* the user can print a qr-code, encoding the conode ID, to the terminal. Once the qr-code is displayed, the user has to scan it with his device: the result of the scan will be a string of the form *'cisc://ipAddress:port/conodeId'*. If the qr-code is a valid "*cisc* qr-code", the application will then extract the conode address and the ID from it. Since a given device can be added only once to a single access-control-list, we need to check whether the device has already been registered to the conode before sending a *ConfigUpdate* message to it. Finally the *Config*[3] file of the conode is extracted from the conode response: we move to the next phase.

[3]Figure 3 shows its structure

Phase two
In this phase the user has to insert a unique ID that will be used to identify the device inside the access-control-list. After that a valid ID has been inserted, a new key pair will be created using the CryproJS library. The key pair, together with the unique ID and the information extracted from the qr-code, will be stored into the **conodes** table. Then an entry for the device is added to the *Config* obtained in phase one and a *ProposeSend* is sent to the conode. One of the fields of a *ProposeSend* message is the *Config* structure: in order to make a proposition it suffices to initialize this field to be the updated *Config*. The server will then send a response to confirm it received the proposition; when this happens an informative message is displayed to the user and the procedure is ended. The user has to wait that his proposition reaches the needed threshold of vote.

3.5 Check Config

After proposing a new device you may want to check whether it has been accepted yet or not. The simplest way to do this is sending a *ConfigUpdate* and verifying if your device appears inside the *Config* you receive as a response. Another thing you can do, knowing a conode address and ID, is to check if there are any propositions ready to be voted. This two functionalities are very similar and, for this reason, are provided both by the same page inside the application.

The "Check Config" page comes with a simple tab menu and with a list of all skipchains joined by the device, constructed from the **conodes** table. The entries of the tab menu are "Status" and "Update". Clicking on one of the conodes will send a *ConfigUpdate* or a *ProposeUpdate* to it, depending on the current tab. In the first case the response of the server will contain the actual *Config* of the conode. In the second, the response will contain the modified *Config* that has not reached the threshold yet, or it will be empty in case there is no active proposition. At this point the received *Config* is displayed to the user, unless the response was an empty message. Fortunately the user can finally check whether his proposition has been accepted or if there is any proposition to vote. Additionally, if there is any proposition to vote, the "vote button" will appear when sending a *ProposeUpdate*, but this will be discussed in the next subsection.

For what concerns implementation detail, the idea explained before – changing programmatically the style of a given html element – is reused here. Apart from that, there is not much to say.

3.6 Vote

As anticipated, Cothority Mobile offers to the user the possibility to vote propositions and make them reach the votes threshold. The procedure to

vote a proposition is the following one.

1. Compute the hash of the *Config* containing the proposition.
2. Sign the resulting hash with the private key used to register the device.
3. Send a *ProposeVote* message to the interested conode with the Schnorr signature generated in the previous point.

Points 1. and 2. are completed thanks to the CryptoJS library. It suffices to have the *Config* and the correct key pair to reduce their implementation to a simple function call. Usually the *Config* is obtained by sending a *ProposeUpdate*, which implies you know the conode address. Given this latter you can easily retrieve from the database the corresponding key pair.

Once the Schnorr signature has been computed, all what remains to do is to send the *ProposeVote* message, which is done with the aid of the Cothority Protobuf library and of the WebSocket pseudo API described above.

3.7 SSH keys registration

From a certain point, of view adding SSH keys to access-control-lists can be defined as the main feature of the *cisc* service. As usual, I will now list the steps involved in this procedure one by one.

1. Choose a conode between the ones to which your device has been added.
2. Send a *ConfigUpdate* message to it, in order to obtain the actual *Config* of the conode.
3. Create a new SSH key pair, give it a unique name and store it into the local database.
4. Add the public key along with its unique name to the *Config*.
5. Propose the updated *Config*.
6. Vote your own proposition.

Those steps are subdivided in three different phases, which – once again – are implemented by changing programmatically the visibility of each element of the html *body*.

Phase one
It includes the first two points and it is implemented in a similar way as the Check Config page described in section 3.5. All conode addresses stored in the database are shown to the user, he can select one of them and, as a result, a *ConfigUpdate* message is sent to this conode.

Phase two
The user has to insert a unique ID for the new SSH key pair. Once a valid ID has been inserted, a new SSH key pair is generated and stored as a new row in the **ssh** table, together with its fresh ID and the conode address. Then, the *Config* obtained in phase one is updated with the new ssh public key and used to create a *ProposeSend* message. As soon as the conode answers to this message, a *ProposeVote* is sent with the purpose of voting the proposition you just made. Obviously the vote is constructed as explained in the previous section.

Note that what just described, from the user point of view, is reduced to a simple button click, but in reality it includes the last four points listed above.

Phase three
Phase three basically consists in showing a simple informative message to the user. This is done when receiving from the conode a message that confirms it has received your vote.

