Live video streaming

Robin Moussu 2A SLE

internship of 2nd year as engineer assistant

15 June – 15 September
3 months

tutor – Olivier Kilh
supervised by – Steven Durand
Ensimag tutor – Roland Groz

Smart Me Up
4 chemin des prés
Meylan

I want to thank warmly M. Steven Durand (CTO) and M. Loïc Lecerf (CEO) for guiding me during my internship, together with M. Olivier Kilh my tutor.

My thanks also goes to all the employs and interns of Smart Me Up, specially Maude Premillieu and Gabriel Mattos Langeloh. It was really a pleasant experience to work with them.

1. Introduction

My internship have been done in Smart Me Up, a start-up specialized in facial recognition, from June 15th to September 15th. My tutor was Olivier Kilh, a researcher in facial recognition. During my internship, I had many contacts with Steven Durand, the CTO, and Loïc Lecerf, the CEO. I have worked with Gabriel Mattos Langeloh and Maude Premillieu, two other interns.

Smart Me Up is a start-up specialized in video recognition, more precisely in facial recognition. Its employees have developed a library that is able to make real-time facial analysis. My goal during my internship was to create a software. This one will be installed on Smart Me Up's client hardware (**Linux**, **Windows** or **Android**), then it will grab a video, and send the stream to Smart Me Up's server. Gabriel has developed the server side of that project, and Maud the web interface were the client can manage their subscriptions.

First I will present Smart Me Up and its employees. Then I will introduce the goal of my internship, and the choices I have made. Next, I will describe the methodology I have used during my internship. Afterward I will present the results of my work and finally conclude.

Key points

- **video capture**
- **video compression**
- **stream protocols (rtp, rtcp)**
- **C++**
- **real time**
- **cross-compilation**

Contents

2. Presentation

2.1 The company

Smart Me Up[1] is a start up from Meylan which has created a library for facial analysis. That solution is usable as a base that can be embedded is any device, like drone, robots, interactive advertisements, ...

Smart Me Up was found in February 2012 by Loïc Lecerf. He has a PhD in artificial intelligence. The initial founding for his start-up came from the selling of a website he had previously created.

The first clients of Smart Me Up are Netatmo (for home automation) and Photomaton (to ensure conformity of official photos). After finding those customers, three new members has been hired: Steven Durand (polytecthnicien), Olivier Kihl and John Ruz Hernandez (doctors in artificial intelligence and image recognition). A new office has been created in Paris. The revenue of Smart Me Up in 2014 was 225'000€.

In January 2015, Smart Me Up has participated in the CES (Consumer Electronics Show) in Las Vegas. That event has increase it popularity, and new clients have been found, including SNCF. Smart Me Up was as well present at the **French Tech**[2] (an organization that promote the french start-up). Nowadays, Smart Me Up want to make a fundraising of 1 up to 2 millions euros.

Figure 2.1: French Tech's logo

.

2.2 The employees

The organization chart is presented on the figure 2.2.

Smart Me Up is a start-up of seven employees. Loïc Lecerf is the PDG and founder, Matthieu Marquenet is the commercial director, and Jessica Lecerf is the administrative director. For R&D, the CTO is Steven Durand, and three researchers work under his orders: Oliver Kihl, John Ruiz Hernandez and Enguerrand Quilliard. As soon as they will obtain their fundraising, Smart Me Up will hire new permanents.

