Secure P2P data transfer

Using WebRTC

BACHELOR THESIS

GUILLAUME BONVIN

December 2020

Thesis supervisors :

Prof. Dr. Jacques PASQUIER–ROCHA
Software Engineering Group

Graduate assistant Arnaud DURAND
Software Engineering Group

UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Software Engineering Group
Department of Informatics
University of Fribourg
(Switzerland)

Preamble

Acknowledgements

I want to express my special thanks to Arnaud Durand for his continuous support throughout this project and the many help he provided me during our discussions.

I also want to thank Prof. Dr. Jacques Pasquier for the supervision of this thesis as well as the numerous skills he has passed on to me through the various courses he gave since I joined the university.

Notations and Conventions

Throughout this document, the same example of file transfer will be used. By convention, the sender will be called Alice and the receiver Bob. Their actions on the different figures will always be coloured blue for Alice and pink for Bob. The steps requiring a Signaling server will be displayed in yellow.

All code-related vocabulary like variable names, object types, file names, functions or methods, fields and so on will be displayed in **bold**.

This document comports a lot of references to STUN servers, TURN servers, and Signaling server. From an architectural point of view, those are normally separated. In this project though, they all run on the same VPS and should therefore rather be considered as distinct services offered by the same server, even though they will be referred as servers in this document.

Table of Contents

List of Figures

List of Tables

List of Source Code

1 Introduction

1.1 Project description

The goal of this project is to create a simple command-line application which allows for two users to safely exchange files of any kind or size. The final program is launched from any operating system through its command-line interface. It follows a rather intuitive and interactive style, asking the user for input such as the name of the file or its location, guiding him through all the steps in a verbose and accessible way.

Unlike classical file exchange services like Dropbox or Google Drive, this application uses no relay server to get the files to the recipient. Instead of a classical upload-download scheme, data will be sent over via a direct channel to the receiver's computer. Such channel creation can be achieved using the WebRTC technology.

A big part of the project is the focus on security. All files are exchanged over a direct channel between the users, going through no third-party in-between, taking down the risk of a man-in-the-middle attack. The critical part here is the mutual authentication between sender and receiver needed to initialise the connection. As soon as both users can authenticate each other, the resulting channel is considered safe.

1.2 Use case and demonstration

To get a good grasp at the application features and usability, let us set a basic scenario where Alice wants to send a picture to Bob. The complete protocol is discussed in the third chapter, about the application structure.

1.2.1 Sender side

From Alice's perspective, she first launches the program and indicates she wants to send a file.

```
Welcome aboard, cabin boy !
Are you sender or receiver of the file? ('s'/'r')
You can stop the program by typing quit ('q')
send
```

Figure 1: Alice - role picking

She is now given a passphrase and asked for the receiver's passphrase.

```
Preparing to send...
Your passphrase is: woman-queen-mist-balance-floor
Enter your receiver's passphrase:
event-stranger-airplane-linen-jellyfish
```

Figure 2: Alice - remote passphrase input

Once the remote user is found, she can indicate her file name ad location.

```
Receiver identity confirmed!
Receiver is ready for a file offer, please enter file path and name:
Example - somefolder/image.png
in/pic.png
```

Figure 3: Alice - file selection

The transfer begins as soon as Bob accepts the file offer.

```
File ready, waiting on answer...
```

```
File offer accepted! Your file is being sent...
|||| -->Upload done
Waiting for confirmation...
```

Figure 4: Alice - file successfully sent

Once the file is properly received, Alice gets a confirmation and is then given the possibility to send another file.

```
File has been received successfully!
Send another file ? ('y'/'n')
yes
```

Figure 5: Alice - integrity confirmed

1.2.2 Receiver side

From Bob's perspective, the process is similar, he first choses to be receiver, and enters Alice's passphrase.

```
Welcome aboard, cabin boy !
Are you sender or receiver of the file? ('s'/'r')
You can stop the program by typing quit ('q')
r
Preparing to receive...
Your passphrase is: event-stranger-airplane-linen-jellyfish
Enter your sender's passphrase:
woman-queen-mist-balance-floor
```

Figure 6: Bob - role picking and remote passphrase input

When the file offer is received, he gets a quick look at the file info and is given the choice to accept or reject it.

```
Received a file offer:
Name: pic.png
Size: 222028 byte
Type 'yes' to accept offer:
yes
```

Figure 7: Bob - file offer accepted

Soon as the file is received, and its integrity is confirmed, it is saved in Bob's default output directory and a confirmation is sent to Alice. Bob is now waiting for another file offer.

```
|||| -->Download done
File integrity confirmed! Saving file...
```

Figure 8: Bob - file received and integrity confirmed

We notice both users are given a passphrase and asked for the remote one. This is used to mutually authenticate each other and will be discussed in detail in the security chapter (4.1).

1.3 Thesis outline

This document has the purpose of describing the needed steps and the different challenges we can meet while implementing such an application.

We will follow a guideline using this logic: starting from an overall view of the technologies involved and going progressively into the actual implementation of the different aspects, only to conclude on the real concrete and precise problem of mutual authentication between the peers.

The next chapter, WebRTC Architecture, has the purpose of getting an overall look at the technology, the base block of this project. We will see the logic behind RTC, the protocols involved as well as the components we need to setup.

In chapter 3, Implementation, we will get closer to the actual code by first introducing Go, the programming language mostly used throughout this application and the corresponding library used for WebRTC. The deployment of the external components will be explained before jumping to the application structure, its main features, the protocols we defined as well as their implementation.

Finally, having a good understanding of the inner workings of the program, we will focus on a crucial part of the project: Security. This last chapter will describe the signaling process, and how to deal with the problematic of mutual authentication using certificates and passphrases.

2

WebRTC Architecture

2.1 WebRTC overview

WebRTC, which stands for Real Time Communication (RTC) is a set of standardized technology setup by the IETF (Internet Engineering Task Force). It offers web application developers a way to implement high quality real-time multimedia applications without the need for any external plugin. This technology is open source and does not need any third-party software. The source code can be found for free at http://www.webrtc.org/.

WebRTC performs especially well when it comes to media capture, video encoding and decoding as well as transport layer and session management. All these features allow for an easy way to setup a live video and chat webapp. You can try it out yourself on the following demo site: https://apprtc.appspot.com/.

The latest published version is described at https://www.w3.org/TR/webrtc/.

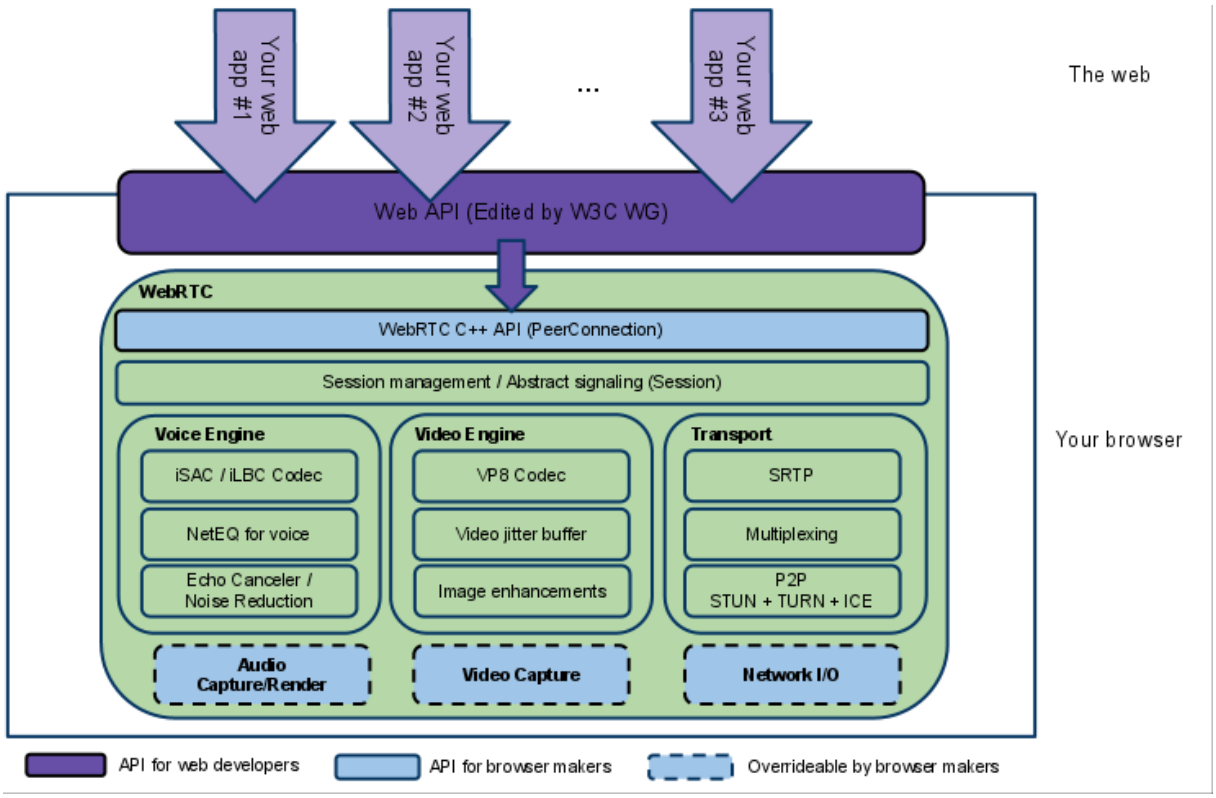

Figure 9: WebRTC overall architecture[1]

This diagram (see Fig. 9) shows us the two distinct layers WebRTC offers.

On one side, the C++ API is aimed at web browsers developers. Using this API and its different hooks like media capture and render, each browser has developed its own implementation of WebRTC. Nowadays, most desktop web browsers do fully support this technology.

On the other side, the part we are interested in is the Web API. This JavaScript API is developed by W3C. The various objects it offers will be seen in section 2.3: Introduction to the Web API.

[1] Figure taken from https://webrtc.github.io/webrtc-org/architecture/

2.2 Protocol stack

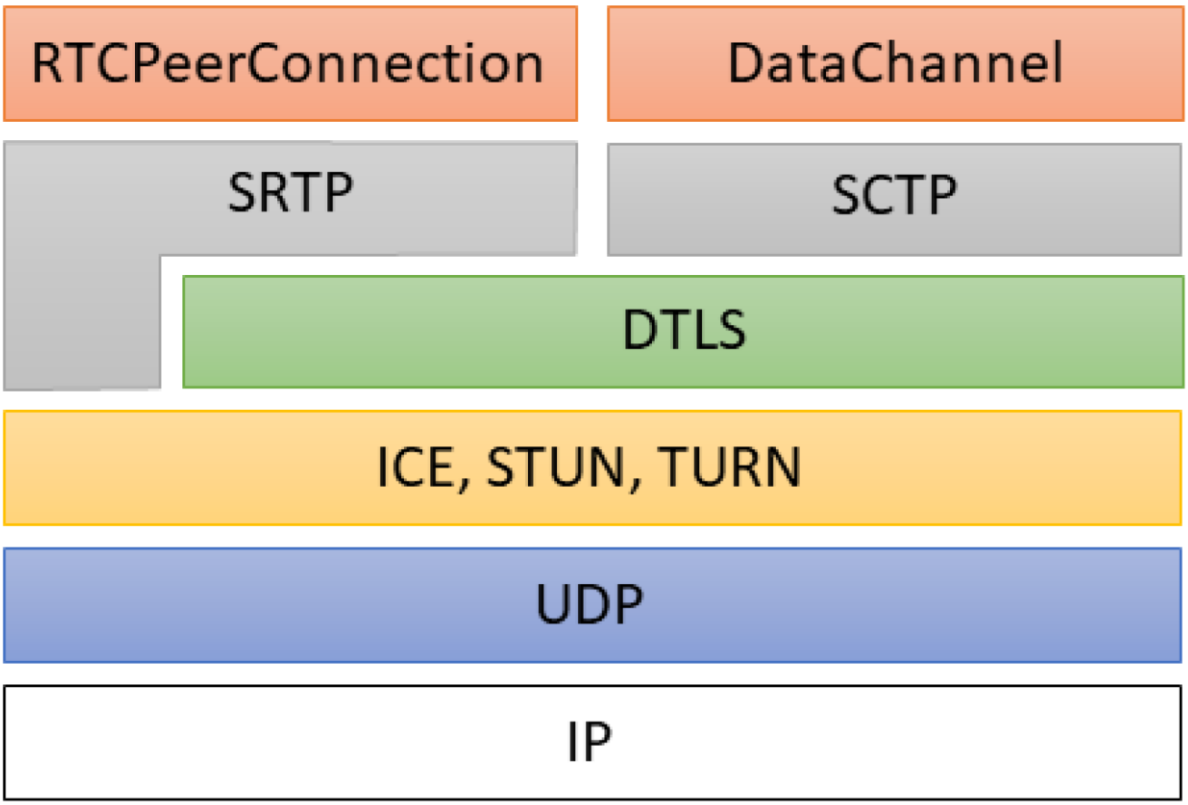

Figure 10: WebRTC protocol stack[2]

This graph (see Fig. 10) is a view of the protocol stack for web Real-Time Communication.

