ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Web-Frontend for Cothority

Bastian Nanchen

School of Computer and Communication Sciences

Decentralized and Distributed Systems lab

Semester Project

December 2016

Responsible
Prof.Bryan Ford
EPFL / DEDIS

Supervisor
Linus Gasser
EPFL / DEDIS

Contents

1 Introduction

The decentralized and distributed systems (DEDIS) team at EPFL is working among others on a software project called Collective Authority (Cothority). Cothority is composed of multiple conodes, which are servers running protocols and services. It implements different applications as Collective Signing (CoSi), Cisc (a distributed key/value storage handled by a blockchain with an SSH-plugin), Proof of Personhood (prove the existence of humain-being), Guard (use distributed servers to hash passwords), Status (returns the status of a conode) [?].

Distributed cryptography spreads the operation of a cryptosystem among a group of servers in a fault-tolerant way [?]. Cothority (Decentralized Withness Cosigning) is a "multi-party cryptographic signatures" [?].
In order to accomplish that they developed the CoSi protocol, which will produce a collective signature by decentralized servers. To perform a collective signature the protocol needs the hash of a file. The outcoming signature has the same verification cost and size as an individual signature. A digital signature is used to verify a file's origin and content.
This endeavor addresses a significant issue. For example a certificate or a software update now needs one single signature from a corporation or a government (or anything/anybody) to validate it. This represents a high-value item for criminals, intelligence agency,...
The CoSi protocol provides a validation, which is produced by a group of independent parties (named conodes), at every authoritative statement before any device or client uses it [?].

This report initially describes the aims and goals of the semester project. Then it introduces the tools used for its implementation. Afterward it describes the problems, that arose during the development of the project and the solutions found to solve them. This leads the report to present the results of the project and the theorical and practical limitations of it. Therefore the report suggests possible future works to improve the project. At the end the theoritical and practical knowledge gained through the project are depicted and a step-by-step guide for installation and running of the final product is detailled.

2 Aims and goals

The goal of this semester project is to furnish a web-interface to the Cothority project. As Cothority's applications, the website uses CoSi and Status.
The aims stated are to provide a status-table with informations like port number, name or bandwidth used for each contacted conode, to be able to submit a file for a digital signature using the CoSi protocol and to verify if a digital signature corresponds to a particular file.

3 Tools used

Obviously the language choosen to implement the web-interface is JavaScript. The HTML markup language and the CSS style sheet language are used as well.
The Bootstrap framework [?] is employed for designing the website.

3.1 Libraries

Several libraries are used in the semester project.
The jQuery library [?] is used to facilitate the selection and modification of DOM (Document Object Model) elements.

The protobuf.js [?] is a pure JavaScript implementation of Google's Protobuf. It uses the same format of .proto file. The Cothority uses Google's Protobuf. It seems evident to use the same way for the web-interface.
"Protocol buffers are a flexible, efficient, automated mechanism for serializing structured data-think XML, but smaller, faster, and simpler." [?].

```
message Foo{
            required bytes a = 1;
            optional bytes b = 2;
        }
```

Listing 1: Example of .proto file

A .proto file is composed of protocol buffer message(s). Each message contains name-value pair(s). The value is a unique tag. Each tag is "used to

identify your fields in the message binary format" [?]. Each pair has a type. There are multiple disponible tags.
The last element to define is the rule field. The web-interface uses two rule fields: "required" and "optional". The protocol buffer message with a "required" field is obligate to send or receive the field, contrary to the message with a "optional" field, which is not obligate to send or receive the field in question.

The js-nacl [?] library is adopted in the Verification part of the project. As said on the library's GitHub page: "A high-level Javascript API wrapping an Emscripten-compiled libsodium, a cryptographic library based on NaCl. Includes both in-browser and node.js support.". NaCl (Networking and Cryptography library) is a software library written in C for network communication, encryption, decryption, signatures,... [?]. Its goal is to "provide all of the core operations needed to build higher-level cryptographic tools" [?]. Little disclaimer NaCl is pronounced "salt".

Another JavaScript NaCl library TweetNaCl.js [?] is used because the js-nacl doesn't implement two essential functions. More will be told when the time comes.