3.8 General considerations

The philosophy I adopted is to wrap – when possible – all functions needed for a given functionality into a single JavaScript file. As a side effect, this implies some duplicate code.

Regarding test implementation, as you will discover in the "Results Analysis" section I had to completely overhaul some parts of the project more than once. I started by coding tests together with functionalities, but seeing that most of them became useless one week after being coded, I stopped doing it. At the end of the project I found a node module[4] which allows to setup a mock server for unit testing, but unfortunately I did not have enough time to set it up. As a result the coverage and the quality of some tests are not optimal.

4 Results analysis

Now it should be clear enough what Cothority Mobile can do, thus in this section we will mainly discuss what it cannot do. But first, I think it would be worth the effort to quickly go through the problems encountered during the development, in order to fully understand those limitations.

4.1 Ecountered problems

There are some minor issues and a couple of major ones I encountered while developing this application.

[4] karma-websocket-server

Minor issues
With "minor issues" I mean those that only made me lose time but were not a big deal. Nevertheless some of them were easily avoidable, thus I think it would be useful to quickly go through them to allow anyone who will build a similar application to prevent those situations and to save time.

I already said that in reality I did not implemented the libraries and the application in the same order they were presented in this report. As a result I had to rewrite some parts of code in order to adapt them to the new task I just received. As already explained, this mostly happened with the database.

In the introduction section I outlined how, in order to build an iOS application with PhoneGap Build, you need a valid Apple certificate. During the first part of the semester I had no certificate and, as a consequence, I was unable to test the iOS version. When I finally received the certificate[5] I had to partially revise the code – especially for what concerned the layout – in order to make the application work correctly in both Android and iOS devices.

React and ES6
The new website of the laboratory, implemented during the semester, is written in React and ES6, therefore the initial idea was to do the same with the mobile application. I found an official template [10] to make it possible and started building the application on top of it. After a series of problems I discovered, reading an issue on the github page of the template, that at this time[6] the template was not working on real devices. As a result I had to restart almost from scratch.

From PoP to cisc
At the beginning of the project I started working on *PoP*, another service developed by the DEDIS, but I was unable to contact a conode. At a certain moment we decided, together with Mr. Gasser, to give up with *PoP* and to try with *cisc*. After a week I was stuck at the same point: the server was not sending any response to my messages. During a meeting, Mr. Gasser setup PuTTY on my laptop so that I could connect to the conode and see the output generated by my messages, but there was none. At this point, talking with the master student doing a project in the same laboratory, I discovered two things. First of all the conodes are not sending any response in case of error, they simply ignore the received message, exactly as in my case. Secondly they may not even write information about the error to the terminal. In other words, the only thing you can do in this situation is to

[5]The 12 April 2017 (week 8 of the semester)
[6]It was the 25 March April 2017 (week 5 of the semester)

use debug printing and try to understand what is going wrong. After a few weeks, I finally knew how to proceed in order to solve this issue.

In the end the problem was not bounded to my code as we thought, but rather to the server itself. Unfortunately I did not have enough time to verify whether the problem with PoP was a similar one.

4.2 Limitations

Now that you have a global overview of all problems that arose during the development of this mobile application, you may better understand the origins of its limitations.

PhoneGap
There is one major limitation directly bounded to *PhoneGap*. After reaching a blind alley with the React/ES6 template, we decided to use vanilla *PhoneGap*. With it, you cannot use node modules, the require syntax or everything different from a script tag to import an external library.

Beside this if you develop a native Android application – for example – you have access to tools like *Espresso*, which simplifies automated testing. When using *PhoneGap* there is no such tool available, thus automated testing is more complicated.

Cothority Mobile
Considering Cothority Mobile itself, there is a main weakness: in case an error occurs while communicating with the server, it will stuck. The problem is due to the fact that – as already pointed out – the server is not sending any message in case of error, making this situation hard to handle. While developing the database and the WebSocket pseudo API, I experimented a couple of different ways to manage asynchronous events in *PhoneGap*. The only one – between my tries – that seemed to be trustworthy, and worked on both Android and iOS, was to use some kind of handler, but in order to use this approach we have to receive an error message. Probably – keeping the server as it is – the best solution would have been to use some sort of timer, although the inability to use node modules would have made the implementation more complex.

Beside this robustness problem, this mobile application comes with another quite important limitation. As you should know at this point, the idea behind *cisc* is to add multiple SSH public key to an access-control-list. Obviously before adding an SSH public key you have to create it, but right now you cannot. The mobile application allows the user to make a proposition, but in reality what you are adding to the access-control-list is an EdDSA public key instead of an SSH one.

Another little problem is the layout. You are not sure that with the same code you obtain the same result both on iOS and on Android. Usually

you can find a compromise to make it work on both devices, sometimes this is harder to do. Cothority Mobile tries to find a compromise whenever is possible, giving priority to Android when it is not.

5 Future work

Talking about future work, I think the list here below resumes quite well what can be done.