RTCPeerConnection and DataChannel are two APIs used by browsers to transmit their user's media, and respectively custom data. Medias are sent through Secure Real-Time Transfer Protocol (SRTP), while data are sent through Stream Control Transfer Protocol (SCTP).

Secure transaction between the peers comes with the underlying layer, Datagram Transport Layer Security (DTLS), a protocol based on TLS and responsible for the data encryption.

The next layer, Interactive Connectivity Establishment (ICE) is a protocol used to allow for user to establish a connection while being connected to the internet behind firewalls and Network Address Translators (NAT). More about ICE and STUN/TURN services is detailed in section 2.4.2: ICE protocol

Finally, WebRTC being aimed at real-time media exchange, and thus latency being more important than reliability, it relies on User Datagram Protocol (UDP) as the transport layer.

[2] Figure taken from https://www.mdpi.com/2079-9292/9/3/462/htm

2.3 Introduction to the Web API

The WebRTC JavaScript API is the part we will use in this project. It is made of several objects which help us establish real time connection between web browsers without having to deal with classical issues such as packet loss or temporary connection drop.

In this project, the API is used for its ability to create and handle peer-to-peer data channels. These channels let us send any kind of data outside of audio and video stream in a fast and secure way.

The main interfaces we will use are the following:

- RTCPeerConnection
    - It represents a connection between two peers. This will be used by the browser to transmit acquired audio and video from MediaStream or custom data from the RTCDataChannel
    - It is used with the dictionary RTCConfiguration, which provides options for configuring the PeerConnection, such as using a specific certificate.
- RTCSessionDescription
    - RTCSessionDescription is attached to the PeerConnection to represent its parameters. It consists of the description type, which indicates the side of the negociation process between offer and answer as well as a Session Description Protocol (SDP) containing all metadata about the client's media.
- RTCDataChannel
    - Once a PeerConnection is established, we can attach it this bidirectional channel to communicate any wanted data.

2.4 Overview of the components

WebRTC has a great level of complexity. In this chapter, we will look at the essential components needed for this project.

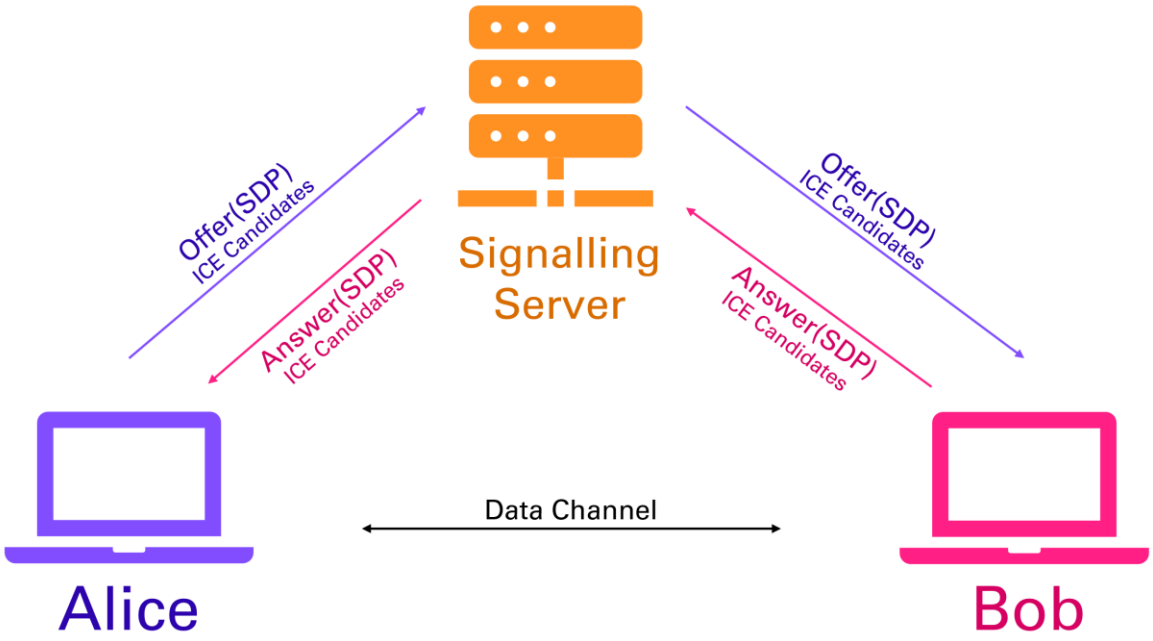

Figure 11: WebRTC architecture

The goal is to open a Data Channel between the peers, linking them directly without the need of any third-party. Information about the hardware as well as network information are needed for both clients to find each other and to know how to communicate together.

To first exchange these data, we will use a signaling server. Its role is to allow for two clients to exchange their SDP and ICE candidates. The SDP, standing for Session Description Protocol, contains all media specifications like resolution, codec, encryption, etc. The ICE candidates are information about the network connection. These candidates are gathered by each user by connecting to a STUN or TURN server.

Once all this data is acquired and exchanged via the signaling server, both clients can disconnect and communicate directly on their newly opened DataChannel.

We will now take a more detailed look at these components.

2.4.1 SDP

The Session Description Protocol takes a crucial part in this project. Not only does it contain metadata about the client's media, but it also contains a self-signed certificate, which we will use later on for the mutual authentication (4.1).

The following is the JSON 'SDP' field extracted from a PeerConnection offer.

```
v=0
o=- 1130848905439104866 1619480298 IN IP4 0.0.0.0
s=-
t=0 0
a=fingerprint:sha-256
BB:53:EA:A3:F6:66:90:CD:0E:80:2D:D3:9D:E9:3D:64:3A:0B:93:7F:E4:83:E0:1
7:89:76:8A:71:CC:98:46:9A
a=group:BUNDLE 0
m=application 9 DTLS/SCTP 5000
c=IN IP4 0.0.0.0
a=setup:actpass
a=mid:0
a=sendrecv
a=sctpmap:5000 webrtc-datachannel 1024
a=ice-ufrag:mQMPdwCzntgwSNjU
a=ice-pwd:QOKKNwfvqlBegNKkEPJMiElHCxJextFP
```

Complete and more detailed tables of SDP offer and answer can be found in section 4.2: Certificate and fingerprint

2.4.2 ICE protocol

The Interactive Connectivity Establishment is a protocol used to avoid the issues we could get while connecting two web browsers. It gives a way to work around firewalls, give public IPs when necessary or use an external server when the router does not allow direct connections.

2.4.3 STUN / TURN servers

STUN stands for Session Traversal Utilities for NAT. This protocol allows for a client to know its public address and to know about all the possible router restrictions which could make a direct connection impossible.

A client requesting a STUN server receives back its public address and its accessibility behind a NAT router.

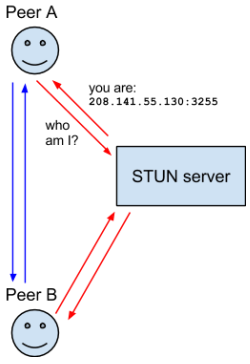

Figure 12: STUN server[3]

[3] Figure taken from https://developer.mozilla.org/fr/docs/Web/Guide/API/WebRTC/WebRTC_architecture

When a symmetric NAT restriction is used by a router, only known addresses will be allowed for a connection. To work around this restriction, we can use a TURN server (Traversal Using Relays around NAT). Its job is to act as a relay server between the peers, by transferring all data between two clients. This server defeats the base purpose of WebRTC and is used only when no alternative is possible.

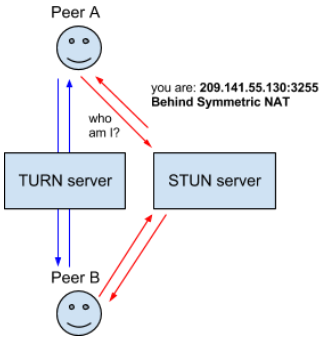

Figure 13: TURN server[4]

2.4.4 Signaling Server

To open a PeerConnection, all information mentioned before must be exchanged by the clients by using a signaling server. Its role is to match the users and allow them to exchange their offer and answer in a classical way. There is no predefined way to implement the signaling protocol as it solely depends on the application's purpose.

In this project, our signaling server does the following job: it registers all logged in users and checks for their respective sender or receiver. Once a match is made, it only acts as a relay server, transferring all messages between the peers. These messages contain the offer – made essentially from the SDP and the ICE candidates – and its corresponding answer, built the same way. As soon as all information is exchanged, both peers disconnect from the signaling server.

[4] Figure taken from https://developer.mozilla.org/fr/docs/Web/Guide/API/WebRTC/WebRTC_architecture

3 Implementation

3.1 Go language

All code written for the application itself is done using Go, often referred as Golang, due to the website name. Go is an open-source coding language developed by Google and continuously improved by the open-source community contributors.

Currently in its 1.15 version, its first release happened in March 2012, making it a fairly recent language.

Go can be described as a simplified version of C. It is a high-performance compiled language aimed at simplicity and designed for its ease of readability and understandability, making it friendly and efficient to learn for newcomers.

3.1.1 Why Go?

Using Go comes with many advantages. In our case, the first thing is the availability of the Pion WebRTC API[5]. This library gives us all the needed tools to work with WebRTC technology within Go through a wide set of objects like RTCPeerConnection, DataChannel, ICEConnection handler and even cryptography tools like fingerprint reader, certificate generator and cyphering. Those last elements will play a huge role in the security part we will describe later, which is a central point in our project.

Among the many benefits of using Go, its cross-platform hallmarks allow for easy code porting and this might be its greatest advantages. Building Go executables for multiple platforms can even be done without having access to the specific OS[6]. A wide range of OS and architectures are available for native application building.

Go comes with many other significant advantages. Here are some:

- Instant compilation to machine code
- Statically typed
- Efficient at concurrent and parallel processing
- Excellent cloud compatibility
- Built-in infrastructure to support testing

Overall, Go is a fast high-performance professional language with a clean syntax aimed at simplicity and used in a wide variety of projects.

3.1.2 Environment setup

To setup a Go environment, all we need is a text editor and the Go compiler. In our case, we additionally used JetBrains' GoLand IDE[7], which allows us to work in a more efficient way. The IDE helps with code writing trough auto-completion, errors highlight, testing, modules management, and way more. It makes for a faster way to cycle through the editing, compiling and testing loop, as all code can be executed and debugged separately from within the IDE.

Initially, Go had to be installed and worked with in a specific location designated by the GOPATH environment variable[8]. The code, for example **hello.go**, is compiled using

```
go build hello.go
```

which will create an executable file. The run command makes it easier and faster by directly compiling and executing the code, leaving no executable file afterward:

[6] [24]'How To Build Go Executables for Multiple Platforms on Ubuntu 16.04'. n.d. DigitalOcean. Accessed 12 February 2021. https://www.digitalocean.com/community/tutorials/how-to-build-go-executables-for-multiple-platforms-on-ubuntu-16-04.
[7] IDE available on https://www.jetbrains.com/go/
[8] [17]'Go - Environment Setup - Tutorialspoint'. n.d. Accessed 12 February 2021. https://www.tutorialspoint.com/go/go_environment.htm.

```
go run hello.go
```

Since 2018 though, we can now use Go Modules to manage our environment[9].

```
go mod init
```

will generate a go.mod file. Placed at the root of our program tree, it tells Go to consider all files within the hierarchy as a module that can be run on its own. Obviously, all dependencies can stay outside the module as they are specifically listed in the go.mod file.

3.2 External Components

Before jumping to the actual application code, we still need to define how our signaling and TURN services are deployed and how they can be reached.

In this section, we will have a brief look at these external component's implementation. The signaling server as well as STUN and TURN servers all run on the same VPS and can be reached at flying-dut.ch.

3.2.1 TURN

While some free STUN servers are available online, it is much harder to find TURN servers because of the potential large amount of data they may have to deal with. For this reason, we chose to deploy a TURN service in our own Linux server, using Coturn. Coturn project is a free open-source implementation of a STUN/TURN server. Originally from the Google Code archive[10], it has since moved to GitHub[11] and its development is still going on as Coturn Project[12].

The first step is to configure a base Ubuntu 18.04 server. The OS is installed on our personal VPS, at vps718907.ovh.net. To secure the access, an SSH key authentication is set.

The configuration is relatively simple. For this project, the whole process was done following an article from ourcodeworld.com[13]. We first install Coturn using:

```
sudo apt-get install coturn
```

To enable the TURN server, we modify the configuration file located in etc/default/coturn by adding the following line:

[9] [19]'Go - How Do I Configure Goland to Recognize "mod" Packages?' n.d. Stack Overflow. Accessed 2 September 2020. https://stackoverflow.com/questions/51910862/how-do-i-configure-goland-to-recognize-mod-packages.
[10] [21]'Google Code Archive - Long-Term Storage for Google Code Project Hosting.' n.d. Accessed 30 December 2020. https://code.google.com/archive/p/rfc5766-turn-server/.
[11] [7]*Coturn/Coturn*. (2015) 2020. C. coturn. https://github.com/coturn/coturn.
[12] [8]*Coturn/Rfc5766-Turn-Server*. (2015) 2020. C. coturn. https://github.com/coturn/rfc5766-turn-server.
[13] [25]'How to Create and Configure Your Own STUN/TURN Server with Coturn in Ubuntu 18.04 | Our Code World'. n.d. Accessed 2 September 2020. https://ourcodeworld.com/articles/read/1175/how-to-create-and-configure-your-own-stun-turn-server-with-coturn-in-ubuntu-18-04.

```
TURNSERVER_ENABLED=1
```

The configuration is done in etc/turnserver.conf file. Here, we define the following parameters:

- Listening port (UDP and TCP)
- TLS listening port
- Server name and realm
- Guest username and password
- SSL certificate and private key location

To make the servers reachable from an URL, the domain flying-dut.ch is acquired from Infomaniak.ch and the STUN and TURN services are linked by setting type A records from the DNS manager.