Other libraries like js-nacl, protobuf.js,etc. are used and will be presented in subsections below.

4 Problems that arose and their solutions

In the following subsections the problems arose and their solutions are treated. First the issue of the communication between the client and the server is depicted. Issue which will lead to the complication generated by asynchronization and how using JavaScript Promise APIs and generator function the program solves it.
Afterwards the approach used to send a file for collective signing and the verification of a signature is detailled.

4.1 Communication client/server

The first problem to tackle was to create a communication between the website and a conode.

The object Websocket [?] of the JavaScript Web APIs offers the tools to create a communication between a browser and a server and send/receive data.
At first an empty protocol buffer message is sent in a Blob [?] object containing the .proto file in bytes.

```
message Request {
}
```

Listing 2: Empty protocol buffer message

The request being sent, through a Websocket object, the web page waits for response from the conode.

```
message ServerIdentity{
     required bytes public = 1;
     required bytes id = 2;
     required string address = 3;
     required string description = 4;
}

message Response {
     map<string, Status> system = 1;
     optional ServerIdentity server = 2;

     message Status {
          map<string, string> field = 1;
        }
}
```

Listing 3: Response protocol buffer messages

The response is is a map with field corresponding to another message format, which is a map of a string with a string. Each key corresponds to an element of the conode's status (port number, hostname, number of bytes received and sent, number of services, connexion type, uptime, name and version) and each field to its value.

4.2 Asynchronicity

Nevertheless the response message is triggered by our opening socket message. Therefore the response message will be asynchronous.

To tackle the asynchronous problem, the introduction of the JavaScript Promise object shall be made.
First it is important to know that JavaScript is single threaded. It means that if a code snippet is waiting on data, the thread can't preempt and doesn't execute the remaining part of the program. In that case JavaScript programs would be very slow and no user-integration while waiting would be possible. So through the history of the JavaScript language many tools were created to handle asynchronous part of code. Like callback functions that were widely used, but widely disliked too. It even has a nickname: "Callback Hell".

```
  fs.readdir(source, function (err, files) {
  if (err) {
    console.log('Error finding files: ' + err)
  } else {
    files.forEach(function (filename, fileIndex) {
      console.log(filename)
      gm(source + filename).size(function (values) {
        // some code
        )
          }
        }
      })
    })
  }
})
```

Listing 4: Example of Callback Hell with its typical pyramid shape

Other libraries (e.g. jQuery) begun implementing promises to help developers overcome the "Callback Hell". This eventually leads ECMAScript 6 to adopt the Promise API [?] natively in JavaScript [?]. The object Promise allows to retrieve a value in the "future".

```
var promise = new Promise(function(resolve, reject) {
  // asynchronous snippet code
  if (/*everything goes well*/) {
    resolve(/*result of async element*/);
  } else {
    reject(/*result of async part*/);
  }
});
```

Listing 5: Structure of a Promise

A callback function needs to be passed at the Promise's constructor. This function will have at most two parameters. The resolve function (mandatory) will be called if everything worked in the asynchronous part, otherwise the reject function (optional) will be called.
In the case of this semester project, the resolve function will return the response protocol buffer message containing all the status data from the conode.

To deal with the Promise object returned the program will use another feature of ECMAScript 6 called a Generator. It's a new kind of function. It can be paused in the middle and resumed later. Naturally other parts of the code are running during the pause.

```
function* foo() {
  var x = yield 1;
}
```

Listing 6: Structure of a generator function

The "*" marks the function as a generator one.
The other different element, with a traditional function, of the code snippet is the keyword: "yield". "The yield _____ is called a 'yield expression' (and not a statement) because when we restart the generator, we will send a value back in, and whatever we send in will be the computed result of that yield _____ expression." [?]. The function halts when it encounters the "yield" keyword.
Whenever (if ever) the generator is restarted, the "yield 1" expression (from the above generator function) will send "1" back. The part following the "yield" keyword is the expression that will be returned.

```
var a = foo();
a.next();
```

