Extending the web-frontend for the cothority-framework

Gaylor Bosson

School of Computer and Communication Sciences

Decentralized and Distributed Systems lab

Semester Project

May 2017

Responsible
Prof. Bryan Ford
EPFL / DEDIS

Supervisor
Linus Gasser
EPFL / DEDIS

Contents

Chapter 1

Introduction

The DEDIS laboratory in the EPFL has developed a cryptographic protocol framework name Cothority[2] for collective authority. This framework delivers distributed security primitives as signatures or proof of person. We will come back to the explanations of the framework in the next chapter.

In order to improve this framework, a first iteration of a front-end has been developed and this project has the purpose to improve it. The previous website was able to display a static list of available nodes of the Cothority with the different statistics as the traffic or the uptime of each node. You were also allowed to upload a file to sign it with the available members of the Cothority and verify it later on.

The project will bring different improvements as the possibility to choose the current skip-chain. The concept will be explain also in the next chapter but you can update the current roster by choosing a different skip-chain. Another one is the storage of a website in a suite of skip-chains in order to insure the user that the HTML content is verified the by the collective authority. Those were the big changes but the project will also improve underlying performance of the web-socket management and simplify the usage of cryptographic primitives as hashing and aggregating public keys.

Chapter 2

Cothority

This project uses three different services of the Cothority framework. In order to display the status of the roster, the front-end client needs to request the Status service of each node. It also uses the skip-chain service mainly to fetch the blocks of a chain and finally the CoSi service to sign and verify.

2.1 Status

With this service we can contact each node by a web-socket protocol to know its status. This is very important because we need to know how many nodes are available in order to check if more than two third of the roster is reachable. If at least one third of the collective authority is down, we shouldn't ask for a signature.

2.2 CoSi

The motivation behind the collective signature[1] comes from the single point of failure issue of a centralized authority. Take for instance the Central Authority that web browsers trust for the SSL certificates. In order to insure the user that the website he's visiting is indeed the correct and not a tentative of phishing, we trust a chain of certificate issuers leaded by a CA. In the case of a security breach, an attacker can potentially deliver a set of bad certificates and that would be a disaster.

The collective authority can fill this security gap by removing the single point and replacing it by a set of nodes so that if one of them is successfully attacked, it won't be sufficient for the attacker. In our context, we will use the collective signature to let a user sign a file. If an attacker can take the control of a node, even if he knows the private key that is used by this node, it won't be sufficient to break the signature and he won't be able to modify the file and build a correct signature.

The front-end will allow a user to upload and sign a file according the available list of nodes. He can then send the file and the signature file to another person who can verify the file has not been modified by uploading the received file and signature. We'll see later how the signature file allows the interface to reach the roster and to execute the verify procedure.

2.3 Skip-chain

The structure of a skip-chain can be seen as one or several linked lists of blocks where a list can be linked to a parent list and so on. A block has different fields used for different purposes. One of the more important one is the `Hash` that represents the ID of the block used to identify the block but also for the different back and forward links. `BackLinkIDs` is an array of the hash of the previous blocks and `ForwardLink` is an array of the hash of the block linked to the current block.

As we said before, you can have different level of chain and you can know the position of your block with the fields `Index` and `Height`. The index is the horizontal position and the height is the vertical position. In the case the block is linked to an upper chain, the hash of the parent block will populate the field `ParentBlockID`.

2.3.1 Manager

The Cothority provides a tool to manage the skip-chains as the creation of a new chain or a helper to add a block to a existing chain. At any time you can use the parameter `-h` to get help.

In order to create a new skip-chain, there is the command `scmgr create $file` where `$file` is the path of a roster definition file that you get when deploying the nodes of the Cothority, namely `public.toml`. The parameter `--url` allows you to create an HTML skip-chain that will contain a web page. We will explain this in details in the Chapter 5.

In the case you already have a skip-chain and you want to change the roster, the command `scmgr add $id $file` will create a new block and add it the skip-chain with genesis ID `$id`. If you want to update the html content of a web skip-chain, you'll have to use `scmgr addWeb $id $file` where the ID is the same as before and `$file` is the path of the file that contains the HTML data.