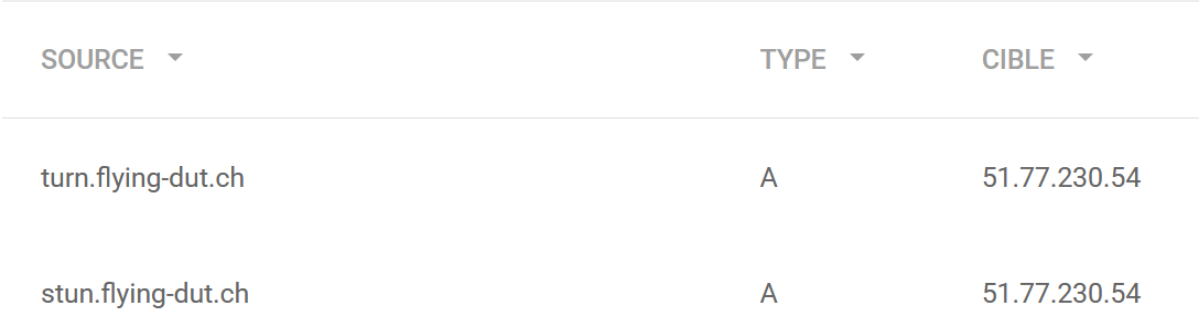

Figure 14: type A DNS records for STUN/TURN

STUN and TURN can now be reached via their respective subdomains: stun.flying-dut.ch and turn.flying-dut.ch.

The service is finally enabled with the following command:

```
systemctl start coturn
```

To test it, we use TrickleICE[14], an online tool which initiate a **RTCPeerConnection** with the given ICE server. By specifying our URI and credentials, the page initiate an ICE candidate gathering and displays all results.

Time	Component Type	Foundation	Protocol	Address	Port	Priority
0.015	1 host	0	udp	ddb787ff-7b42-49b6-b592-466b97918a59.local	55760	126 | 32512 | 255
0.017	1 host	4	tcp	ddb787ff-7b42-49b6-b592-466b97918a59.local	9	125 | 32704 | 255
0.018	2 host	0	udp	ddb787ff-7b42-49b6-b592-466b97918a59.local	55761	126 | 32512 | 254
0.019	2 host	4	tcp	ddb787ff-7b42-49b6-b592-466b97918a59.local	9	125 | 32704 | 254
0.042	1 srflx	1	udp	178.238.165.230	62917	100 | 32543 | 255
0.084	1 relay	3	udp	51.77.230.54	56166	5 | 32542 | 255
0.110	2 srflx	1	udp	178.238.165.230	62956	100 | 32543 | 254
0.290	2 relay	3	udp	51.77.230.54	55658	5 | 32542 | 254
0.291						Done

Figure 15: TrickleICE testing results

[14] [56]'Trickle ICE'. n.d. Accessed 2 September 2020. https://webrtc.github.io/samples/src/content/peerconnection/trickle-ice/.

This table (see Fig. 15) shows the result of testing both our services at once. A srflx type candidate means our STUN service is working, while a relay type candidate indicates a working TURN.

3.2.2 Signaling Server

The signaling server is the last external part we need for our application to work. Its role is to allow for two users to exchange the necessary information to establish a secure direct connection.

Each client will communicate with the server trough a WebSocket connection and all exchanged messages will have the JSON format. The service itself is setup using NodeJS[15] and its ws npm package[16].

Setting up this signaling service on our remote VPS is straightforward. NodeJS is installed via command-line, as well as the ws package both locally and on the remote server. For comfort reason, the actual code for the signaling protocol is written in the server.js file, which is frequently modified and tested locally, and uploaded onto the actual remote server using a basic SCP command. Locally, the service is run directly trough the GoLand IDE. On the remote machine, it is run with the following command:

```
node server
```

Just like for our STUN and TURN services, this signaling server must be reachable trough an address, which is why we setup another A record from our DNS manager, giving it the signal.flying-dut.ch subdomain.

signal.flying-dut.ch	A	51.77.230.54

Figure 16: Type A DNS record for signaling

The last thing we need is to setup a port. Within the server.js code, we optionally specify that we want the service to be run on port 9090.

```
var wss = new WebSocketServer({port: 9090});
```

The service can now be reached by the application with the following address: ws://flying-dut.ch:9090

[15] [40]Node.js. n.d. 'Download'. Node.Js. Accessed 10 February 2021. https://nodejs.org/en/download/.
[16] [62]'Ws'. n.d. Npm. Accessed 10 February 2021. https://www.npmjs.com/package/ws.

3.3 Application structure

3.3.1 Design choices

We saw it in the use case, the application is highly interactive and guided. It follows a straight path, giving the user simple choices throughout the process. This design makes for a great ease of use, but it comes with its counterparts. For the client, easy does not always mean efficient and some users may prefer a less intuitive single command program using flags for example. For us, this design means more ways for the user to input incorrect information, and a wider range of potential error situations that must all be considered and handled properly.

One of the crucial choices in the application design is the clear distinction between sender and receiver. This state is defined at the very start of the process and is kept for one exchange cycle at least. At this point of development, roles between the two users cannot be exchanged without going through the signaling process again. This clear distinction makes the code easier to read and edit, and the signaling and file exchange protocols way clearer. In its current state, impossibility to switch between roles decreases user efficiency and further development is needed to skip the signaling step in case of bidirectional file exchange.

3.3.2 Process

To make the individual code parts easier to understand, we first need to have a look at how the whole structure is behaving.

The following model (see Fig. 17) is a simplified representation of how the four main code files interact with each other during one complete file exchange cycle.

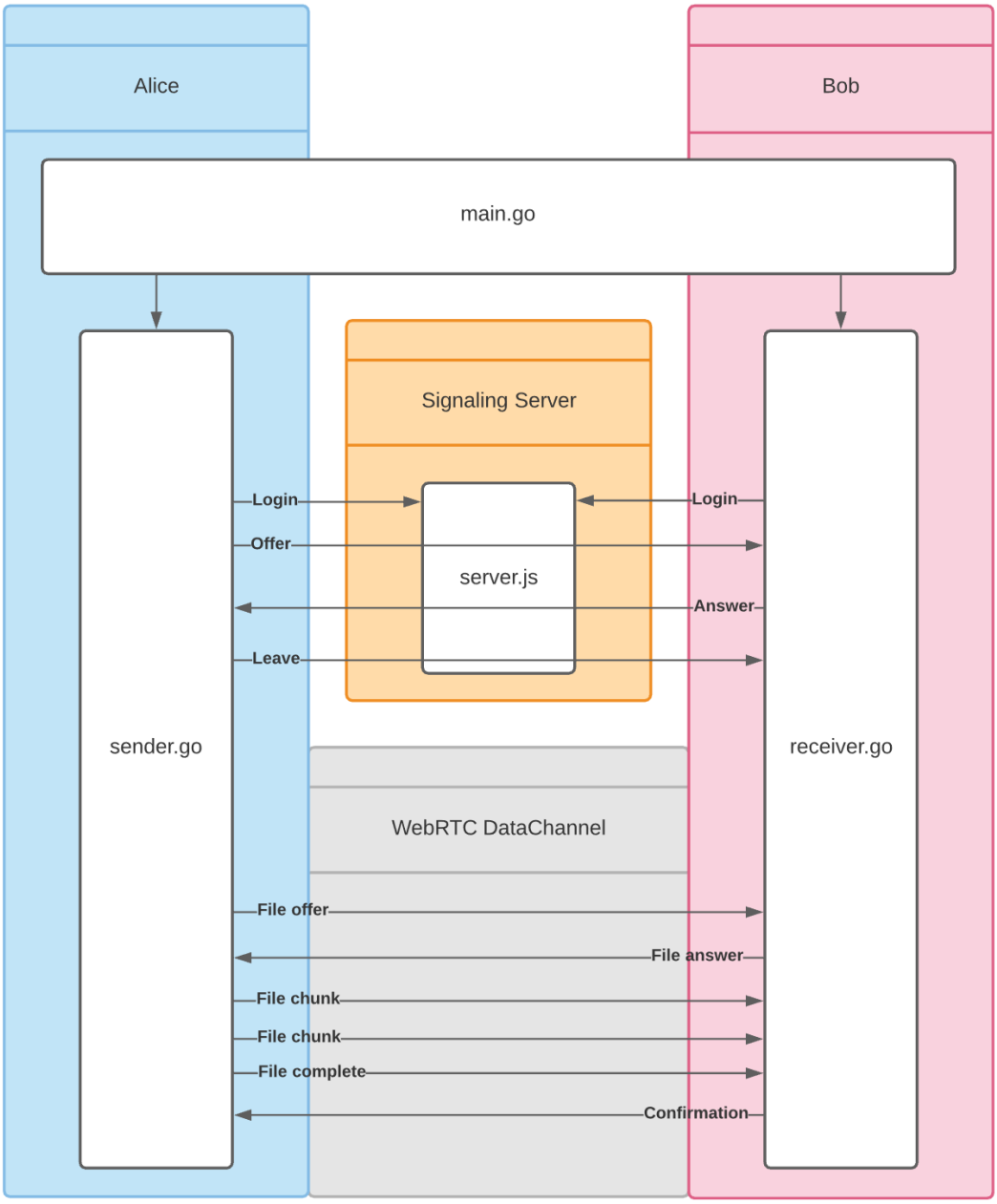

Figure 17: Application architecture and interactions

From top to bottom, we can decompose this process major steps:

- Each user picks a role in **main.go** and is redirected to the corresponding Go file.
- Each user gathers the required information needed for the signaling step. This includes peerConnection configuration, certificate generation, fingerprint extraction and passphrase exchange.
- The WebRTC SDP and ICE candidates are exchanges over the signaling server via a WebSocket connection.
- A DataChannel is opened.
- The file transfer protocol is done over the newly created DataChannel.

Signaling protocol

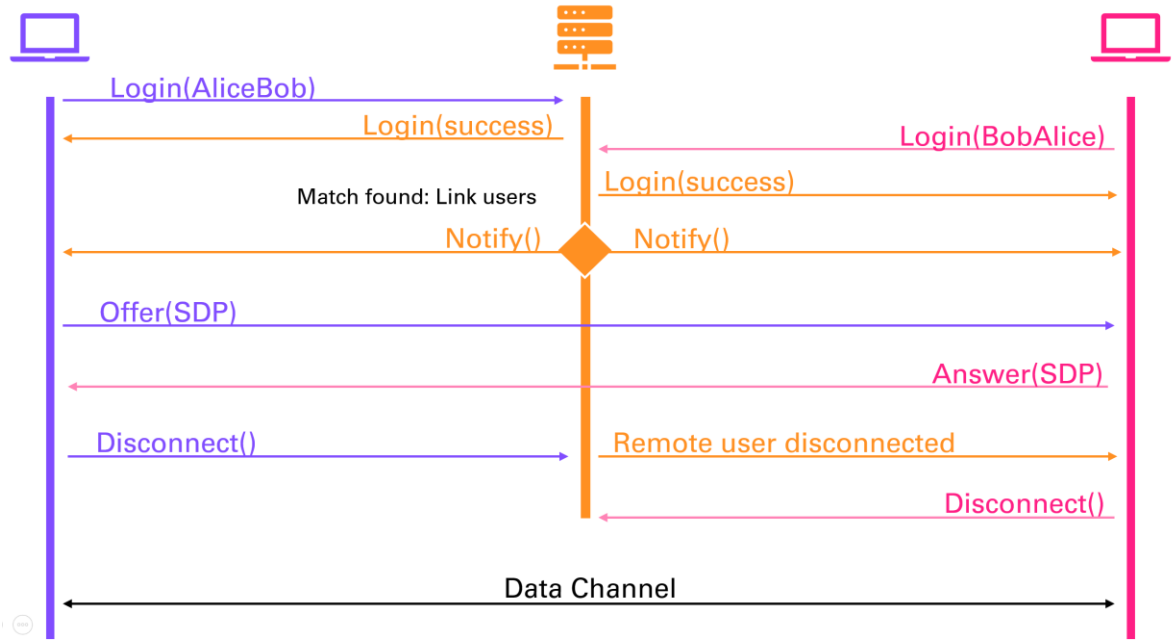

Figure 18: Simplified Signaling Protocol

By taking a closer look at the signaling protocol (see Fig. 18), we can see that the server only interacts during the login phase. Once a match is made between two users, it only acts as a relay point for all following messages.

