Web-Frontend for Cothority

Bastian Nanchen

School of Computer and Communication Sciences

Decentralized and Distributed Systems lab

Semester Project

December 2016

Responsible	**Supervisor**
Prof.Bryan Ford	Linus Gasser
EPFL / DEDIS	EPFL / DEDIS

Contents

Chapter 1

Introduction

Distributed cryptography spreads the operation of a cryptosystem among a group of servers in a fault-tolerant way [2].
The DeDis team at EPFL is working among others on a software project called "Cothority". Cothority (Decentralized Withness Cosigning) is a "multi-party cryptographic signatures" [6].
In order to accomplish that they developed the CoSi protocol (Collective Signing), which will produce a collective signature by decentralized servers. 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. Per example a certificate or a software update now needs one single signature from a corporate 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 at every authoritative statement a validation, which is produced by a group of independent parties (named conodes), before any device or client use it [10].

Chapter 2

Aims and goals

The goal of this semester project is to furnish a web-interface to the Cothority project.
The aims stated are to provide a status-array with informations like port number, name or bandwidth used for each contacted conodes, to be able to send a file for a digital signature using the CoSi protocol and to verify if a digital signature corresponds to a particular file.

Chapter 3

Tools utilised

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 [1] is employed for designing the website.

3.1 Libraries

Several libraries are used in the semester project.
The jQuery library [9] is utilized to facilitate the selection and modification of DOM (Document Object Model) elements.

The protobuf.js [19] is a pure JavaScript implementation of Google's Protobuf. It uses the same format of .proto file. The Cothority's approach for serializing structured data is a Google's Protobuf-like. 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." [5].

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

Listing 3.1: 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" [5]. Each pair has a type. There is multiple disponible tags.
The last element to define is 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 [8] 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,... [21]. Its goal is to "provide all of the core operations needed to build higher-level cryptographic tools" [21]. Little disclaimer NaCl is pronounced "salt".

Another JavaScript NaCl library TweetNaCl.js [3] is used because the js-nacl doesn't implement two essential functions. More will be tell when the time comes.

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

Chapter 4

Problems arose and their solutions

In the following sections the problems arose and their solutions are treated by in which part of the development they showed up.

4.1 Communication client/server

The first problem to tackle was to create a communication between the website and a conode.
The object Websocket [14] 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 [12] object containing the .proto file in bytes.

```
message Request {
}
```

Listing 4.1: 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 4.2: response protocol buffer message

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 corresponding 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 be waiting on it and doesn't execute the remaining part of the program. In that case JavaScript program would be very slow. 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 (err, values) {
        if (err) {
          console.log('Error identifying file size: ' + err)
        } else {
          console.log(filename + ' : ' + values)
          aspect = (values.width / values.height)
          widths.forEach(function (width, widthIndex) {
            height = Math.round(width / aspect)
            console.log('resizing ' + filename + 'to ' + height + 'x' + heigh
            this.resize(width, height).write(dest + 'w' + width + '_' + filena
              if (err) console.log('Error writing file: ' + err)
            })
          }.bind(this))
        }
      })
    })
  }
})
```

Listing 4.3: 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 [13] natively in JavaScript [7]. 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 4.4: 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 4.5: 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." [16]. 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 4.6: 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 websocket_status(7003);
    listNodes.push(nodeCreation(message));
    // some code
}
```

Listing 4.7: Extract from the project’s code

The function “websocket_status()” is the one returning the Promise object. The generator function pauses when reaching “websocket_status()”. “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.” [17]. 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 4.8: Extract from the project’s code [17]

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 not, 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, has been presented. It only remains to do it for each conodes (here in port 7003, 7005, 7007).

```
runGenerator(function* generator() {
     listNodes = [];
     let message = yield websocket_status(7003);
     listNodes.push(nodeCreation(message));
     message = yield websocket_status(7005);
     listNodes.push(nodeCreation(message));
     message = yield  websocket_status(7007);
     listNodes.push(nodeCreation(message));
     // some code
   });
```

Listing 4.9: Extract from the project’s code reaching conodes at port 7003, 7005 and 7007

As said before the “websocket_status()” function returns a Promis object. Thus the generator function “generator()” will halt (“yield”) before each “websocket_status()” 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 Verification Part

The following problem to tackle is the implementation 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” [4].
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.” [15]. 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 [20].