It can happen you need to get the latest block of a skip-chain and for that you can simply use the command `scmgr update $id` given the ID of the skip-chain. It's also possible to get the data of the latest block by using the parameter `--data`.

Finally you can check the list of available skip-chains with `scmgr list known` but this will return only the ones known by the local manager and

then the command `scmgr list fetch $file` will fetch the list of skip-chains according to the given roster definition file in parameter. The front-end needs the list of known skip-chains and for that, there is the third usage `scmgr list index $path` where `$path` is the folder where the script will create the `*.html` files. It will generate an index with the list of blocks and also one file per skip-chain named with the ID of the genesis block. The individual file allows the front-end to fetch the list of public addresses for a given skip-chain.

Chapter 3

Front-End

This chapter will introduce briefly the technologies used in the project to develop the front-end. The objectives were to develop a single page with the different modules as the signature management and the status table but also to be able to show a single module in stand-alone page so that we can use it for instance in a different web page in a IFrame. Another one was to produce an easy template to be able for further developers to add new modules without too many constraints and the combination of React and Bootstrap is a good solution for those objectives.

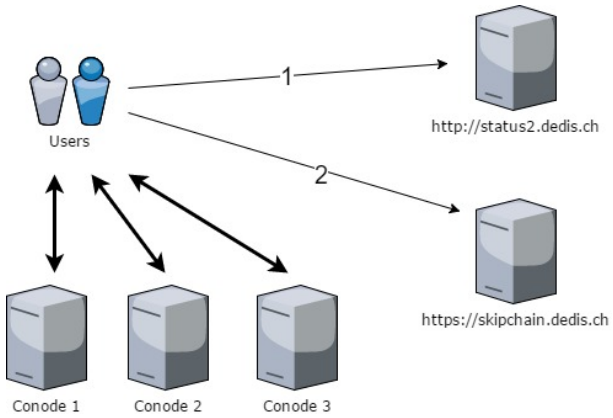

Figure 3.1: Front-End Network

The Figure 3.1 shows you the network mapping of the front-end. First of all a user will fetch the web application with the address `http://status2.dedis.ch` and then the client will fetch the list of available skip-chains with the address `https://skipchain.dedis.ch`. Finally every user can open websockets to the conodes in order to use the services.

3.1 React

This framework developed by Facebook is a new way to create a single-page application because previous frameworks were more focus on the Model-View-Controller pattern (MVC). With React[11] you are only focused on the View component and it is better for this kind of project because we don't have models and data to manage and then you can create an application quicker and in an easier way with a simpler framework.

Aside of that React is powerful in terms of rendering because you can control the life-cycle of the components to limit useless render steps.

3.2 Bootstrap

Bootstrap is essentially a CSS framework even if it also provides JS pre-made scripts for UX components. In our case we are interested in the grid layout that uses the Flex Box technology with the version 4 of Bootstrap. It's important to understand how this works because you can change the grid as you wish, with predefined or automatic width, and this for each individual cell. In order to use the JSX syntax, the application uses the reactstrap[12] module.

3.3 SASS

The basic CSS language can be quite hard when you start to deal with a lot of children because you can only use one depth when defining your rules and then the style files can become hard to read very quickly. That is one of the reasons why SASS (or the concurrent LESS) has been developed[9] but we can also quote the variables which are not available in the legacy CSS.

Chapter 4

Architecture

As you can see in the Figure 4.1 the front-end[7] is built with two additional libraries because one goal of the semester project was to share a cryptographic and a Protobuf library with the mobile application which is also written in Javascript.

The purpose of the first one is to use the primitives implemented for the Cothority and for that, the code is written in Go and then translated in Javascript using GopherJS[8]. The library itself is used to write a interface between Go and Javascript data structures.

The second one is a simple declaration of the Protobuf messages and a list of helpers to reduce the code needed to encode or decode a message transmitted between the Cothority and the front-end.

4.1 CryptoJS

As stated before, this library[6] is written in Go but compiled in JavaScript to be usable in the web and the mobile applications. The tool used for the compilation is GopherJS[8] and the cryptographic primitives are taken from the DEDIS repository crypto[3].