Listing 7: Restart of a generator function

The call to the "next()" method executes the generator (from the last "yield" until the next encounter of the keyword (if there is any left)).
Returning to the website's code, a generator function is used with the function returning a Promise object containing the response protocol buffer message.

```
function* generator() {
    // some code
    var message = yield websocketStatus("localhost:7003");
    listNodes.push(nodeCreation(message));
    // some code
}
```

Listing 8: Extract from the project's code

The function "websocketStatus()" is the one returning the Promise object. The generator function pauses when reaching "websocketStatus()". "The main strength of generators is that they provide a single-threaded, synchronous-looking code style, while allowing you to hide the asynchronicity away as an implementation detail." [?]. The idea is to "yield" out promises and let them restart the generator function when they are fulfilled.
To do so we need to create a new function that will control the generator function's iterator.

```
function runGenerator(g) {
  let iterator = g();
  (function iterate(message) {
    let ret = iterator.next(message);

    if (!ret.done) {
      ret.value.then(iterate);
    }
  })();
}
```

Listing 9: Extract from the project's code [?]

This function takes a generator function as parameter. The "iterator" function variable is the generator function. The inner-function "iterate()" looks if a promise returns the value "message" from "iterator.next(message)". If so, the function waits on it.
If the generator function is not completed, which implies "ret.done" is equal to false, the function will iterate again.

Now all the elements needed, to bring a solution for the asynchronous problem, have been presented. It only remains to do it for each conode (here in port 7003, 7005, 7009, 7011, 7013).

```
runGenerator(function* generator() {
  const listAddresses = ["localhost:7003",
      "localhost:7005", "localhost:7007",
      "localhost:7009", "localhost:7011",
      "localhost:7013"];

  window.listNodes = [];
  for (let address of listAddresses) {
        let message = yield websocketStatus(address);
        window.listNodes.push(nodeCreation(message));
  }
  // some code
});
```

Listing 10: Extract from the project's code reaching some conodes

As said before the "websocketStatus()" function returns a Promis object. Thus the generator function "generator()" will halt ("yield") before each "websocketStatus()" call.
The generator function is given in parameter to the "runGenerator()" function.
The Promise APIs, the generator function and the util function "runGenerator()" allow to hide the asynchronous part of the code. It is more readable and modular.

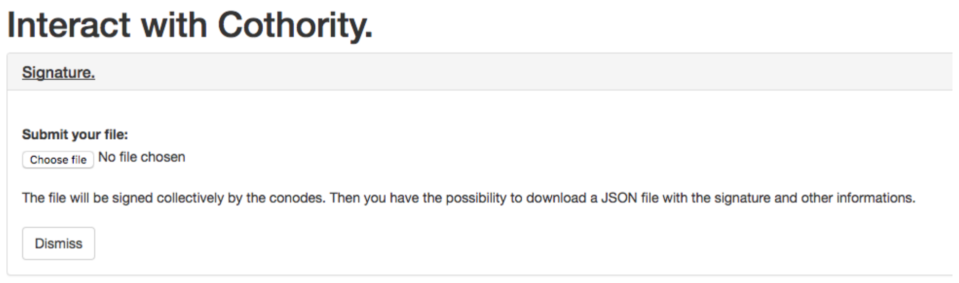
4.3 Collective Signature and Verification

The following problem to tackle is the implementation of the possibility to send a file for a collective signature and to verify a signature.
In the Cothority project the signature is a Schnorr signature, which is a zero-knowledge proof presented by Mr.Schnorr in 1989. It is "based on the intractability of certain discrete logarithm problems" [?].
In the part where the user can send a file and sign it by the conodes, the website needs to calculate the aggregate-key using the public-keys of the conodes. More details will be presented in the Implementation subsection.
The private and the public-key used in the Cothority project require the usage of the Edwards-curve Digital Signature Algorithm (edDSA). "edDSA is a variant of Schnorr's signature system with Twisted Edwards curves." [?].
The Cothority project uses Ed25519, which is an instantiation of EdDSA, for the public-key signature. Ed25519 has some interesting aspect as a fast key generation, small keys and an high security level [?].