File exchange protocol

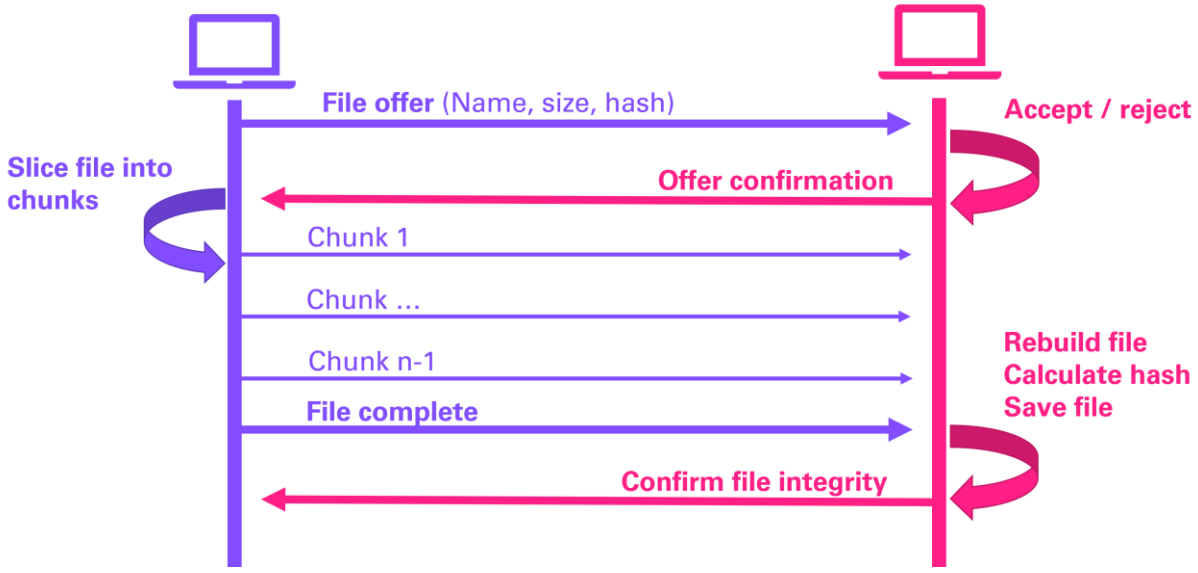

Figure 19: File Exchange Protocol

The file exchange protocol (see Fig. 19) is rather classical. First, the sender makes an offer and waits for confirmation. The file is then sliced into multiple chunks and sent to the receiver. Once the file successfully recomposed, a confirmation is sent by the receiver.

3.4 Code overview

The program starts execution at **main.go**. This part redirects the user to either **sender.go** or **receiver.go**. Those two last files have the exact same structure and everything from generating certificate, exchanging SDPs, establishing connection, and transferring files is done inside of them. Both files can be decomposed in three distinct sections.

- WebRTC initialization
- WebRTC file exchange
- WebSocket Signaling

The catch to understand these files is that even though the file exchange protocol is written first, in reality, the signaling step has to be done beforehand. This is made possible by the fact that the whole file exchange is event-based, and its trigger is the opening of a WebRTC **DataChannel**, which is our last signaling step.

To sum it up, the time-based execution goes WebRTC initialization first, WebSocket Signaling and finally WebRTC file exchange.

The whole code package looks the following (see Fig. 20):

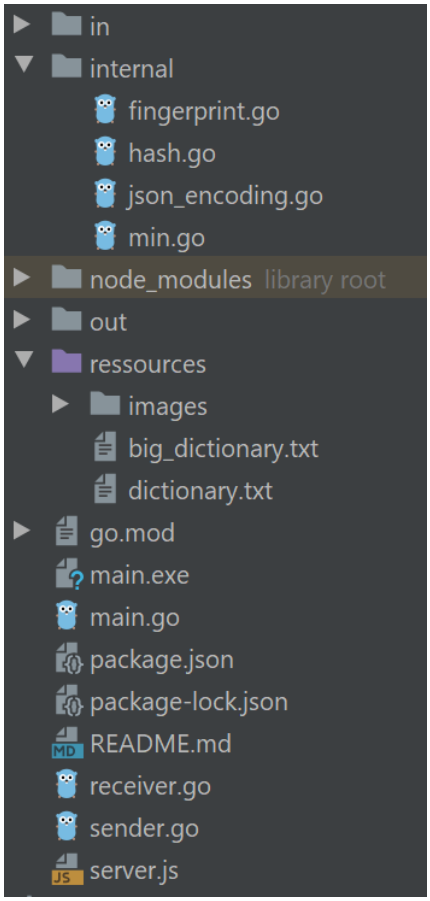

Figure 20: Application source files

We will look at the main features of each Go file. The whole source code can be found at https://github.com/TheWarWolf/FlyingDutchman.

3.4.1 Main

Like in many other languages, this is where our execution will begin. The role of this file is to orient the user on the sender or receiver path.

It is basically composed of a user input receiver and a switch-case condition to orient the execution to the right Go file.

```
package main

import (
   "fmt"
   "log"
   "syscall"
)

var keepExe = true

func recoverMain() {
   if r := recover(); r != nil {
      log.Println("An error occured!\nRecovered from ", r)
      keepExe = true
   }
}
func choseState() {
   defer recoverMain()

   fmt.Println("\nWelcome aboard, cabin boy !")
   var userResponse string
   var userStateDefined = false
   var userIsDone = false

   for !userStateDefined && !userIsDone {
      fmt.Println("Are you sender or receiver of the file? ('s'/'r')\nYou
can stop the program by typing quit ('q')")
      fmt.Scanln(&userResponse)

      switch userResponse {
      case "s", "send":
         userStateDefined = true
         fmt.Println("Preparing to send...")
         Sender()
         userIsDone = true

      case "r", "receive":
         userStateDefined = true
         fmt.Println("Preparing to receive...")
         Receiver()
         userIsDone = true

      case "q", "quit":
         userStateDefined = true
         userIsDone = true
         syscall.Exit(0)

      default:
         fmt.Printf("Sorry, \"%s\" is not a functionnal command, please try
again:\n", userResponse)
```

```
        }
    }
    fmt.Println("The glowing boat disappeared in the mist...")

}
func main() {
    for keepExe {
        keepExe = true
        choseState()
    }
}
```

Code 1: main.go

Execution starts at **func main()** and the **recoverMain()** function is a way to catch at top-level all eventual unhandled errors that might occur. This prevents a total application crash and returns a friendlier error message to the user. It also resets the execution and keeps the program ready for a second run.

The **choseState()** function listens for a user input and redirects to **sender.go** for an "s" or "sender" input, respectively to **receiver.go** for "r" or "receiver". Application can also be closed by typing "quit". Any other input will just result in the user given the choice to re-enter a command.

3.4.2 Sender

The **sender.go** file is where everything takes place on the sending user side. The code is separated into three distinct sections: the WebRTC base configuration, the WebRTC file exchange part and the WebSocket Signaling part.

WebRTC initialization – sender's side

```
func Sender() {

    // Prepare the configuration
    config := webrtc.Configuration{
        ICEServers: []webrtc.ICEServer{
            {
                URLs:       []string{"turn:turn.flying-dut.ch:3478",
"stun:stun.flying-dut.ch:3478"},
                Username:   "captain",
                Credential: "Axp2oSr56d5"},
        },
    }

    // Create a new RTCPeerConnection
    peerConnection, err := webrtc.NewPeerConnection(config)
    if err != nil {
        panic(err)
    }

    // Generate your personal certificate passphrase
    tlsFingerprints, err :=
peerConnection.GetConfiguration().Certificates[0].GetFingerprints()
    fingerprint := internal.FingerprintToString(tlsFingerprints[0])
    localPassphrase := internal.FingerprintToPhrase(fingerprint)
    fmt.Println("Your passphrase is: " + localPassphrase)
```

```
   // Create a datachannel with label 'data'
   dataChannel, err := peerConnection.CreateDataChannel("data", nil)
   if err != nil {
      panic(err)
   }

   // Set the handler for ICE connection state
   // This will notify you when the peer has connected/disconnected
   peerConnection.OnICEConnectionStateChange(func(connectionState
webrtc.ICEConnectionState) {
      log.Printf("ICE Connection State has changed: %s\n",
connectionState.String())
      if connectionState.String() == "disconnected" {
         fmt.Println("Remote user disconnected: Taking you back to main
menu.")
         peerConnection.Close()
         main()
      }
   })
```

Code 2: sender.go - WebRTC initialization

In this code extract, all the WebRTC configuration takes place. No link is established at this point, we only generate all needed information for the next part: the signaling step.

The first thing we need is to set our webrtc.Configuration variable. It contains the URLs of our STUN/TURN server as well as required credentials. We can now initiate an **RTCPeerConnection** object. At this point, a self-signed certificate is automatically generated and will be used in the TLS connection.

The signaling part will be done on each side by logging in with a passphrase. These passphrases are obtained by a derivation of the fingerprint we extract from the generated certificates. This whole process is done by the **FingerprintToPhrase()** function, which we will see later.

The last step is to create our **dataChannel** using the **CreateDataChannel()** function and setup an ICE handler which notifies the user for every change of ICE connection state, as well as redirecting the user to **main()** in case the remote user disconnects.

WebSocket signaling – sender's side

Assuming the user properly got the receiver's passphrase via some other channel, i.e. WhatsApp, we now have every bit of information needed to find this remote user. The whole point of the signaling step is to exchange these data through a classic offer-answer protocol over our signaling server to open a secure **DataChannel** and start sending files.

```
type Message struct {
   Type    string
   Success bool
   Offer   string
   Answer  string
   Name    string
   Sender  string
}
```

Code 3: sender.go - signaling - struct definition

All WebSocket transactions with the server are done using JSON messages. For this reason, we initiate a struct with all possible fields used during the exchange process.

```
var remote string

// ask user for remote passphrase
fmt.Println("Enter your receiver's passphrase:")
fmt.Scanln(&remote)
```

Code 4: sender.go - signaling - remote passphrase input

The user is now asked for the remote passphrase. The value is stored under the "remote" **var**.

```
// define websocket connection to signaling server
socket := gowebsocket.New("ws://signal.flying-dut.ch:9090")
```

Code 5: sender.go - signaling - WebSocket connection

A WS connection is initialized with our server. For testing purposes, the server can be run locally and reached with "ws://127.0.0.1:9090".

```
// on connection: send login info to signaling server
socket.OnConnected = func(socket gowebsocket.Socket) {
   log.Println("Connected to server")

   ans := Message{Type: "login", Name: localPassphrase + remote}
   b, err := json.Marshal(ans)
   if err != nil {
      panic(err)
   }
   socket.SendBinary(b)
}
```

Code 6: sender.go - signaling - onConnected

OnConnected is triggered by a successful connection notification sent by the server. From this point on, our signaling protocol starts. The first message we send is aimed at the server itself and will be used for login and matching the remote user. The name used for the login is our local passphrase concatenated with the remote passphrase. This is used for matching purpose and will be explained in detail in the security chapter (4.5).

```
// on text message: read its content and switch between cases
socket.OnTextMessage = func(message string, socket gowebsocket.Socket) {
   var m Message

   err := json.Unmarshal([]byte(message), &m)
   if err != nil {
      panic(err)
   }
   switch m.Type {
```

Code 7: sender.go - signaling - onTextMessage

All incoming messages from the server are now treated the following way: the message **Type** field is read, and a switch-case loop is used to take the corresponding action.

The following **Types** can be received:

Type	Signification	Resulting action
"login"	Message sent by the server to confirm login success or fail	For a successful login, notifies the user, and wait for a "linked" Type message
"linked"	Sent by the server for matching the remote user	Initiate the SDP exchange by sending an "offer" Type message to the remote user
"nomatch" or "rejected"	Sent by the server if no match is found or sent by the remote user for a rejected offer	Notifies the user and asks for a new passphrase to regenerate an offer.
"answer"	Sent by the remote user after accepting the offer	Check for remote user's certificate validity, notifies user, initiates DataChannel, send a "leave" Type message and close WebSocket connection
"leave"	Sent by server upon linked user's disconnection	Notifies the user and closes the WebSocket connection

Table 1: sender.go - signaling messages Types

The offer-answer exchange is the central part of this signaling process. Here is the detail of its workings.

```
case "linked":
   log.Println("Linked !")
   // create a new peerConnection offer
   offer, err := peerConnection.CreateOffer(nil)
   // gather candidates
   gatherComplete := webrtc.GatheringCompletePromise(peerConnection)

   err = peerConnection.SetLocalDescription(offer)
   if err != nil {
      panic(err)
   }

   // Block until ICE Gathering is complete, disabling trickle ICE
   // we do this because we only can exchange one signaling message
   <-gatherComplete
```

```
   // output the answer in base64 so we can send it
   encodedOffer := internal.Encode(*peerConnection.LocalDescription())

   // send offer to remote user connected with given passphrase
   ans := Message{Type: "offer", Name: remote, Offer: encodedOffer, Sender:
localPassphrase}
   b, err := json.Marshal(ans)
   if err != nil {
      panic(err)
   }
   socket.SendBinary(b)

   log.Println("Sending offer to " + remote)
```

Code 8: sender.go - signaling - linked and offer generation

The offer is generated upon a "linked" **Type** message reception. Using our **peerConnection** object, we can store its generated answer under the "offer" var using the **CreateOffer()** function. We now need to assign this offer to our **peerConnection** using **SetLocalDescription()**.

Because we only make one signaling exchange, candidates can all be gathered directly with **GatheringCompletePromise()** function. What it does is gather all ICE candidates and add them to our **peerConnection** object.