After importing the library, it will be available in `global.cryptoJS` but you can directly access it with `cryptoJS` as the JavaScript interpreter searches for the reference in the global object.

4.1.1 Hash

The functions `sha256(buf)` and `sha512(buf)` take a buffer as input and they will return a new buffer containing the hash of the input using the corresponding algorithm, namely SHA-256 or SHA-512.

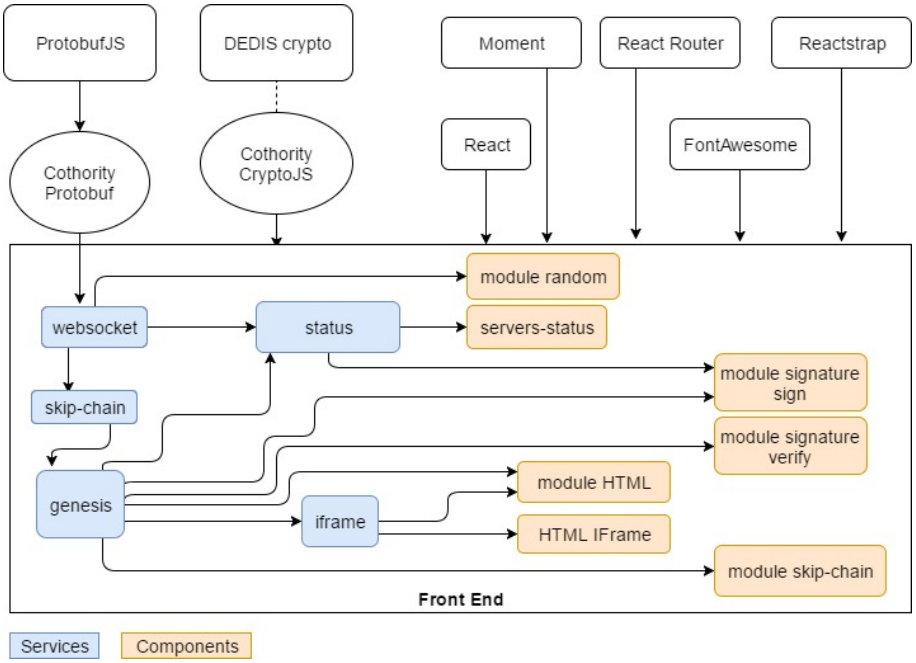

Figure 4.1: Front-End Architecture

4.1.2 Public and Private Keys

The Cothority uses a binary Marshal format for the EdDSA and then the library provides helpers to convert this format into public and private keys. The function `keyPair()` will generate a buffer representing the binary marshaling of the EdDSA and in the case where you only have the private key, you can use `keyPairFromPrivate(privateKey)` to rebuild the EdDSA. If you need the public key from the binary Marshaling, you can simply use `publicKey(marshal)` and provide the binary marshaling.

For the more specific case of the signature where you need to aggregate the public keys, the library provide the function `aggregateKeys(keys)` that takes an array of buffer and each buffer contains a public key.

4.1.3 Signature

If you want to sign a message or a file you need to have a valid EdDSA and a buffer containing the data you want to sign. Then you can use the function `sign(marshal, message)` that will return a buffer with the resulting signature.

For the verifying process, in that case you need the aggregated public key, the message and the corresponding signature that you provide to the function `verify(publicKey, message, signature)` and it will return `true` or `false`.

4.1.4 Skip-chain

Two functions are provided for the skipchains. `hashSkipBlock(block)` will return the hash of the given block. You must provide the block object that you received from the Cothority because the Go implementation will search for specific fields.

The second is useful to verify the forward links of the skipchain. You must provide the forward link to the function `verifyForwardLink(link)` and it will return a boolean of the result.

4.2 Cothority Protobuf

The library[5] uses proto files for the definition of the messages[4]. But as we want to create a bundle for the library mostly because of the mobile application, we need a way to convert those files in a Javascript piece of code. That's why you have to run `build_proto.js` before trying to compile if you made modifications on the message definitions. This script will transpile every file ending by `.proto` located in the `src/models/` folder into a string that can be exported in any Javascript module. This string is simply the JSON definition of the messages that will be loaded in the root.