4.3.1 Send a file for collective signing

The user has the possibility to submit a file and download a JSON file with the signature and other informations.

The implementation of the HTML part is straightforward. It uses an input tag to submit a file and all the collapse-elements are managed by the Bootstrap framework [?].


Interact with Cothority.

Signature.

Submit your file:

Choose file No file chosen

The file will be signed collectively by the conodes. Then you have the possibility to download a JSON file with the signature and other informations.

Dismiss

Figure 1: Interface

The file submitted is read as an ArrayBuffer [?]. The Promise APIs, a generator function and the "runGenerator()" function are used to deal with the asynchronous part of submitting a file. The reading of the file is returned inside a Promise object in a generator function that is given in paramter to the "runGenerator()" util function (see above for more details on the manner to tackle the asynchronous part).

Afterward the file needs to be signed.

As soon as the websocket is appropriately opened, the communication is done through a websocket using 4 protocol buffer messages.

```
message ServerIdentity{
    required bytes public = 1;
    required bytes id = 2;
    required string address = 3;
    required string description = 4;
}

message Roster {
    optional bytes id = 1;
    repeated ServerIdentity list = 2;
    optional bytes aggregate = 3;
}

message SignatureRequest {
    required bytes message = 1;
    required Roster roster = 2;
}

message SignatureResponse {
    required bytes hash = 1;
    required bytes signature = 2;
}
```

Listing 11: .proto file

For safety reason, the calculation of the aggregate-key shall be done in the program. The risk is to have a server, which could make up a pair of public and private keys and then sends the make up public-key to the website as the aggregate-key. The security of all the servers would be questioning.

Initially a list of each conode's public-key needs to be created. The status part of the website looks after collecting servers' informations every 3 seconds. A variable, which contains all the informations of the conodes, is added to the "window" object. Adding an element to the "window" object is a proper way to define a global variable in JavaScript [?]. Thus this list is used to collect each public-key of the conodes. Having that the calculation of the aggregate-key can begin. The program utilizes another NaCl library: "TweetNaCl.js" [?] to pack, negatively unpack and addition the points. The function "pack()" transforms a point x, y, z, t in Ed25519 into a JavaScript Uin8Array object. On the other hand the function "unpackneg()" transforms a JavaScript Uint8Array object into point x, y, z, t and multiplies the x-axis by -1. For the reason that it is preferable for operations in the TweetNaCl.js library.

```
const listServers = window.listNodes.map(function(node, index) {
  const server = node.server;
  const pub = new Uint8Array(server.public.toArrayBuffer());
  pub[31] ^= 128;
  const pubPos = [gf(), gf(), gf(), gf()];
  unpackneg(pubPos, pub);
  if (index === 0) {
    agg = pubPos;
  } else {
    add(agg, pubPos);
  }
```

```
    // some code
});
pack(aggKey, agg);
```
Listing 12: Extract of the code calculating the aggregate-key

In the above code the variable “pub” is the public-key of a conode retrieved thanks to the global variable “window.listNodes”.
Then the line “pub[31] ˆ= 128” is a multiplication of the x-axis by -1, due to the fact that the TweetNaCl.js has no an “unpack” function but only an “unpackneg” function as said before.
The function “gf()” from TweetNacl.js is just a function that returns an array of “0” of size 16. The variable “pubPos” is a zero-point.
Afterward the addition of the point is done using the “add()” function from the TweetNaCl.js library.

Having calculated the aggregate-key, the website sends to one conode the list of servers and the hash of the file. On the server side, the server, receiving these informations, contacts the other conodes to collectively sign the hash of the file using the CoSi protocol. Then the server responds with a message containing the signature and the hash of the file. The server’s response and the aggregate-key are entrusted to a function “saveToFile(fileSigned, filename, message)”.
The function takes as parameters the signed file contained inside an ArrayBuffer, the filename and an array message containing the signature and the aggregate-key. Then it calculates the SHA-256 hash of the file using a function from js-nacl library. The signature, the aggregate-key and the hash are translated in base64 for inclusion as a string in the JSON file.

From there on the JSON file is created and proposed to be downloaded to the website’s user. The JSON file contains the signature’s file, the filename, the date, the aggregate-key and the file’s hash.