The actual offer we want to send is extracted from our now completed **peerConnection** object by using **LocalDescription()**.

The last step is to encode this description to JSON format and send it under an "offer" **Type** message.

```
case "answer":
   log.Println("Received answer from " + remote)
   var encodedAnswer = m.Answer

   answer := webrtc.SessionDescription{}

   internal.Decode(encodedAnswer, &answer)

   // Checking remote certificate's fingerprint matches given passphrase
   parsed := &sdp.SessionDescription{}
   if err := parsed.Unmarshal([]byte(answer.SDP)); err != nil {
      panic(err)
   }
   fingerprint := internal.ExtractFingerprint(parsed)
   remotePassphrase := internal.FingerprintToPhrase(fingerprint)

   // If certificate matches, set as remote description
   if remotePassphrase == remote {
      fmt.Println("Receiver identity confirmed!")
      err = peerConnection.SetRemoteDescription(answer)
      if err != nil {
         panic(err)
      }
   } else {
      fmt.Println("Receiver's certificate is not matching")
      break
   }
```

```
   // notify and close connection
   msg := Message{Type: "leave", Name: remote}
   c, err := json.Marshal(msg)
   if err != nil {
      panic(err)
   }
   socket.SendBinary(c)

   socket.Close()
   return
```

Code 9: sender.go - signaling - answer received

Assuming the remote user accepted our offer, he added it to its local description and generated its answer in the same way. Its answer is received under an "answer" **Type** message. Before adding the answer as our remote description, we check for the authenticity of the certificate. This is done by extracting the certificate's fingerprint, converting it to a passphrase and comparing it to the remote passphrase we are expecting. All these steps were also done on the receiver's side when receiving our offer.

If the extracted passphrase matches, we can safely set the answer to our **peerConnection** object with **SetRemoteDescription().** From this point on, the WebRTC is all setup, and we can disconnect from the signaling server.

To make sure the other user also disconnects, a **Type "leave"** message is sent before the WebSocket connection is closed using **Close()**.

WebRTC file exchange – sender's side

The signaling part being completed, it is finally time to send files over our newly opened **DataChannel**. The workings of this part are exactly like for the signaling, an exchange of JSON messages having different **Type** fields, only this time we use our WebRTC connection to communicate directly with the remote user.

```
type Exchange struct {
   Type     string
   FileName string
   FileSize int64
   Hash     []byte
   Data     []byte
}
```

Code 10: sender.go - file exchange - struct defintion

We define a new struct called Exchange which contains all the required fields we will need.

```
// Register channel opening handling
dataChannel.OnOpen(func() {
   log.Printf("Data channel '%s'-'%d' open.\n", dataChannel.Label(),
dataChannel.ID())

})
```

Code 11: sender.go - file exchange - onOpen

On the sender's side, the opening of the channel only triggers a notification to the user. The first action will be taken on the remote user's side, which will send a "ready" **Type** message.

```
// Register text message handling
dataChannel.OnMessage(func(msg webrtc.DataChannelMessage) {

   var m Exchange

   err := json.Unmarshal(msg.Data, &m)
   if err != nil {
      panic(err)
   }
   switch m.Type {
```

Code 12: sender.go - file exchange - onMessage

The same switch-case condition is used to take the appropriate action, in regards of the incoming message **Type**.

Here is a table of the possible incoming messages:

Type	Signification	Resulting action
"ready"	DataChannel is setup and the receiver is ready to receive a file offer	The user is asked for a filepath, the file info are extracted and an "offer" Type message is sent to the receiver
"accept"	Receiver accepted our offer, the file can be sent	Proceed to slice the file into chunks and sent them over through multiple "fileChunks" Type messages A "fileComplete" message is sent when all chunks have been uploaded
"received"	Receiver has successfully received all file chunks, has recomposed the file and confirmed its integrity	User is now asked to send another file offer or exit the transaction.
"reject"	Receiver rejected the file offer	User is given the choice to retry, chose another file path or go back to main()
"transferfailed"	An error occurred during the transfer	User is now asked to retry with another file offer or exit the transaction.

Table 2: sender.go - file exchange messages Types

The crucial part in this process is the actual file slicing[17] and sending which happens on an "accept" message reception.

[17] [1]262588213843476. n.d. 'Golang Split Byte Slice in Chunks Sized by Limit'. Gist. Accessed 2 September 2020. https://gist.github.com/xlab/6e204ef96b4433a697b3.

```
case "accept":
   fmt.Println("File offer accepted! Your file is being sent...")
   log.Println("Uploading")

   // sends selected file chunk by chunk
   limit := 45000
   for i := 0; i < len(file); i += limit {
      batch := file[i:internal.Min(i+limit, len(file))]

      msg := Exchange{Type: "fileChunk", Data: batch}
      m, err := json.Marshal(msg)
      if err != nil {
         panic(err)
      }
      sendErr := dataChannel.Send(m)
      if sendErr != nil {
         panic(sendErr)
      }

      chunks := len(file) / limit
      if chunks < 100 {
         chunks = 100
      }

      // each 1%, send a notification message
      if (i/limit)%(chunks/100) == 0 && i != 0 {
         fmt.Print("|")
         msg := Exchange{Type: "mega"}
         m, err := json.Marshal(msg)
         if err != nil {
            panic(err)
         }
         sendErr := dataChannel.Send(m)
         if sendErr != nil {
            panic(sendErr)
         }
      }
   }
   fmt.Println(" -->Upload done\nWaiting for confirmation...")
   msg := Exchange{Type: "fileComplete"}
   m, err := json.Marshal(msg)
   if err != nil {
      panic(err)
   }
   sendErr := dataChannel.Send(m)
   if sendErr != nil {
      panic(sendErr)
   }
```

Code 13: sender.go - file exchange - "accept" case

The size limit of each chunk is set to be as close as the maximal limit of a WebRTC DataChannel message, increasing transfer speed drastically. Each pass in the for-loop saves the next chunk of our binary file under the “batch” variable, which is then sent over through a “fileChunk” **Type** message.

Each 1% of the chunks being sent, a “mega” **Type** message is sent in order to get a visual feedback of the transfer’s progress on the receiver’s side.

Once the upload done, a “fileComplete” message is sent, for the receiver to start recomposing the file and check its integrity. The sender is now waiting for confirmation.

3.4.3 Receiver

The **receiver.go** file is extremely close to **sender.go**. It is composed of the exact same three distinct parts.

The WebRTC initialization having no significant difference with the sender’s side, we will look at the differences in both the signaling step and the file exchange step.

WebSocket signaling – receiver’s side

On the receiver’s side, all messages are also treated in a switch-case condition, here are the different possible incoming message **Types** for the signaling step:

Type	Signification	Resulting action
“login”	Message sent by the server to confirm login success or fail	For a successful login, notifies the user, and wait for a “linked” Type message
“linked”	Sent by the server for matching the remote user	Notifies the user and wait for an offer
“offer”	Sent by the remote user after accepting the offer	Check for remote user’s certificate validity, notifies user, set offer as remote description, build the answer, and send it to remote user as an “answer” Type message
“leave”	Sent by server upon linked user’s disconnection	Notifies the user and closes the WebSocket connection

Table 3: receiver.go - signaling messages Types

The most important step here is the offer case.

```
case "offer":
   log.Println("Received offer from " + m.Name)

   var encodedOffer = m.Offer
   offer := webrtc.SessionDescription{}
   internal.Decode(encodedOffer, &offer)

   // Checking remote certificate's fingerprint matches given passphrase
   parsed := &sdp.SessionDescription{}
   if err := parsed.Unmarshal([]byte(offer.SDP)); err != nil {
      panic(err)
   }
   fingerprint := internal.ExtractFingerprint(parsed)
   remotePassphrase := internal.FingerprintToPhrase(fingerprint)

   // If certificate matches, set as remote description
   if remotePassphrase == remote {
      fmt.Println("Receiver identity confirmed!")
      err = peerConnection.SetRemoteDescription(offer)
      if err != nil {
         panic(err)
      }
   } else {
      fmt.Println("Receiver's certificate is not matching")
      break
   }

   // Create an answer
   answer, err := peerConnection.CreateAnswer(nil)
   if err != nil {
      panic(err)
   }

   // Create channel that is blocked until ICE Gathering is complete
   gatherComplete := webrtc.GatheringCompletePromise(peerConnection)

   // Sets the LocalDescription, and starts our UDP listeners
   err = peerConnection.SetLocalDescription(answer)
   if err != nil {
      panic(err)
   }

   // Block until ICE Gathering is complete, disabling trickle ICE
   // we do this because we only can exchange one signaling message
   <-gatherComplete

   encodedAnswer := internal.Encode(*peerConnection.LocalDescription())

   ans := Message{Type: "answer", Name: remote, Answer: encodedAnswer}
   b, err := json.Marshal(ans)
   if err != nil {
      panic(err)
   }
   socket.SendBinary(b)
   log.Println("Sending answer to " + remote)
```

Code 14: receiver.go - signaling – “offer” case

The same steps as the one on the sender's side are executed. We first extract the fingerprint from the received certificate and convert it to a passphrase using **FingerprintToPhrase()**. If the obtained passphrase matches the expected one, the offer can safely be added as the **peerConnection**'s remote description.

From this point on, we generate an answer, set it as local description, gather the ICE candidates, and send over an "answer" **Type** message, containing our **peerConnection**'s local description.

WebRTC file exchange – receiver's side

Once the **DataChannel** opens, the receiver is the one to initiate the message exchange.

```
// Register channel opening handling
dataChannel.OnOpen(func() {
   log.Printf("Data channel '%s'-'%dataChannel' open.\n",
dataChannel.Label(), dataChannel.ID())

   // Notify sender we are ready to receive a file offer
   msg := Exchange{Type: "ready"}
   m, err := json.Marshal(msg)
   if err != nil {
      panic(err)
   }
   sendErr := dataChannel.Send(m)
   if sendErr != nil {
      panic(sendErr)
   }
})
```

Code 15: receiver.go - file exchange - onOpen

A "ready" **Type** is sent to the remote user to let him know the channel is up and working.

From this point on, all incoming messages and respective actions taken are again handled through a switch-case condition. The following Types can be received:

Type	Signification	Resulting action
“fileInfo”	The sender is making a file offer	All received info about the file offer are displayed, such as name, type and size The user is asked to accept or deny the offer, a resulting “accept” or “reject” message is sent
“fileChunk”	The sender	For each chunk message received, its data field is appended to the “rebuiltFile” object
“mega”	1 more percent of the chunks has been sent	Displays a | char to track download progress.
“fileComplete”	All file chunks have been sent	The “rebuiltFile” object is now completed and its hash is calculated and compared to the one we received from the “fileInfo” message to confirm integrity If the integrity is confirmed, the file is rebuilt and saved in the default /out output folder.
“newFile”	The sender wants to make another file offer	User is now asked to continue receiving files or exit the program

Table 4: receiver.go - file exchange messages Types

An incoming “fileInfo” offer is handled the following way:

```
case "fileInfo":
   // display received file
   fmt.Printf("Received a file offer:\nName: %s\nSize: %d byte\n",
m.FileName, m.FileSize)
   var userResponse string
   fmt.Println("Type 'yes' to accept offer:")
   fmt.Scanln(&userResponse)
   switch userResponse {
   case "yes", "y":
      log.Println("downloading")

      outputPath = "out/" + m.FileName
      fileHash = m.Hash

      msg := Exchange{Type: "accept"}
      m, err := json.Marshal(msg)
      if err != nil {
         panic(err)
      }
      sendErr := dataChannel.Send(m)
      if sendErr != nil {
         panic(sendErr)
      }
```

```
    case "n", "no":
        msg := Exchange{Type: "reject"}
        m, err := json.Marshal(msg)
        if err != nil {
            panic(err)
        }
        sendErr := dataChannel.Send(m)
        if sendErr != nil {
            panic(sendErr)
        }

    default:
        fmt.Println("Unknown command, please try again:")
    }
```

Code 16: receiver.go - file exchange - "fileInfo" case

File name and size are displayed to the user so he can make a decision on whether accepting or rejecting the offer. If the offer is accepted, the file hash is saved for later integrity check, the output path is set to “out/FileName” and an “accept” **Type** message is sent. Otherwise, an “reject” message is sent.

3.4.4 Server

We had a look at the signaling step from both perspectives and we will now see how the signaling server handle these communications in-between.

```
//require our websocket library
var WebSocketServer = require('ws').Server;

//creating a websocket server at port 9090
var wss = new WebSocketServer({port: 9090});

//all connected to the server users
var users = {};
```

Code 17: server.js - WebSocket initialisation

This is the initiation of our server, it starts a WebSocket on port 9090 and initiates an empty **users** array.

```
//when a user connects to our sever
wss.on('connection', function (connection) {

    console.log("User connected");

    //when server gets a message from a connected user
    connection.on('message', function (message) {

        var data;
        //accepting only JSON messages
        try {
            data = JSON.parse(message);
        } catch (e) {
            console.log("Invalid JSON");
            data = {};
        }

        //switching type of the user message
        switch (data.Type) {
```

Code 18: server.js - user login and incoming message handling

The server deals with incoming messages the exact same way sender and receiver do. The **Type** field of each new JSON message is read and corresponding actions are taken using a switch-case condition.