The `src/cothority-protobuf.js` file declared the base class of the library with two important public functions `encodeMessage(name, fields)` and `decodeMessage(name, buffer)`. The first will encode the given fields with the message of the given name. Of course it must defined in the models. The second function will decode a given buffer typically received from the Cothority with the message of the given name.

Finally the class `CothorityMessages` declared in the index file contains a list of helpers that can be useful to check the types of the parameters or to simplify the code. You can take a look at the documentation located in the folder `doc/index.html`

4.3 Services

The different services are useful to synchronize the state as for instance the current skipchain chosen to be displayed in the status table. The React components have their own states but you cannot share them directly to other components except the children. With the services you can manage the events triggered by a global state change.

4.3.1 Genesis

This is the entry point of the web application in the sense that this service will fetch the list of available skip-chains from `https://skipchain.dedis.ch`

and pick the first one that doesn't contain a HTML page. The result of the request has the format:

```
{
    "Blocks": [{
        "GenesisID": "0b8d24c8d3...",
        "Servers": [
            "192.33.210.8:7002",
            "192.33.210.8:7004",
            "192.33.210.8:7006"
        ],
        "Data": "3c1b8rA7XN66xq/fn3jvQQoA"
    }, ...]
}
```

Those three fields are the minimal information required to set up the web application. The genesis ID is used to fetch the blocks of the skipchain and we need the IP addresses of the roster. The data field is only used to differentiate a simple skip-chain used for signatures from an HTML skip-chain that we use to store a web site.

You can then interact with the service with different public functions. If a component needs to listen for changes of the current active skip-chain, you can use the function `subscribe(listener)` and `unsubscribe(listener)` when the component will be destroyed. The listener you pass as parameter should have `onGenesisUpdate(blocks, skipchainList, currentSkipchain)` and `onGenesisError(error)` declared as functions if you want to get the events but they are not compulsory. The update event will give the list of blocks of the active skip-chain, the list of available skip-chains and finally the genesis ID of the active skip-chain.

You can also change the current skip-chain with `setCurrentGenesisID(id)` and the ID of a skip-chain. After fetching the latest blocks, it will trigger the update event with the new data to every subscribed listener. In the case you need the latest block of a skip-chain given a genesis ID but without changing the active skip-chain, you can use `getLatestFromGenesisID(id, blockID)`. If you specified the block ID it will return this block instead of the latest of the chain if it exists.

4.3.2 Status

It will first subscribe to the genesis service to listen for the active skip-chain. After getting the latest block of the active one, it will fetch the status of every server in the roster and trigger an update event for every response.

As for the previous service, you can use `subscribe(listener)` and `unsubscribe(listener)` to get the update events and you need to declare

the function `onStatusUpdate(status, skipchainList)`. The `status` object has the format:

```
{
    "192.33.210.8:7003": {
        "server": ServerIdentity,
        "system": {
            "Status": Status
        },
        "timestamp": 1495281043289
    }
}
```

`ServerIdentity` and `Status` are not written down for simplicity but you can check their definition with `StatusResponse.proto`. The `timestamp` field is the timestamp of the response of the status for this server.

The public functions `getAvailableRoster()` and `getOfflineRoster()` return an array containing the online, respectively offline, roster. The objects inside the array have the same format than the ones from the status update event.

4.3.3 Skip-chain

This service has only one public function that is `getLatestBlock(servers, genesisID)`. Given a list of servers it will try to get the list of blocks using the addresses one by one. When a server respond, the service will first check the validity of the blocks and if everything is fine, it will resolve the promise with the list. In the other case, the promise will simply reject.

4.3.4 Websocket

Every request to the Cothority's nodes are made using the websocket protocol. This service takes care of maintaining the connection so that you don't have to create a new socket every time you want to make a request. It also provide helpers for the requests:

```
getStatus(address) { ... }

getSignature(hash, address, roster) { ... }

getLatestBlock(address, id) { ... }

getRandom(address) { ... }
```

The addresses must include the port and you can use either the IP or the URL but the protocol is managed by the service. In the case of the signature, the hash is a buffer of the message you want to sign and the roster is the object you received from the Cothority. For the list of blocks, the id must be a buffer.