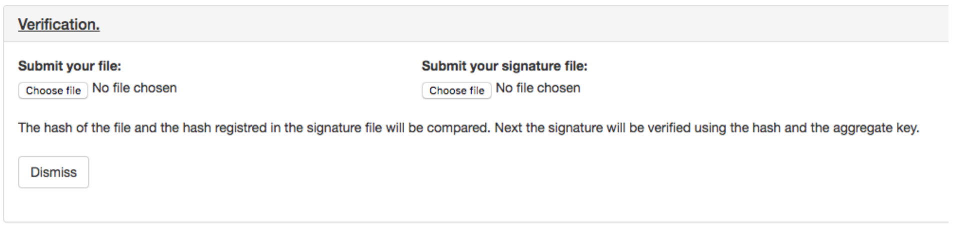
```
    "filename": "file",
    "date": "3/12/2016",
    "signature": "vVppwEgya0T22mGlKBfj4Tx+BVQQx0EAH3XFLC
                  lfwSbskCxEsPIJ62ZUoD3N7ksRCEK2M/XA6flV
                  2tLsiQmrAf4=",
    "aggregate-key": "IjgFxLpeV8IOVShIGC6ESh4cnczF1m5RRS
                      E8jguueG4=",
    "hash": "9UmFDLT4jzzfTMZv/5O71Bh73KTlrOTQXKgKYNC/Z0Y="
```
Listing 13: Example a downloadable JSON file

The JSON file is downloadable using a Blob object [?].

4.3.2 Verification of the signature

Having the possibility to sign a file, it seems natural to propose the verification of the latter.
The user needs to submit a JSON file in the same format as the JSON file downloadable as previously presented. The website have the ability to perform two verifications.
First it verifies if the hash of the file is the same as the hash on the JSON file. Second if the signature is correct knowing the hash and the aggregate-key. The hash is the same as in the first verification and the aggregate-key is collected from the JSON file.

The UI section is completed following the same procedure as in the section: "Send a file for a collective signature".


Verification.

Submit your file: No file chosen

Submit your signature file: No file chosen

The hash of the file and the hash registred in the signature file will be compared. Next the signature will be verified using the hash and the aggregate key.

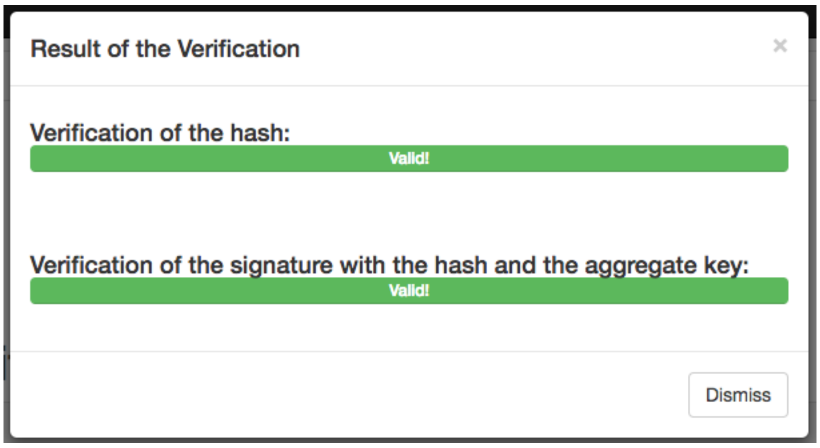
Figure 2: Interface

The submit of the two files is done in the same way as in the "Send a file for a collective signature" part using Promise APIs, a generator function and "runGenerator()" function.

The program translates the JSON file into an object due to the native function "JSON.parse()".
It calculates the SHA-256 hash of the submitted file using the same method as in the section: "Send a file for a collective signature" from the js-nacl library. Next the hash is translated in base64 and the comparison is done character by character.

The verification of the signature is accomplished using the function from the js-nacl library "nacl.crypto_sign_verify_detached".
Careful, the function is experimental but it passed all the tests done during the development.