1. Handle server errors.
2. Allow the user to create SSH key pairs through the application.
3. Finalize and test the Cothority Protobuf library.
4. Finalize the CryptoJS library.
5. Extend the application with other DEDIS services.

The first two points were already discussed in the previous section and should be quite clear. For what concerns the other three, we may give them a closer look here.

Cothority Protobuf
Right now the library does not provide all the messages supported by the conodes. Actually, beside the *cisc* messages, I have also implemented the ones used by *PoP*, but – as you now know – they have not been tested yet. It may be interesting to test them and to finalize the library by adding all remaining messages. Note that an important part of this work would be the test phase, which may imply to debug the server itself.

CryptoJS
I did not implemented a function to verify a Schnorr signature: such function is not used by Cothority Mobile but it would make the library more complete. Beside this function there may be other functionalities that can be added to this library. While working on the CryptoJS library, the main difficulty would be linked to the compatibility between JavaScript and Go, as explained in the "Architecture" section.

Other services
There are some other services beside *cisc* (like *PoP* for example) that have been developed by DEDIS. Thus, a possible idea for future work, would surely be to extend Cothority Mobile to use one or more of those services.

6 Installation

Note that this sections, as well as the whole report, is based on the source code as it is today, the 9^{th} June 2017. This also holds for the libraries Cothority Protobuf and CryptoJS. A link to the source code of both the two libraries and of Cothority Mobile is provided below.

Cothority Mobile:	*https://github.com/lromerio/cothority-mobile*
Cothority Protobuf:	*https://github.com/Gilthoniel/CothorityProtoBuf*
CryptoJS:	*https://github.com/Gilthoniel/CryptoJS*

In order to get the project working on your machine, first run the command *'npm i'* to install the dependencies, then *'npm i -g phonegap'* to install *PhoneGap*.

To preview the application on a real device you need to download *PhoneGap Developer* on it, then run the command *'phonegap serve'* on your computer, insert into *PhoneGap Developer* the ip address that will be displayed on the terminal, and finally press connect. Note that your computer and your device have to be connected to the same WLAN.

If you want to test the application on an Android emulator you first have to install the Android SDK, then simply run *'phonegap run android'*. A similar command for iOS should exist, but I never used it since in order to run an iOS emulator you have to work on an iOS machine and it is not my case. You can give a look to the official tutorial [8] for further information.

Note that the source code includes the *apk* of the application. I could not do the same for the iOS version because of the certificate problem already explained.

7 Conclusion

PhoneGap

One of the goals of this project was to experiment *PhoneGap* and evaluate this cross-platform framework.

In my opinion, *PhoneGap* can be very powerful for developing an application which aims to provide some kind of information, and thus has not a lot of functionalities. If instead your application has a lot of functionalities, in particular if some of them are really specific, I don't think *PhoneGap* would be the best pick. In other words, when choosing this framework, you are giving priority to saving time – by developing both Android and iOS version at once – over having full access to the native API of the device operating system.

In the specific case of Cothority Mobile there are some other factors to consider and – from my point of view – the line is very subtle. It needs some very specific functionalities like sending messages over WebSocket and the use of cryptographic functions. Therefore it seems that *PhoneGap* would

be the wrong choice. In reality, given the two libraries implemented during this semester, those functionalities are no more complicated to provide with JavaScript. Anyway it still remains the fact that, with vanilla *PhoneGap*, you cannot use node modules.

Thus, in this case, the determiner factor would probably be a personal preference.

Personal experience
I have to say that the final result is not exactly what I hoped at the beginning of this project, mainly because of the already discussed problematics that slowed me down. However I think one of the main objectives of a bachelor project is to acquire new knowledges: during this semester I have learned how to develop an application with PhoneGap, how to setup a web project from scratch and many other things. Also I now have more confidence in my capabilities as a programmer, thus, in conclusion, I am overall satisfied.

8 Bibliography

[1] Barcodescanner Plugin. Master, 3 June 2017

[2] Cothority Protobuf. Master, 3 June 2017

[3] DEDIS. Cisc, 2 June 2017

[4] DEDIS. Cothority, 3 June 2017

[5] GopherJS. Master, 3 June 2017

[6] Kokoris-Kogias E., Gasser L., Khoffi I., Jovanovic P., Gailly N., Ford B. *Managing Identities Using Blockchains and CoSi*

[7] Nikitin K., Kokoris-Kogias E., Jovanovic P., Gasser L., Gailly N., Khoffi I., Cappos J., and Ford B. *CHAINIAC: Software-Update Transparency via Collectively Signed Skipchains and Verified Builds*

[8] PhoneGap. Getting Started, 7 June 2017

[9] PhoneGap. Homepage, 2 June 2017

[10] PhoneGap. PhoneGap CLI 6.0.0, 7 June 2017

[11] PhoneGap. Signing, 8 June 2017

[12] PhoneGap Build. Homepage, 2 June 2017

[13] Wikipedia. PhoneGap, 2 June 2017