4.3.5 IFrame

This service uses again a observer design pattern with `subscribe(listener)` and `unsubscribe(listener)` and the events will trigger either `onOpenHTML(html)` or `onCloseHTML()`. The first will trigger after you used the function `open(id)` with the ID of the genesis block of the HTML skip-chain. This will fetch the latest block and send the HTML data to every listener. The second will warn every listener that the function `back()` has been called.

4.4 Components

The homepage of the web application is built around the grid system of Bootstrap and different modules that use the `Module` component as a base. The status table is slightly different and doesn't use the base module component.

As we want to display some modules as stand-alone, React Router is used to enable the navigation and then you can create a page to display one or more modules and the using an IFrame, you can display this page in an external website.

4.4.1 Module

As stated before, this component renders the base structure of a module. It is composed with a header where you can set the icon and the title and with body that will be filled with the children.

```
<Module title="..." icon="..." className="...">
    {children}
</Module>
```

The icon must be taken from `http://fontawesome.io/icons/` and the title can be any valid string. The component will always have the `module` class and it will concatenate the `className` property. Inside the module tag you can write any valid JSX code.

4.4.2 HTML Module

The purpose of this module is to display the list of HTML skip-chains and give to the user a way to open the website emulator. One thing to note is

that we only show the pages with a base URL meaning we don't want to show every single web skip-chains but only the entry point of each.

4.4.3 Random Module

This component will fetch a number from the random service of the Cothority and display it alongside with the timestamp. An update will be triggered every thirty seconds and a graphical counter is shown inside the component.

4.4.4 Signature

There are two modules, one to sign and another to verify a signature. In both we first render a dropzone so that the user can either click to open a file picker or drag a file from outside the browser.

For the sign module, after uploading a file, the user will be able to click on a cancel button or a confirm button to execute the signature request. At the end of the request, a file will automatically be downloaded with the result of the signature process as the content.

For the verify module, you have to upload first the file and right after the signature JSON file. The component will guide the user with a graphical help. Directly after the JSON file has been correctly read, the verify request is sent and the result will be shown right under the help.

4.4.5 Skip-chains

As the HTML module, this one will display the list of skip-chains but this time only the ones that doesn't store HTML content. By clicking on it, the user will trigger a change of skip-chain for the status and the signatures.

4.4.6 Servers Status

Even if this component stands with the modules in the home page, it is not using the base module component. It will display a table where the rows correspond to a node of the roster and are populated with the status information. In particular the timer which displays the time since the last update is a simple component that refreshes itself every second because each status object contains the timestamp of the last update that is managed the service explained in the previous section.

4.4.7 HTML IFrame

In order to render the HTML content of the web skip-chains, we use this dedicated component that will render a fixed IFrame after the user clicked on a web skip-chain. When there is no content to show, this component renders `null`.

4.5 Utils

The functions explained in this section are used in different places in the project and are not specially bound to a component or a service.

4.5.1 Buffer

We need quite often to convert an hexadecimal string, e.g. a genesis block ID, in a byte buffer and inversely because for instance protobuf messages take a buffer as parameter. And then the file `src/utils/buffer.js` defines two functions `hex2buf(hex)` and `buf2hex(buffer)` to carry out those conversions.

4.5.2 Files

In particular for the signature flow, we need to open and to hash the files. In order to use a `Promise` syntax, the file `src/utils/file.js` defines `hashFile(file)` and `readAsString(file)` which both will return a promise that will resolve a buffer containing the hash in the first case and the content in the second.

The last function is `reduceFileSize(file)` that will return a human readable size of the file.