Only the “login” **Type** is fully read by the server, all other messages are directly transferred to the remote user:

Type	Signification	Resulting action
“login”	New user wants to login	Check if username is already registered: if not, add to users array and send a login Type message with Success = “true”. Send a message with Success = “false” otherwise. On each new login, check if remote user is already connected: if it is, link them together and send them both a “linked” Type message.
default	All other messages of the signaling protocol	For any other types of messages, all the server does is transfer it to the linked user.

Table 5: server.js - incoming messages Types

Here is how this first case works:

```
case "login":
    console.log("User logged", data.Name);

    //if anyone is logged in with this username then refuse
    if (users[data.Name]) {
        sendTo(connection, {
            Type: "login",
            Success: false
        });
    } else {
        //save user connection on the server
        users[(data.Name)] = connection;
        connection.Name = data.Name;

        sendTo(connection, {
            Type: "login",
            Success: true
        });

        //if UserB exists then link connections A and B
        var conn = users[reverseString(data.Name)];
        if (conn != null) {
            //setting that UserA connected with UserB
            connection.otherName = reverseString(data.Name);
            conn.otherName = data.Name;
            console.log("Users linked", connection.Name +
connection.otherName);

            sendTo(conn, {
                Type: "linked",
                Offer: data.Offer,
                Name: data.Sender
            });
            sendTo(connection, {
                Type: "linked",
                Offer: data.Offer,
                Name: data.Sender
            });
        }
    }

    break;
```

Code 19: server.js - "login" case

The match is done by searching for a user in the “users” array which has the exact reverse string as login name[18]. If someone is found, the “connection” object, which is our current user, the one which just logged in, is set as **conn.otherName** and “conn”, representing the matched user is added to **connection.otherName**. Each user now has an “otherName” attribute, which will be used for any following message transfer.

[18] This procedure of logging in with reversed usernames is explained in the security chapter

```
default:
    //any message sent by A is transferred to B
    var conn = users[connection.otherName];
    sendTo(conn, data);
    console.log("Sending message to: ", conn.Name);

    break;
```

Code 20: server.js - default case

The last part of the signaling server is about handling a linked user disconnection at any point in the transaction.

```
connection.on("close", function () {

    if (connection.Name) {
        delete users[connection.Name];
        console.log("Disconnecting from ", connection.Name);


        if (connection.otherName) {
            console.log("Notifying linked user");
            var conn = users[connection.otherName];
            conn.otherName = null;

            if (conn != null) {
                sendTo(conn, {
                    Type: "leave"
                });
            }
        }
    }
});
```

Code 21: server.js - connection closed handler

This allows for any side of the exchange to be notified to leave whenever the linked user lost the WebSocket connection before sending himself the “leave” **Type** message.

3.4.5 Internal directory

The internal package is made of four Go files each containing various methods used by the previously seen files.

The main file, **fingerprint.go**, holds the following functions:

func	input	output	use
ExtractFingerprint()	Sdp.sessionDescription	string	Extracts the fingerprint attribute contained in the remote received SDP and convert it to a string.
FingerprintToString()	webrtc.DTLSFingerprint	string	Extracts the fingerprint from our own generated certificate.
FingerprintToPhrase()	string	string	Converts a given string fingerprint into a passphrase of 5 words taken from a dictionary.
readLines()	string	[]string	Converts a text document located at the given string path into an array of strings representing each new line. This is used to read the dictionary file.
Reverse()[19]	string	string	Outputs the mirrored version of a given string (first char becomes last char). Used to reverse passphrases in the signaling step.

Table 6: fingerprint.go methods

The **hash.go** file contains the **CreateHash()** function. It is used during the file transfer protocol for the receiver to confirm integrity.

func	input	output	use
CreateHash()	[]byte	[]byte	Outputs the sha-1 hash of any given byte array.

Table 7: hash.go methods

[19] This method is also used within the signaling server code and is therefore defined there as well.

The **json_encoding.go** file contains encoding and decoding methods for JSON. Both are used respectively on the offer and answer during the signaling step.

func	input	output	use
Encode()	interface{}	string	Encodes the input in base 64
Decode()	String, interface{}		Decodes from base 64

Table 8: json_encoding.go methods

The last file **min.go** only has a **Min()** function returning the minimal value between two integers. This simple method is used during the decomposition of a file into chunks.

func	input	output	use
Min()	int, int	int	Returns the smaller of the two given integers

Table 9: min.go method

3.5 Conclusion

This chapter gave us a good glance at the overall design choices and the execution of one complete file exchange. We went over the main functionalities of the major files and the different methods they use to accomplish their respective tasks.

We have seen how the different protocols were defined and executed, but we are missing the reason for such design choices. Why is the code separated in two distinct roles or why are the login and matching done with passphrases are questions we will answer in the next chapter (4) about a major aspect of this project: security.

4
Security

4.1 Mutual authentication problem

To make the exchange safe, the first thing to do is to clearly define where the flaws in our program are. Which transaction are secured, and which are not?

The first assumption we make is that once the **DataChannel** is opened, all transactions on this medium are secured. This can be said in our project for two reasons:

- The use of a peer-to-peer connection:
    - this one-to-one simplistic model let us use all WebRTC features and make the resulting DTLS connection end-to-end encrypted. This could not be the case in a multiple users' scenario, where media servers would be needed in-between.
- The use of DataChannel
    - DataChannel in one of the possible channels that can be used in a **peerConnection**. Its encryption can easily be done, which would not be the case with other channels. For instance, real time video feed encryption is still one of the main challenges in security today.

The challenge therefore lays before the opening of the channel, during the signaling step. Precisely, the real issue is to prove the authenticity of the remote user while exchanging SDP and ICE Candidates.

Furthermore, the signaling server must be considered untrusted. For this reason, we need a way to safely recognize the incoming WebRTC offers and answers as being the one expected.

4.2 Certificate and fingerprint

The solution to this problem lays into the certificates used inside the SDP.

For a P2P **DataChannel**, the offer and answer look the following:

Offer SDP Contents	RFC#/Notes
v=0	[RFC4566]
o=- 20518 0 IN IP4 0.0.0.0	[RFC4566] - Session Origin Information
s=-	[RFC4566]
t=0 0	[RFC4566]
a=group:BUNDLE data	[I-D.ietf-mmusic-sdp-bundle-negotiation]
a=ice-options:trickle	[I-D.ietf-mmusic-trickle-ice]
****** Application m=line *********	******************************
m=application 54609 UDP/DTLS/SCTP webrtc-datachannel	[I-D.ietf-rtcweb-data-channel]
c=IN IP4 203.0.113.141	[RFC4566]
a=mid:data	[RFC5888]
a=sendrecv	[RFC3264] - Alice can send and recv non-media data
a=sctp-port:5000	[I-D.ietf-mmusic-sctp-sdp]
a=max-message-size:100000	[I-D.ietf-mmusic-sctp-sdp]
a=setup:actpass	[RFC5763] - Alice can act as DTLS client or server
a=tls-id:1	[I-D.ietf-mmusic-dtls-sdp]
a=ice-ufrag:074c6550	[RFC5245] - Session Level ICE parameter
a=ice-pwd:a28a397a4c3f31747d1ee3474af08a068	[RFC5245] - Session Level ICE parameter
a=fingerprint:sha-256 19:E2:1C:3B:4B:9F:81:E6:B8:5C:F4:A5:A8:D8:73:04 :BB:05:2F:70:9F:04:A9:0E:05:E9:26:33:E8:70:88:A2	[RFC5245] - Session DTLS Fingerprint for SRTP
a=candidate:0 1 UDP 2113667327 192.0.2.4 61665 typ host	[RFC5245]
a=candidate:1 1 UDP 1694302207 203.0.113.141 54609 typ srflx raddr 192.0.2.4 rport 61665	[RFC5245]
a=end-of-candidates	[I-D.ietf-mmusic-trickle-ice]

Table 10: offer SPD example[20]

[20] Example taken from https://tools.ietf.org/id/draft-ietf-rtcweb-sdp-08.html

Answer SDP Contents	RFC#/Notes
v=0	[RFC4566]
o=- 16833 0 IN IP4 0.0.0.0	[RFC4566] - Session Origin Information
s=-	[RFC4566]
t=0 0	[RFC4566]
a=group:BUNDLE data	[I-D.ietf-mmusic-sdp-bundle-negotiation]
****** Application m=line *********	**********
m=application 49203 UDP/DTLS/SCTP webrtc-datachannel	[I-D.ietf-mmusic-sctp-sdp]
c=IN IP4 203.0.113.77	[RFC4566]
a=mid:data	[RFC5888]
a=sendrecv	[RFC3264] - Bob can send and recv non-media data
a=sctp-port:5000	[I-D.ietf-mmusic-sctp-sdp]
a=max-message-size:100000	[I-D.ietf-mmusic-sctp-sdp]
a=setup:active	[RFC5763] - Bob is the DTLS client
a=tls-id:1	[I-D.ietf-mmusic-dtls-sdp]
a=ice-ufrag:c300d85b	[RFC5245] - Session Level ICE username frag
a=ice-pwd:de4e99bd291c325921d5d47efbabd9a2	[RFC5245] - Session Level ICE password
a=fingerprint:sha-256 6B:8B:F0:65:5F:78:E2:51:3B:AC:6F:F3:3F:46:1B:35:DC:B8:5F:64:1A:24:C2:43:F0:A1:58:D0:A1:2C:19:08	[RFC5245] - Session DTLS Fingerprint for SRTP
a=candidate:0 1 UDP 2113667327 198.51.100.7 51556 typ host	[RFC5245]
a=candidate:1 1 UDP 1694302207 203.0.113.77 49203 typ srflx raddr 198.51.100.7 rport 51556	[RFC5245]
a=end-of-candidates	[I-D.ietf-mmusic-trickle-ice]

Table 11: answer SDP example[21]

[21] Example taken from https://tools.ietf.org/id/draft-ietf-rtcweb-sdp-08.html

The following figure (see Fig. 21), represents the content of a session description which we will use:

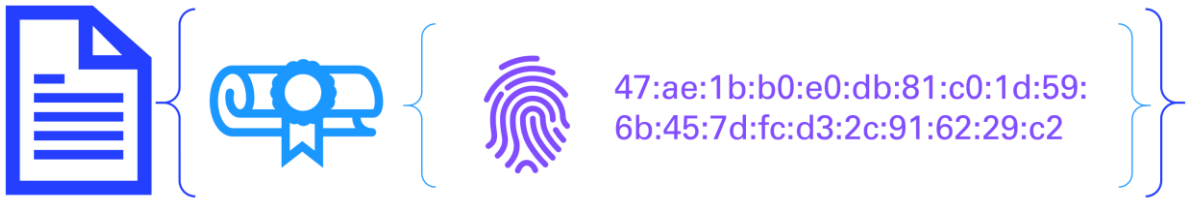

Figure 21: SDP decomposition

Each SDP contains a self-signed certificate fingerprint. In our case, this certificate is automatically generated when creating our **peerConnection** object.

Although we technically cannot extract the fingerprint from the DTLS connection on our opened **DataChannel**, we can do it directly from the SDP before the channel is even opened.

Using this fingerprint for authentication is the best way to prove one's identity as a false fingerprint within an offer or answer will result in an error when setting the **peerConnection**. If the remote user can communicate us its fingerprint via a secure external channel, all there is to do left is to check whether the incoming SDP contains this exact fingerprint or not.

4.3 Fingerprint extraction

The fingerprint can be found in our SDP offer and answer both under the field a=fingerprint, as seen in Table 10 and Table 11. Getting our own fingerprint is a slightly different and easier process than extracting the remote one.

From the sender's perspective, getting its own fingerprint can be done directly from the WebRTC **peerConnection** object:

```
tlsFingerprints, err :=
peerConnection.GetConfiguration().Certificates[0].GetFingerprints()
fingerprint := internal.FingerprintToString(tlsFingerprints[0])
```

Code 22: own fingerprint extraction

The **GetFingerprints()** method returns a **webrtc.DTLSFingerprint** object, which we need to convert to a simple string using the internal method **FingerprintToString()**. This string conversion allows us to get the fingerprint under the exact same format as we would find it in the SDP, meaning all capital and ":" separated.

To get the remote fingerprint now, which in this scenario means extracting from the receiver's answer, we defined another internal function called **ExtractFingerprint()**.

```
parsed := &sdp.SessionDescription{}
if err := parsed.Unmarshal([]byte(answer.SDP)); err != nil {
   panic(err)
}
fingerprint := internal.ExtractFingerprint(parsed)
```

Code 23: remote fingerprint extraction

This last method takes a pointer to an **sdp.SessionDescription** object and outputs the contained fingerprint as a string.

At this point, we now have both our fingerprint and the one extracted from the remote user's answer under the following format: **6B:8B:F0:65:5F:78:E2:51:3B:AC:6F:F3:3F:46:1B:35:DC:B8:5F:64:1A:24:C2:43:F0:A1:58:D0:A1:2C:19:08**

From the receiver's perspective, the exact same methods are applied to get respectively the self-generated certificate fingerprint and extract the remote one from the sender's SDP offer.

