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

Chapter 1

Introduction

1.1 History

Distributed cryptography spreads the operation of a cryptosystem among a group of servers in a fault-tolerant way [2].
The DEDIS lab at EPFL created the Cothority project, which implements decentralized and distributed cryptographic protocols.

TODO:presentation of the report!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

1.2 Aims and Goals

The goal of this semester project is to furnish a web-interface to the Cothority project.
The aims stated at a summer meeting with Mr.Linus Gasser are to create an interface with status' conodes, be able to send a file for a collective signature and to verify a signature.

Chapter 2

Analysis

The first part of the semester project was to acquire knowledge in JavaScript and HTML and become familiar with the JQuery library. For delivering a skeleton of the website before the beginning of the semester.

Next thing to settle was the communication between the website and a conode. The JavaScript Web APIs contain all the necessary to handle this problem. The object Websocket [11] offers all the tools to create a communication between a browser and a server.

TODO:presentation of the .proto files!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

The Cothority's approach for serializing structured data is a Google's Protobuf-like. As said on the website of Google's Protobuf:"Protocol buffers are a flexible, efficient, automated mechanism for serializing structured data - think XML, but smaller, faster, and simpler." [5].

Taking this into account and knowing that Google's Protobuf doesn't support generated code in JavaScript, the choice was made on the library protobuf.js [16]. It is a pure JavaScript implementation of Google's Protobuf. It uses the same format of .proto file.

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

Listing 2.1: example of .proto file

2.1 Status part

The JavaScript Web APIs contain all the necessary to handle the communication part. The object Websocket [11] offers all the tools to create a communication between a browser and a server and send/receive data. All the elements send to conodes are .proto files.

First it is necessary to establish the connection. A connection is established with each conode.
An empty .proto file is send in a Blob object containing the .proto file in bytes.

```
message Request {
}
```

<div align="center">Listing 2.2: empty .proto file</div>

The request being sent the webpage waits for a response. The response will be the status of the concerning conode. It is received as a .proto file.

```
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;
                                }
                                }
```

<div align="center">Listing 2.3: response .proto file</div>

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,...) and each field to its value.

2.1.1   Promise API

That being said, 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 Promise object in JavaScript must 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 code. In that case JavaScript program would be very slow. So through the

history of JavaScript many tools were created to handle asynchronous programs. Like callback functions that was 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 2.4: Example of Callback Hell with its typical pyramid shape

Other libraries (jQuery) begun implementing promises to help developers overcome the "Callback Hell". This eventually leads ECMAScript 6 to adopt The Promise API [10] natively in JavaScript [6]. 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 2.5: Structure of a Promise

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

2.1.2   Generator

To handle the Promise returned the project will use another feature of EC-MAScript 6 called a Generator. It's a new kind of function. It can be paused in the middle and resumed later. Of course, other parts of the code are running during the pause.

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

Listing 2.6: Structure of a generator function

The "*" marks the function as a generator one. The other different element of the code snippet is the keyword:"yield". "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." [13]. The function will stop when it encounters the "yield" keyword. Whenever (if ever) the generator is restarted, the "yield 1" expression will send "1" back. The part following the "yield" keyword is the expression that will be returned.

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

Listing 2.7: Restart of a generator function

The call to the "next()" method will execute the generator (from the last "yield") to the next encounter of a "keyword" (if there is any).

Returning to the project's code, a generator function is used with the function returning a Promise object.

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

Listing 2.8: Extract from the project's code

The function "websocket_status()" returns a Promise object. The generator function stops 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." [14]. 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 2.9: Extract from the project's code [14]

This function takes a generator function as parameter. The "iterator" function is the generator function. The inner-function "iterate()" will look if a promise returns the value "message" from "iterator.next(message)". If not ("ret.done" == false), the function will iterate again.

2.1.3 Implementation

Now all the elements needed are presented, it only remains to do it for each node.

```
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 2.10: Extract from the project's code reaching conodes at port 7003, 7005 and 7007

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

Then it only remains to do it for each node.

2.2   Signature part

The semester project must implement 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." [12]. 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 [17].

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

2.2.1 Send a file for a collective signature

Analysis

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. To accomplish it the program needs to collect the public-key of each servers. Then given that the Cothority project uses Ed25519

Implementation

The implementation of the HTML part is straightforward. It needs an input tag to submit a file and all the collapse-thing is managed by the Bootstrap framework [1].
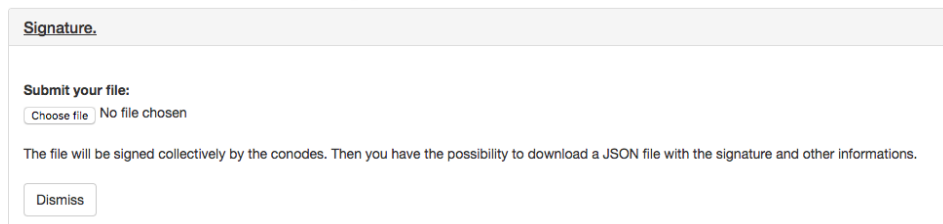

Figure 2.1: Interface

The file submitted is read as an ArrayBuffer [8]. 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 (go to subsection nÂř2.1 for more details on the manner to tackle the asynchronous part).

Afterward the file needs to be signed. To do so the program uses two libraries protobuf.js [16] that was introduced before and js-nacl [7]. 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,... [18]. Its goal is to "provide all of the core operations needed to build higher-level cryptographic tools" [18]. Little disclaimer NaCl is pronounced "salt".

In this semester project the js-nacl library is employed to generate a SHA-256 of the file and to verify the signature knowing the hash of the file and the aggregate-key.

When the websocket is appropriately opened, the communication is done through a websocket using 4 proto file 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 2.11: .proto file

As said in the Analysis subsection the website needs to calculate the aggregate-key. 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 [15]. 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 an other NaCl library: "TweetNaCl.js" [3] to pack and unpack and to addition the points.