The function takes as parameters the signature, the file's hash and the aggregate-key found in the JSON file. The function returns a boolean depending on the result of the verification.


Result of the Verification

Verification of the hash:

Valid!

Verification of the signature with the hash and the aggregate key:

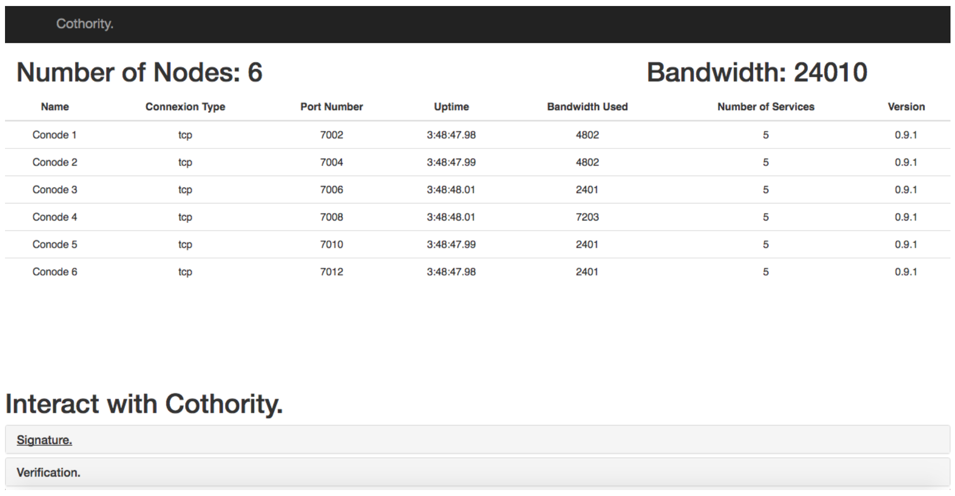
Valid!

Figure 3: Modal box displaying the result of the verifications

The result of the two verifications is displayed in a modal box.

5 Results

The final product looks like the image below.



Cothority.

Number of Nodes: 6

Bandwidth: 24010

Name	Connexion Type	Port Number	Uptime	Bandwidth Used	Number of Services	Version
Conode 1	tcp	7002	3:48:47.98	4802	5	0.9.1
Conode 2	tcp	7004	3:48:47.99	4802	5	0.9.1
Conode 3	tcp	7006	3:48:48.01	2401	5	0.9.1
Conode 4	tcp	7008	3:48:48.01	7203	5	0.9.1
Conode 5	tcp	7010	3:48:47.99	2401	5	0.9.1
Conode 6	tcp	7012	3:48:47.98	2401	5	0.9.1

Interact with Cothority.

Signature.

Verification.

Figure 4: Web-interface

The status table is refreshed every 3 seconds.

The website is compatible with Google Chrome and Firefox. It works with Safari but the browser doesn't let download directly the Blob object on the user's computer. It opens the Blob object in the browser. The website was not tested on Opera, Microsoft Edge and other web browsers.

All the libraries are in the project folder. This may seem a bit cumbersome but it allows more independence and privacy.

6 Theoritical and practical limitations of the project

The website contacts only conodes having their addresses in a static list (as seen in subsection 4.2 Asynchronicity). It is an inconvenient limitation. The implementation of the features presented lets no remaining time to tackle this problem.
At each instantiation of the js-nacl library, the heap is 32 megabytes in size. Therefore the user can submit a file of size at most ±8.5 megabytes. Certainly this number is not correct for each computer and each browser but it

gives a useful hint.

7 Future work

As written in above section the website contacts only 6 conodes. The next feature to bring is a recursive function. It will ask each conode to provide a list of each conode that it knows. Next the program contacts the conodes of the list and so forth.
It will be necessary to be careful to stop the recursive function at the right time and to look out conodes that the recursive function as already contacted.
This new function will also give work on the Cothority project side. When the website contact a conode it should be able to return the list of known conodes.

A website is always in development then any new idea can be added.

8 Theoritical and practical knowledge gained through the project

Before this semester project I had never particed HTML, CSS and JavaScript. This project gave me the opportunity to learn these important languages. The JavaScript language is one of the most used language at the moment. This practical part of web development brought me a theoritical knowledge too. In how a browser and a website operate. The particularity of single threaded programmation.