4.5.3 Network

As you should be aware of, the servers are defined with an address for a TCP protocol but the front-end application is using the websocket protocol to request the nodes and the address is the same except the port that is an increment of one. The conversion can be easily performed with the function `tcp2ws(address)` defined in the file `src/utils/network.js`

Chapter 5

Website Inliner

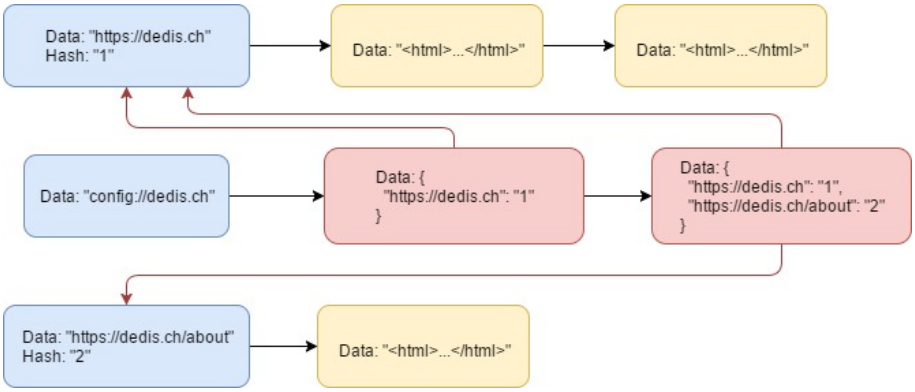

Figure 5.1: Web skip-chain

One possible way to use the skip-chains is to store the pages of a website inside the data field, one chain per page as described in the Figure 5.1. One skip-chain is used as the configuration chain where we store the mapping of the URLs of the website to the ID of the skip-chain (the genesis block ID). Then we store the HTML content in each specific skip-chain but for that you need to pack the different elements of the page inside a inline string.

In order to simplify the process to set up an online site, the front-end repository delivers the script file `website-inliner-v2.js` that you can use to store or update a website with the command `node website-inliner-v2.js $url -f $file -b $blockID`. The `$file` parameter is the `public.toml` file of the cothority and the `$blockID` parameter is the ID of the genesis block of the configuration skip-chain. The $url parameter will be used to search for pages then you need to provide the base URL of the website.

This script use the NodeJS module Inliner[10] because you need to convert the distant assets as the images into an inline HTML content then recursively fetch all the pages to visit the entire public tree of the website

given as input.

Because the world wide web is very heterogeneous, you cannot create a simple script that will convert any existing website into a skip-chain and thus this script is optimized for the DEDIS website.

Chapter 6

Conclusion

This project used the first iteration of the Cothority front-end application as a example and improved it to extend the features available to the users. The previous application let you check the status of a fixed list of servers and also sign and verify signatures of files.

The first improvement is the code itself because it is built in a way it's easy to add or extend the existing features with the modules structure. You can add or create a module for a different usage of the Cothority or create a stand-alone access as we did for the random service.

The second is the usage of the skip-chains to define dynamically the roster of the interface so that a user can switch between different chains and then select a different set of nodes, for instance if a roster is down. The signature flow has also been slightly improved regarding the user experience with more information provided.

Finally we add a totally new feature that is the HTML skip-chains that store the content of a website inside multiple skip-chains and the interface provide a way to extract those contents and show them inside an iframe.

References

[1] E. Syta et al. Keeping Authorities "Honest or Bust" with Decentralized Witness Cosigning. (2016). https://arxiv.org/pdf/1503.08768

[2] DEDIS Cothority Framework. (2017, June). https://github.com/dedis/cothority/wiki

[3] DEDIS Crypto. (2017, June). https://github.com/dedis/crypto

[4] dcodeIO/protobuf.js. (2017, June). https://github.com/dcodeIO/protobuf.js/wiki

[5] Cothority Protobuf library. (2017, June). https://github.com/Gilthoniel/CothorityProtoBuf

[6] Cothority CryptoJS library. (2017, June). https://github.com/Gilthoniel/CryptoJS

[7] Cothority Web. (2017, June). https://github.com/Gilthoniel/cothority-web

[8] GopherJS. (2017, June). https://github.com/gopherjs/gopherjs

[9] SASS Framework. (2017, June). http://sass-lang.com/guide

[10] Remy/Inliner. (2017, June). https://github.com/remy/inliner

[11] React Framework. (2017, June). https://facebook.github.io/react/

[12] Reactstrap. (2017, June). https://reactstrap.github.io/