4.4 Passphrase derivation

4.4.1 Simplifying the fingerprint

We now have a way of identifying the remote user but communicating the whole fingerprint would be a lot of trouble. Therefore, we derive it to a passphrase to get a much simpler and friendlier code for mutual authentication.

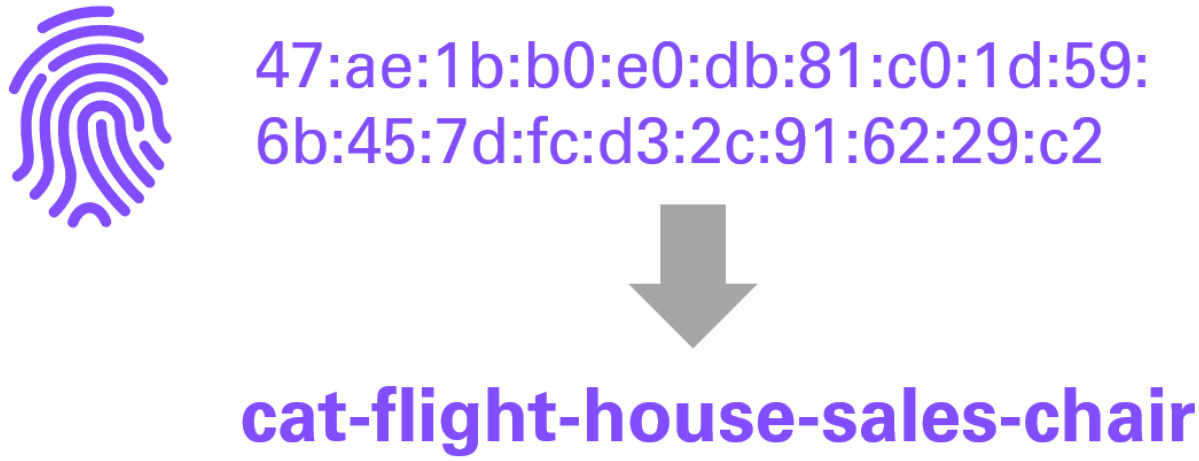

Figure 22: Converting a fingerprint to a passphrase

Such a passphrase (see Fig. 22) can now be communicated easily and even spoken.

4.4.2 Derivation algorithm

The idea for this process is to consider the whole fingerprint as a large number, perform a base change and attribute five words taken from an English dictionary to the resulting number.

```
Pseudocode:
int left = int(fingerprint)
int base = len(dictionary)
For(5){
        int wordIndex = left%base
        passphrase.add(dictionary[wordIndex])
        left = left / base
        }
```

Figure 23: Base change pseudocode

This pseudocode (see Fig. 23) shows the theoretical implementation of such a base change. For each iteration, we pick the word at the dictionary index corresponding to the fingerprint (**left**) modulo the size of our dictionary (**base**). We then divide our fingerprint by the dictionary size and repeat the whole process five time.

The real implementation is done this way:

```go
func FingerprintToPhrase(fingerprint string) string {
   // get the dictionary as an array of words from a text file
   dictionary, err := readLines("ressources/dictionary.txt")
   if err != nil {
      panic(err)
   }
   // format fingerprint to hexadecimal string
   hexa := strings.ReplaceAll(fingerprint, ":", "")

   // value of fingerprint
   left := new(big.Int)
   left.SetString(hexa, 16)

   // length of dictionary
   base := new(big.Int)
   base.SetInt64(int64(len(dictionary)))

   passphrase := ""
```

```
   // for five runs, picks the word at index left%base and adds it to the
passphrase
   for i := 0; i < 5; i++ {
      wordIndex := new(big.Int)
      // wordindex = left % base
      wordIndex.Rem(left, base)
      word := dictionary[wordIndex.Int64()]

      if i != 4 {
         passphrase += word + "-"
      } else {
         passphrase += word
      }

      // divides the fingerprint value by the dictionary length
      left.Div(left, base)
   }

   return passphrase
}
```

Code 24: internal.fingerprint.go - passphrase derivation

We first load the **.txt** dictionary into an array. The string fingerprint must be cleaned of all ":" **char** and converted to a **big.Int** object. base is also defined as a **big.Int** set to the dictionary length. An empty passphrase is initiated.

From this point on, the five iterations can be done, and the resulting passphrase is outputted.[22]

4.4.3 Information loss

Switching from a complete sha-256 fingerprint to a simple passphrase obviously represents a loss of data. Our dictionary is 10'522 words long, making it approximately $2^{13,3}$ strong. By making the passphrase 5 words long, this brings the possibilities up to $2^{66,5}$, approximating a 55 bits security keyspace. Once the derivation being done, only 2^{66} possibilities remain from our original 2^{256} combination strong fingerprint.

Although it is a gigantic reduction, it is still a safe way to identify our remote user, as finding a certificate which matches this passphrase cannot be done in the short time interval between the two connections.

Because of the way the signaling server is built, the time interval one would have to generate a matching certificate is only the delay between the two users logging in, a matter of seconds. Once a match is found, no other link can be established for any of the concerned peers. By simply limiting the login request to one per second, we basically eliminate any possibility of a man-in-the-middle attack.

[22] Note that for readability reasons, "-" is added between each word during the iterations.

4.5 Use in signaling protocol

4.5.1 As a login system

The protocol can be defined more formally. As we now have a way to safely authenticate the other user through the use of passphrases, why not use them directly as login info on our signaling server?

This way of doing the login reduces the user's input to only one: the remote user's passphrase. Before even connecting to the server, both users know each other's passphrases. From this point, the idea is to have them both to login with the concatenation of passphrases as username. There is a catch: one of the sides, in our case the receiver, must totally reverse the resulting string. By doing this, users who are supposed to be matched together have the exact same login username, only reversed.

Figure 24: Alice's login name

Figure 25: Bob's login name

Matching the users is now a straightforward task: each time a new user connects, we check for a user having the exact opposite string.

4.5.2 For authenticity check

Once a match has been made, the linked user is not yet reliably authenticated. At this point, anyone could have basically just got our username and reversed it. The crucial part is to make sure the incoming offer or answer actually contains a fingerprint which deriver to the expected passphrase.

Before adding any incoming SDP as a remote description, we extract its fingerprint, derive it to a passphrase and make sure it matches the one we expected.

On the sender's side, here is how we handle incoming answers:

```
case "answer":
   log.Println("Received answer from " + remote)
   var encodedAnswer = m.Answer

   answer := webrtc.SessionDescription{}

   internal.Decode(encodedAnswer, &answer)

   // Checking remote certificate's fingerprint matches given passphrase
   parsed := &sdp.SessionDescription{}
   if err := parsed.Unmarshal([]byte(answer.SDP)); err != nil {
      panic(err)
   }
   fingerprint := internal.ExtractFingerprint(parsed)
   remotePassphrase := internal.FingerprintToPhrase(fingerprint)

   // If certificate matches, set as remote description
   if remotePassphrase == remote {
      fmt.Println("Receiver identity confirmed!")
      err = peerConnection.SetRemoteDescription(answer)
      if err != nil {
         panic(err)
      }
   } else {
      fmt.Println("Receiver's certificate is not matching")
      break
   }
```

Code 25: authenticity check on sender's side

Obviously, the same procedure is done receiver’s side upon offer reception.

4.5.3 Complete protocol

The following diagram (see Fig. 26) summarises the whole process of signaling, using inverted passphrases as usernames and doing a passphrase check during both the offer and the answer's reception.

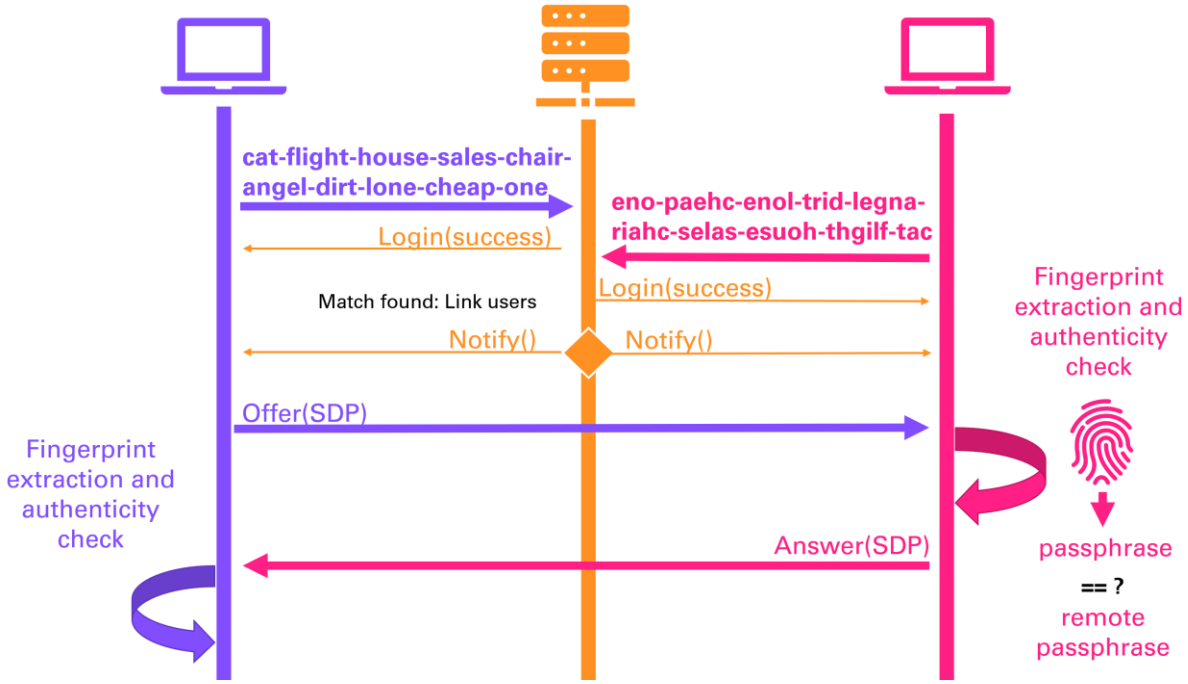

Figure 26: Full Signaling Protocol

There are various advantages of using such a protocol.

Matching inverted usernames is a guarantee of always having a peer-to-peer relation, as there is only one possible match for each user. This also makes the signaling server's code rather simple and lighter. Any error or disconnection during any step of the process is easily handled, the server only having to notify the linked user.

As for the fingerprint extraction and passphrase derivation, it makes for a perfectly secure way of authentication. While it does theoretically not always prevent a malicious user to send us an SDP, it will detect it as not matching the expected passphrase weather it is an offer or answer.

The only possible flaw would be for a third user to generate a certificate which, by chance, derives to the same passphrase as our receiver within the interval of time between the sender and receiver's connections to the signaling server.

The resulting chances of opening a **DataChannel** with the wrong user is therefore brought down to almost zero.

5 Conclusion

5.1 Issues and further improvement

Throughout this document, all the principal aspects and core functionalities were discussed. The final application does work, and the product is easily usable. However, many further improvements can be implemented, and some small issues still need to be taken care of.

Starting with the remaining issues, we must mention the unhandled errors. Throughout the **sender.go** and **receiver.go** different protocols, many **panic(err)** can still be found. Despite a top-level error handler being implemented, and therefore dealing with any of these panic error throwers, each case should be treated individually to get a better feedback on these errors. This top-level handler should be considered a temporary workaround and not a definitive solution.

Another unhandled issue at the time is the possibility of being stuck at the login step, during the signaling protocol. This happens if any side does not correctly enter the remote user's passphrase. The signaling server finding no match, both users are left waiting indefinitely for a "linked" notification. This issue can be solved rather easily either by setting a timeout server's side, or by giving the user the possibility to exit login and type the remote passphrase again by setting a quit/retry command.

A final point concerning the code issues is the hardcoded values used for different steps. The critical one is the TURN server credentials, which remain plainly visible in the code. For ease of use, other values should be left to chose by the user, such as the file output directory, currently set to "/out/" as well as the STUN and TURN servers' locations. Theses last points could be handled using command line parameters.

On the improvement's side, efficiency while using the program can still be widely increased. The first thing to implement would be bidirectional file exchange. At this point of development, users are stuck using precise roles, and switching from sender to receiver, though it can be done, implies going through the whole passphrase exchange and signaling step over again. Due to the architecture of the current code, making this role swap is a hard task and requires a lot of changes in both the signaling and the file exchange protocol.

An easier to implement feature which would drastically increase efficiency would be the possibility of recursive file sending. In its current state, the program only allows for one file transfer at a time, and the ability to send multiples files or full directory hierarchies would be a game changer in usability.

5.2 Personal conclusion

This project has allowed me to practice and apply a lot of the knowledges I acquired during my university path. The use of the WebRTC or Go language were totally foreign to me and getting to know them taught me not only about them as a new skill, but also to have confidence with starting a project which has a lot of unknown components.

A major lesson I learned doing this program is the crucial importance of planning the whole code's structure. If I were to redo the whole project from scratch, knowing the limitations of my current design choices, I would go with a single class sender/receiver and use a state mechanism to define the roles. In a professional context, the long-term cost of such bad design choices would be huge, not to mention starting over may be the only solution for some further features' implementation.

Discovering the WebRTC technology really brought me a lot and I will probably use it for other incoming projects. The same can be said about the Go programming language, and I can already profit from this new knowledge in some of my master's courses.