```
const listServers = window.listNodes.map(function(node, index) {
            const server = node.server;
            const pub = new Uint8Array(server.public.toArrayBuffer()); //
            // the point is represented as a 2-dimensional array
            const pubNeg = [gf(), gf(), gf(), gf()]; // zero-point
            unpackneg(pubNeg, pub);
            const pubPosArr = new Uint8Array(32);
            pack(pubPosArr, pubNeg);
            const pubPos = [gf(), gf(), gf(), gf()]; // zero-point
            unpackneg(pubPos, pubPosArr);
            if (index == 0) {
                agg = pubPos;
            } else {
                // add pubPos to agg, storing result in agg
                add(agg, pubPos);
            }
            return new siProto({public: server.public, id: server.id, add
                description: server.description});
        });
        pack(aggKey, agg);
```
Listing 2.12: Extract of the code calculating the aggregate-key

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 ArrayBuffer, 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+BVQQx0EAH3XFLClfwSbskCxEsPIJ62Z
"aggregate-key": "IjgFxLpeV8IOVShIGC6ESh4cnczF1m5RRSE8jguueG4=",
"hash": "9UmFDLT4jzzfTMZv/5O71Bh73KTlrOTQXKgKYNC/Z0Y="
```
<div align="center">Listing 2.13: Example a downloadable JSON file</div>

The JSON file is downloadable using a Blob object [9].

2.2.2 Verification

Analysis

The user needs to submit a JSON file in the same form as the JSON file downloadable as previously said. The website have the ability to verify two things. First if the hash of the file is the same as the hash on the JSON file. Second if the signature is correct.
To accomplish the first part, the program extracts the hash from the JSON file and calculates the SHA-256 hash of the submitted file and compare the two of them.
The second part,

Implementation

The UI section is completed following the same behavior as before.

<div align="center">Figure 2.2: Interface</div>

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 subsection: "Send a file for a collective signature"
from the js-nacl [7] 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 "nacl.crypto_sign_verify_detached"
from the js-nacl [7] library. Caution 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.

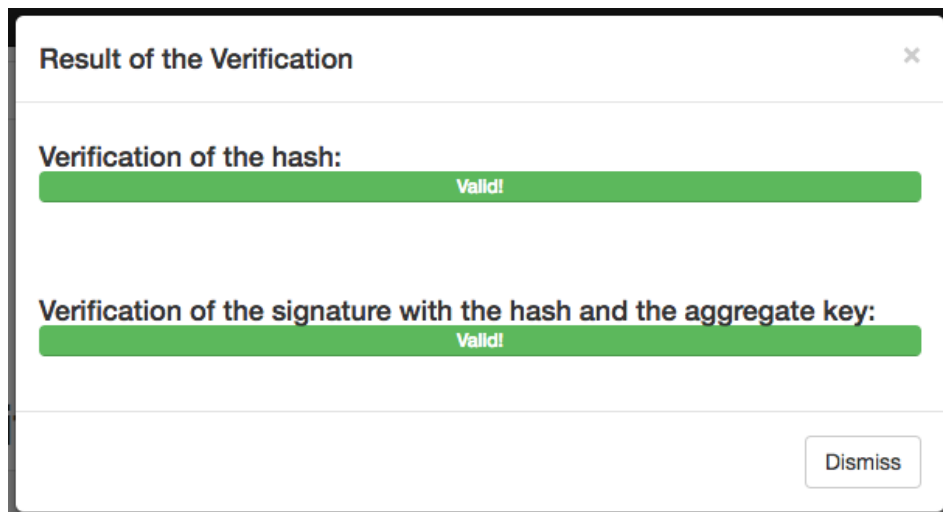The result of the two verifications is displayed in a modal box.

Figure 2.3: Modal box displaying the result of the verifications

2.3   Conclusion

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

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

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

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

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

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

[12] 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`.

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

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

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

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

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

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