4.3.1 Send a file for a collective signature

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 needs an input tag to submit a file and all the collapse-elements are managed by the Bootstrap framework [1].

Figure 4.1: Interface

The file submitted is read as an ArrayBuffer [11]. 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 inner 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;
    required bytes aggregate = 3;
}
```

Listing 4.10: .proto file

The calculation of the aggregate-key shall be done in the website for security reason. The server could make up a pair of public and private-keys, sign with it and send 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 [18]. 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” [3] 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 a JavaScript Uint8Array object into point x, y, z, t and multiply 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()]; // zero-point
  unpackneg(pubPos, pub);
  if (index === 0) {
    agg = pubPos;
  } else {
    add(agg, pubPos);
  }
  // some code
});
pack(aggKey, agg);
```

Listing 4.11: 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] $\hat{=}$ 128” is a multiplication of the x-axis by -1, due to the fact that the TweetNaCl.js has not an “unpack” function but only an “unpackneg” function as said before.
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. 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 Array-Buffer, the filename and an array message containing the signature and the aggregate-key. The function 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 to be more readable.

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/XA6flV2tLsiQmrAf4=",
"aggregate-key": "IjgFxLpeV8IOVShIGC6ESh4cnczF1m5RRSE8jguueG4=",
"hash": "9UmFDLT4jzzfTMZv/5O71Bh73KTlrOTQXKgKYNC/Z0Y="
```


Listing 4.12: Example a downloadable JSON file

The JSON file is downloadable using a Blob object [12].

4.3.2 Verification of the signature

The user needs to submit a JSON file in the same form as the JSON file downloadable as previously presented. The website have the ability to perform two verifications.
First 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 behavior as in the section: "Send a file for a collective signature".

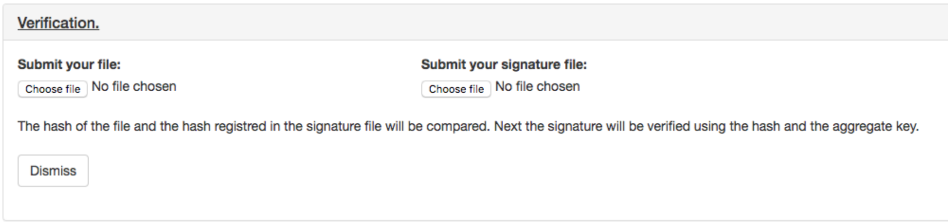

Figure 4.2: Interface

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()”.
The program 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.

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

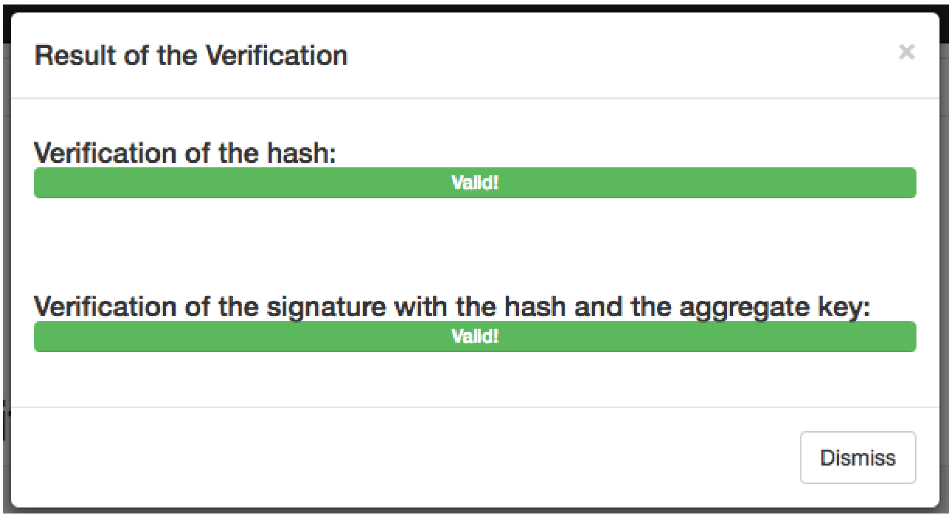

Figure 4.3: Modal box displaying the result of the verifications

Chapter 5

Results

The final product looks like tha image below.