Mutual authentication, a big part of this project, was the most interesting aspect to me. Though it was not clear at first, doing research about it has brought me to learn about a wide variety of ways to authenticate someone and I surely will apply this knowledge further.

The final application is working and though there is still a lot of possible improvement, I am pleased and proud of the result.

Referenced Web Resources

[1] 262588213843476. n.d. 'Golang Split Byte Slice in Chunks Sized by Limit'. Gist. Accessed 2 September 2020. https://gist.github.com/xlab/6e204ef96b4433a697b3.

[2] Baché, Antoine. (2019) 2020. *Antonito/Gfile*. Go. https://github.com/Antonito/gfile.

[3] 'Bienvenue sur flying-dut.ch'. n.d. Accessed 2 September 2020. http://flying-dut.ch/index.html.

[4] Bradner, S. 1997. 'Key Words for Use in RFCs to Indicate Requirement Levels'. RFC2119. RFC Editor. https://doi.org/10.17487/rfc2119.

[5] 'Convert between Byte Array/Slice and String'. n.d. Accessed 2 September 2020. https://yourbasic.org/golang/convert-string-to-byte-slice/.

[6] Cooper, D., S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. 2008. 'Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile'. RFC5280. RFC Editor. https://doi.org/10.17487/rfc5280.

[7] *Coturn/Coturn*. (2015) 2020. C. coturn. https://github.com/coturn/coturn.

[8] *Coturn/Rfc5766-Turn-Server*. (2015) 2020. C. coturn. https://github.com/coturn/rfc5766-turn-server.

[9] Dang, Quynh H. 2015. 'Secure Hash Standard'. NIST FIPS 180-4. National Institute of Standards and Technology. https://doi.org/10.6028/NIST.FIPS.180-4.

[10] Dierks, T., and E. Rescorla. 2008. 'The Transport Layer Security (TLS) Protocol Version 1.2'. RFC5246. RFC Editor. https://doi.org/10.17487/rfc5246.

[11] Eastlake, D. 2011. 'Transport Layer Security (TLS) Extensions: Extension Definitions'. RFC6066. RFC Editor. https://doi.org/10.17487/rfc6066.

[12] 'Emad-Elsaid/Inbox'. n.d. GitHub. Accessed 2 September 2020. https://github.com/emad-elsaid/inbox.

[13] 'Exception Handling - Catching Panics in Golang'. n.d. Stack Overflow. Accessed 15 November 2020. https://stackoverflow.com/questions/25025467/catching-panics-in-golang.

[14] Georgiev, Martin, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. 2012. 'The Most Dangerous Code in the World: Validating SSL Certificates in Non-Browser Software'. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security - CCS '12*, 38. Raleigh, North Carolina, USA: ACM Press. https://doi.org/10.1145/2382196.2382204.

[15] 'Get File Name, Size, Permission Bits, Mode, Modified Time in Go (Golang)'. 2020. *Welcome To Golang By Example* (blog). 16 April 2020. https://golangbyexample.com/file-info-golang/.

[16] 'Go'. n.d. GitHub. Accessed 11 March 2021. https://github.com/golang.

[17] 'Go - Environment Setup - Tutorialspoint'. n.d. Accessed 12 February 2021. https://www.tutorialspoint.com/go/go_environment.htm.

[18] 'Go - Golang "Undefined" Function Declared in Another File? - Stack Overflow'. n.d. Accessed 2 September 2020. https://stackoverflow.com/questions/28153203/golang-undefined-function-declared-in-another-file.

[19] 'Go - How Do I Configure Goland to Recognize "mod" Packages?' n.d. Stack Overflow. Accessed 2 September 2020. https://stackoverflow.com/questions/51910862/how-do-i-configure-goland-to-recognize-mod-packages.

[20] 'Golang Panic and Recover Tutorial with Examples | Golangbot.Com'. 2020. Go Tutorial - Learn Go from the Basics with Code Examples. 8 July 2020. https://golangbot.com/panic-and-recover/.

[21] 'Google Code Archive - Long-Term Storage for Google Code Project Hosting.' n.d. Accessed 30 December 2020. https://code.google.com/archive/p/rfc5766-turn-server/.

[22] Hardt, D. 2012. 'The OAuth 2.0 Authorization Framework'. RFC6749. RFC Editor. https://doi.org/10.17487/rfc6749.

[23] 'How Can I Use Go Append with Two []Byte Slices or Arrays?' n.d. Stack Overflow. Accessed 2 September 2020. https://stackoverflow.com/questions/8461462/how-can-i-use-go-append-with-two-byte-slices-or-arrays.

[24] 'How To Build Go Executables for Multiple Platforms on Ubuntu 16.04'. n.d. DigitalOcean. Accessed 12 February 2021. https://www.digitalocean.com/community/tutorials/how-to-build-go-executables-for-multiple-platforms-on-ubuntu-16-04.

[25] 'How to Create and Configure Your Own STUN/TURN Server with Coturn in Ubuntu 18.04 | Our Code World'. n.d. Accessed 2 September 2020. https://ourcodeworld.com/articles/read/1175/how-to-create-and-configure-your-own-stun-turn-server-with-coturn-in-ubuntu-18-04.

[26] 'Ioutil - The Go Programming Language'. n.d. Accessed 2 September 2020. https://golang.org/pkg/io/ioutil/#WriteFile.

[27] Jones, M. 2015a. 'JSON Web Algorithms (JWA)'. RFC7518. RFC Editor. https://doi.org/10.17487/RFC7518.

[28] Jones, M., J. Bradley, and N. Sakimura. 2015. 'JSON Web Token (JWT)'. RFC7519. RFC Editor. https://doi.org/10.17487/RFC7519.

[29] Jones, M., J. Bradley, and H. Tschofenig. 2016. 'Proof-of-Possession Key Semantics for JSON Web Tokens (JWTs)'. RFC7800. RFC Editor. https://doi.org/10.17487/RFC7800.

[30] Jones, M., and D. Hardt. 2012. 'The OAuth 2.0 Authorization Framework: Bearer Token Usage'. RFC6750. RFC Editor. https://doi.org/10.17487/rfc6750.

[31] Jones, M., N. Sakimura, and J. Bradley. 2018. 'OAuth 2.0 Authorization Server Metadata'. RFC8414. RFC Editor. https://doi.org/10.17487/RFC8414.

[32] Josefsson, S. 2006. 'The Base16, Base32, and Base64 Data Encodings'. RFC4648. RFC Editor. https://doi.org/10.17487/rfc4648.

[33] 'JSON and Go - The Go Blog'. n.d. Accessed 2 September 2020. https://blog.golang.org/json.

[34] Kawamura, S., and M. Kawashima. 2010. 'A Recommendation for IPv6 Address Text Representation'. RFC5952. RFC Editor. https://doi.org/10.17487/rfc5952.

[35] 'Learn Go | Codecademy'. n.d. Accessed 2 September 2020. https://www.codecademy.com/courses/learn-go/lessons/learn-go-fmt-package/exercises/getting-user-input.

[36] 'Learn to Create and Use Go Packages - Golangbot.Com'. 2020. Go Tutorial - Learn Go from the Basics with Code Examples. 16 February 2020. https://golangbot.com/go-packages/.

[37] Legg, S. 2006. 'Lightweight Directory Access Protocol (LDAP): Syntaxes and Matching Rules'. RFC4517. RFC Editor. https://doi.org/10.17487/rfc4517.

[38] Leiba, B. 2017. 'RFC 2119 Key Words: Clarifying the Use of Capitalization'. RFC8174. RFC Editor. https://doi.org/10.17487/RFC8174.

[39] Lodderstedt, T., S. Dronia, and M. Scurtescu. 2013. 'OAuth 2.0 Token Revocation'. RFC7009. RFC Editor. https://doi.org/10.17487/rfc7009.

[40] Node.js. n.d. 'Download'. Node.Js. Accessed 10 February 2021. https://nodejs.org/en/download/.

[41] 'Pion'. n.d. GitHub. Accessed 2 September 2020a. https://github.com/pion.

[42] *Pion/Signaler*. (2018) 2020. Go. Pion. https://github.com/pion/signaler.

[43] 'Pion/Webrtc'. n.d. GitHub. Accessed 2 September 2020a. https://github.com/pion/webrtc.

[44] Rescorla, E. 2018. 'The Transport Layer Security (TLS) Protocol Version 1.3'. RFC8446. RFC Editor. https://doi.org/10.17487/RFC8446.

[45] Richer, J. 2015. 'OAuth 2.0 Token Introspection'. RFC7662. RFC Editor. https://doi.org/10.17487/RFC7662.

[46] Richer, J., M. Jones, J. Bradley, M. Machulak, and P. Hunt. 2015. 'OAuth 2.0 Dynamic Client Registration Protocol'. RFC7591. RFC Editor. https://doi.org/10.17487/RFC7591.

[47] 'RTCIceServer'. n.d. MDN Web Docs. Accessed 2 September 2020. https://developer.mozilla.org/en-US/docs/Web/API/RTCIceServer.

[48] 'Scp - Tranférer En SSH Des Fichiers/Répertoires Entre Des Machines - Www.Octetmalin.Net'. n.d. Accessed 2 September 2020. http://www.octetmalin.net/linux/tutoriels/scp-transferer-donnees-data-ssh-envoyer-telecharger-fichiers-files-machines-serveurs-clients.php.

[49] 'Security - WebRTC Mutual Authentication Using Certificates'. n.d. Stack Overflow. Accessed 2 September 2020. https://stackoverflow.com/questions/60922417/webrtc-mutual-authentication-using-certificates.

[50] shinde, sachin. (2018) 2020. *SacOO7/GoWebsocket*. Go. https://github.com/sacOO7/GoWebsocket.

[51] 'Split/Slice an Array into Chunks (Golang) - DEV'. n.d. Accessed 2 September 2020. https://dev.to/jinagamvasubabu/split-array-into-chunks-in-golang-40n.

[52] Sufitchi, Filip. 2020. 'Go Environment Setup in 2020: Modules!' Medium. 25 February 2020. https://medium.com/@fsufitch/go-environment-setup-in-2020-modules-3ed980bc287e.

[53] 'Temasys | ICE and WebRTC: What Is This Sorcery? We Explain…'. 2016. Temasys.Io. 19 August 2016. https://temasys.io/webrtc-ice-sorcery/.

[54] 'The Go Playground'. n.d. Accessed 2 September 2020. https://play.golang.org/p/Ou2UE3pApi.

[55] 'TheWarWolf/FlyingDutchman'. n.d. GitHub. Accessed 2 September 2020. https://github.com/TheWarWolf/FlyingDutchman.

[56] 'Trickle ICE'. n.d. Accessed 2 September 2020. https://webrtc.github.io/samples/src/content/peerconnection/trickle-ice/.

[57] 'WebRTC - RTCPeerConnection APIs - Tutorialspoint'. n.d. Accessed 2 September 2020. https://www.tutorialspoint.com/webrtc/webrtc_rtcpeerconnection_apis.htm.

[58] ‘WebRTC - Sending Messages - Tutorialspoint’. n.d. Accessed 2 September 2020. https://www.tutorialspoint.com/webrtc/webrtc_sending_messages.htm.

[59] ‘WebRTC - Signaling - Tutorialspoint’. n.d. Accessed 2 September 2020. https://www.tutorialspoint.com/webrtc/webrtc_signaling.htm.

[60] ‘Websocket Server in Node.Js’. n.d. Mastering JS. Accessed 10 February 2021. https://masteringjs.io/tutorials/node/websocket-server.

[61] ‘Why Is GO111MODULE Everywhere, and Everything about Go Modules - DEV’. n.d. Accessed 2 September 2020. https://dev.to/maelvls/why-is-go111module-everywhere-and-everything-about-go-modules-24k#the-raw-go111module-endraw-environment-variable.

[62] ‘Ws’. n.d. Npm. Accessed 10 February 2021. https://www.npmjs.com/package/ws.

[63] ‘Xirsys Says…’. n.d. Accessed 2 September 2020. https://global.xirsys.net/dashboard/signin.

[64] ‘Xirsys TURN Server Cloud - User Documentation’. n.d. Accessed 2 September 2020. https://docs.xirsys.com/?pg=api-turn.

[65] Zeilenga, K. 2006. ‘Lightweight Directory Access Protocol (LDAP): String Representation of Distinguished Names’. RFC4514. RFC Editor. https://doi.org/10.17487/rfc4514.

UNI
FR

UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Faculté des sciences économiques et sociales
Wirtschafts- und sozialwissenschaftliche Fakultät
Boulevard de Pérolles 90
CH-1700 Fribourg

DECLARATION

Par ma signature, j'atteste avoir rédigé personnellement ce travail écrit et n'avoir utilisé que les sources et moyens autorisés, et mentionné comme telles les citations et paraphrases.

J'ai pris connaissance de la décision du Conseil de Faculté du 09.11.2004 l'autorisant à me retirer le titre conféré sur la base du présent travail dans le cas où ma déclaration ne correspondrait pas à la vérité.

De plus, je déclare que ce travail ou des parties qui le composent, n'ont encore jamais été soumis sous cette forme comme épreuve à valider, conformément à la décision du Conseil de Faculté du 18.11.2013.

Granges, le 27.04. 2021

(signature)