On the cryptographic part, I read some articles on decentralized cryptography. I understood the importance in the fields of security, integrity and transparency (among others) of decentralized cryptography. With this semester project in DEDIS laboratory, I was lucky to be able to see it at work through the Cothority project.
I was able to discover the surface of the elliptic curve signature scheme EdDSA and one occurence of it: Ed25519, which was useful to tackle the problem of calculating the aggregate-key.

9 Step-by-step guide for installation and running of the final product

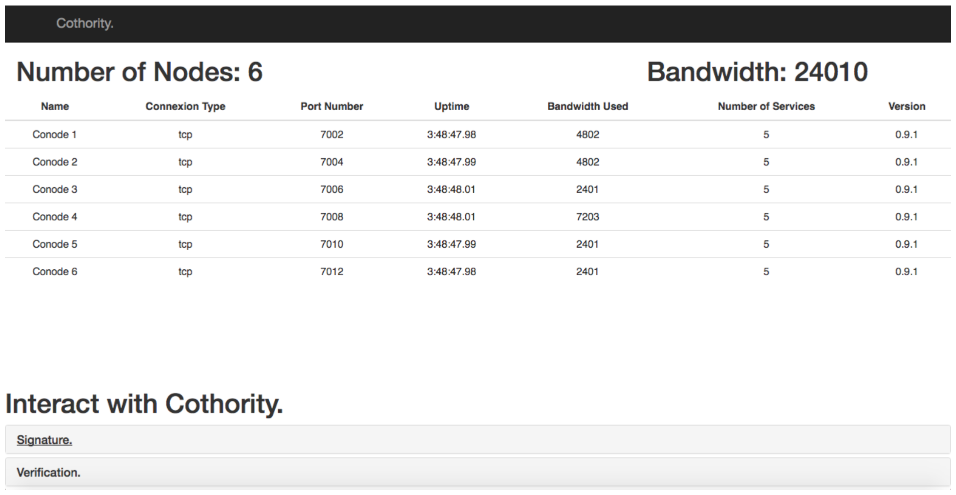
9.1 Installation

It suffices to open the file “cothorityweb.html” in a web browser. The best choice, as written before, is Google Chrome.
In the terminal it is important to run a script, which can be found in the folder: cothority/app/cothorityd/
It is named: “run_cothority.sh”. To run it it suffices to enter: “./run_cothority.sh” in the terminal.

Then the website comes alive.

9.2 Running

The website is straightforward to use.



Number of Nodes: 6

Bandwidth: 24010

Name	Connexion Type	Port Number	Uptime	Bandwidth Used	Number of Services	Version
Conode 1	tcp	7002	3:48:47.98	4802	5	0.9.1
Conode 2	tcp	7004	3:48:47.99	4802	5	0.9.1
Conode 3	tcp	7006	3:48:48.01	2401	5	0.9.1
Conode 4	tcp	7008	3:48:48.01	7203	5	0.9.1
Conode 5	tcp	7010	3:48:47.99	2401	5	0.9.1
Conode 6	tcp	7012	3:48:47.98	2401	5	0.9.1

Interact with Cothority.

Figure 5: Web-interface

The upper part of the web page is dedicated to display the number of conodes, the bandwidth used by all the conodes and a table containing some information for each conode.

The bottom part of the web page is consacred to the Signature and Verification parts of the website.

In clicking on “Signature.” a collapse opens and explain how to proceed to sign a certain file. It only needs to submit a file. After the website has signed it, a download button appears. It suffices to click on it to load the JSON file, containing the signature, on the computer.

In clicking on “Verification.” a collapse opens and explain how to proceed to verify a signature. It only needs to submit the file and the JSON file containing the file’s signature. The JSON file must be in the same format as the JSON file downloadable in the Signature part.
The website displays a monad presenting the result of the verifications. The first if the file’s hash and the hash enclosed in the JSON file are the same. The second if the signature is a correct one knowing the file’s hash and the aggregate-key contained in the JSON file.

References