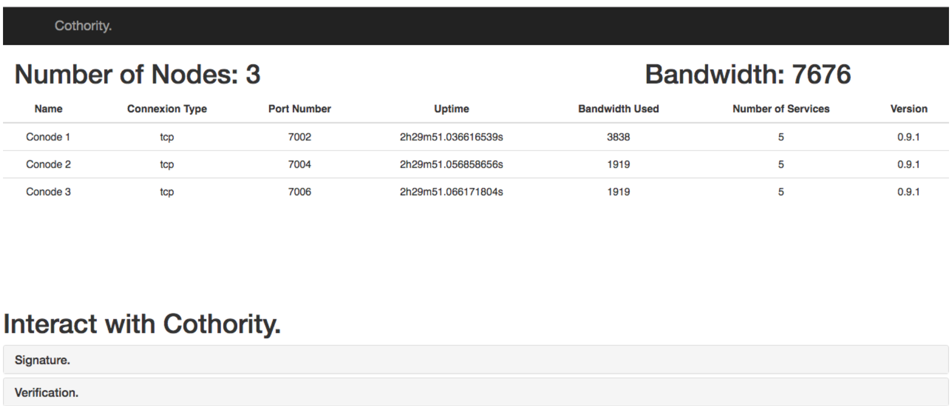

Figure 5.1: Web-interface

The status table is refreshed each 3 seconds. The website is compatible with Google Chrome, Safari and Firefox (not tested on Opera and Edge).

Bibliography

[1] Bootstrap. Bootstrap framework. URL: http://getbootstrap.com.

[2] Christian Cachin. *Security and Fault-tolerance in Distributed Systems*. IBM Research - Zurich, 2012.

[3] Dmitry Chestnykh and Contributors. Tweetnacl.js. URL: https://github.com/dchest/tweetnacl-js.

[4] Wikipedia Contributors. Schnorr signature. URL: https://en.wikipedia.org/wiki/Schnorr_signature.

[5] Google Developers. Protocol buffers guideline. URL: https://developers.google.com/protocol-buffers/docs/overview.

[6] Cory Doctorow. Using distributed code-signatures to make it much harder to order secret backdoors. URL: http://boingboing.net/2016/03/10/using-distributed-code-signatu.html.

[7] exploringjs. 25. promises for asynchronous programming. URL: http://exploringjs.com/es6/ch_promises.html.

[8] Tony Garnock-Jones and Contributors. js-nacl. URL: https://github.com/tonyg/js-nacl.

[9] The jQuery Foundation. jquery. URL: http://jquery.com.

[10] Ewa Syta Iulia Tamas Dylan Visher David Isaac Wolinsky Philipp Jovanovic Linus Gasser Nicolas Gailly Ismail Khoffi and Bryan Ford. Keeping authorities Honest or Bust with decentralized witness cosigning. URL: http://dedis.cs.yale.edu/dissent/papers/witness-abs.

[11] Mozilla Developer Network. Arraybuffer. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/ArrayBuffer.

[12] Mozilla Developer Network. Blob. URL: https://developer.mozilla.org/en-US/docs/Web/API/Blob.

[13] Mozilla Developer Network. Promise. URL: https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Promise.

[14] Mozilla Developer Network. Websocket object. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebSocket.

[15] N. Moeller S. Josefsson, SJD AB. eddsa and ed25519 draft-josefsson-eddsa-ed25519-02. URL: https://tools.ietf.org/html/draft-josefsson-eddsa-ed25519-02#page-6.

[16] Kyle Simpson. The basics of es6 generators. URL: https://davidwalsh.name/es6-generators.

[17] Kyle Simpson. Going async with es6 generators. URL: https://davidwalsh.name/async-generators.

[18] Jonathan Snook. Global variables in javascript. URL: https://snook.ca/archives/javascript/global_variable.

[19] Daniel Wirtz and Contributors. protobuf.js. URL: https://github.com/dcodeIO/protobuf.js.

[20] Daniel J. Bernstein Niels Duif Tanja Lange Peter Schwabe Bo-Yin Yang. Ed25519: high-speed high-security signatures. URL: https://ed25519.cr.yp.to.

[21] Daniel J. Bernstein Niels Duif Tanja Lange Peter Schwabe Bo-Yin Yang. nacl. URL: \url{http://nacl.cr.yp.to}.