[1] http://www.smartmeup.org/
[2] http://www.lafrenchtech.com/la-french-tech

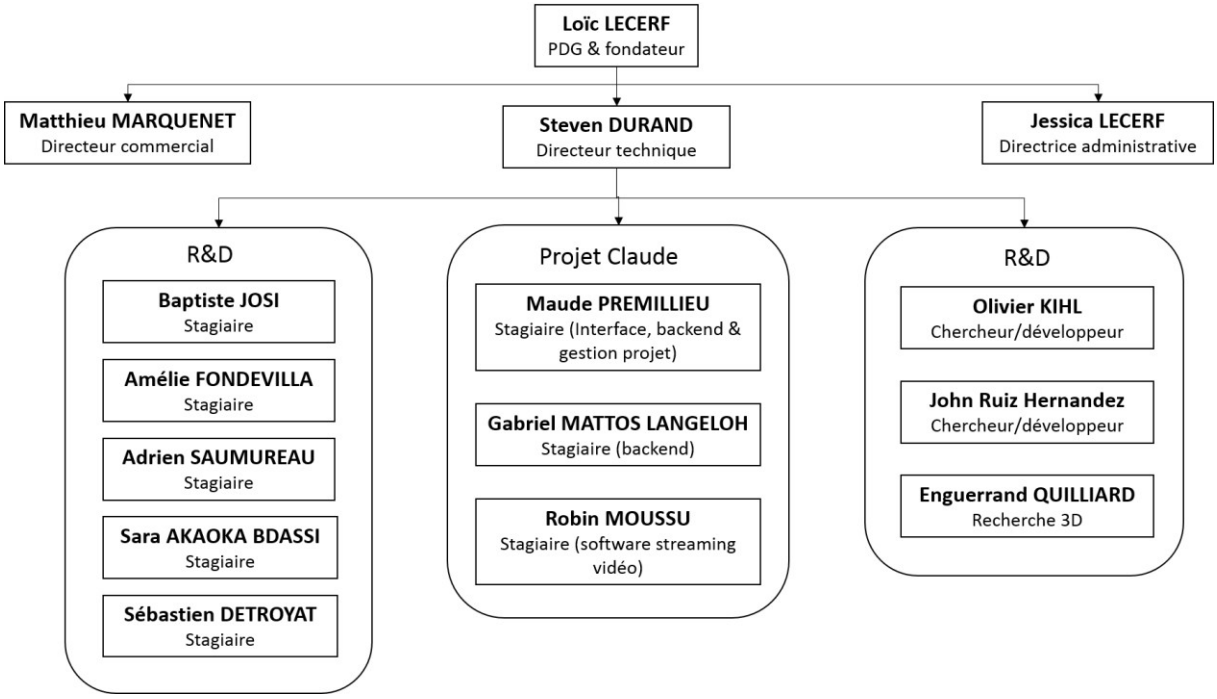


Figure 2.2: Organization chart of Smart Me Up (summer 2015)

.

During this summer, 7 interns has worked in Smart Me Up. Amélie Fondevilla, Adrien Saumureau, Baptiste Josi, Sara Akaoka Bdassi and Sébastien Detroyat have work on specific project. Gabriel Mattos Langeloh, Maude Premillieu and myself have work on the *Claude* project.

3. Goals of the internship

3.1 The *Claude* project

I have worked together with Gabriel and Maud, two others interns. Our goal was to grab videos from customers' cameras, and to analyze them on Smart Me Up's server (deployed on Amazon[1] and ovh[2]'s cloud). At Smart Me Up, we were the *Claude* department (it's a pun with *cloud computing* with a big french accent :-)). During our internship, we worked in the same office, so we were able to exchange a lot.

Even if Oliver Kilh was my tutor, in facts it was Steven Durand and Loïc Lecerf who supervised me and the others interns of the *Claude* project.

My initial goal was to create a software that can stream a video from all type of devices to the servers of Smart Me Up, where it will be analyzed. To be more specific, I had to support all type of webcam, on **Linux**, **Windows** and **Mac**, smartphones (**Android**, **iOS**), and ip cameras.

Gabriel goal was to create the software used on the server. Those softwares have to receive the video stream, and analyze them using the library developed by Smart Me Up.

Maud has created the web interface. That website is the place where customer can access to the results of their analysis and where they can administrate their cameras (the clients have to pay a subscription for all their camera, and the type of analysis they have chosen).

3.2 Technological choices

Our goal (Gabriel, Maud and me) was to select the appropriate technology for what we have to do. To be more specific:

- How to capture the images
- How to encode them

[1] https://aws.amazon.com/
[2] https://www.ovh.com/us/

- How to send them to SMART ME UP server.

And furthermore

- How to establish connection between clients' software and SMART ME UP servers

Our first task was to learn how video are streamed in general. That part is explained in details later (in the section 4.2). The objective was to found witch protocols are used for grabbing, encoding and sending images. Of course it was not in our plan to reinvent the well, so we also had to choose a library that implement those protocols.

Our second goal was to define how ours softwares will exchange informations. Since clients machine, and SMART ME UP server are all links to internet, we have chosen to use http request. We have defined all messages type, what we can send, and their structure. The methodology and the result is explained in the section 4.2.3.

3.3 Platforms

As I said, my personal goal was to write a software that could be run virtually everywhere. To be more specific, my initial targets were **Linux** (on PC, and embedded devices like **raspberry pi** or **beaglebone black**), **Windows**, **Android**, **Mac** and **iOS**. It was a bit too ambitious, so the last two were dropped.

All employees in SMART ME UP work on Ubuntu (a **Linux** distribution), so I have first developed my solution for Linux (x86_64), keeping in mind that I will have to port it after, so I never use platform specific stuff. Then I port it to Linux arm (I have tested it on **raspberry pi** and **beaglebone black**). After that, I try to create an **Android** application, and finally for **Windows**.

I was requested to write my software in C++. As a consequence of using native language, my build system has to be able to create executable for all platforms. I have chosen to use **cmake** as build system, since **cmake** was already use in the company.

3.4 Synthesis

Since I have to support a very heterogeneous set of platforms, I had to be very generic.

The main complicated part was to make tradeoff between performance required for video encoding and the available bandwith. To lower the required bandwith, we can use a better compression algorithm, however better algorithm mean higher compression time, so worse latency, and higher CPU usage. On slow CPU hardware (smartphones and embedded devices like **raspberry pi** and **beaglebone black**) it was a real challenge to found the right algorithm.

The results of the analysis of our needs for the *Claude* project have lead to the software organization presented on figure 3.4. That figure is extracted from the documentation Gabriel wrote. To be more clear, what I will later call *http gateway* is the *serveur HTTP*. The *source* is anything with a webcam (a PC, an embedded device, or a smartphone).

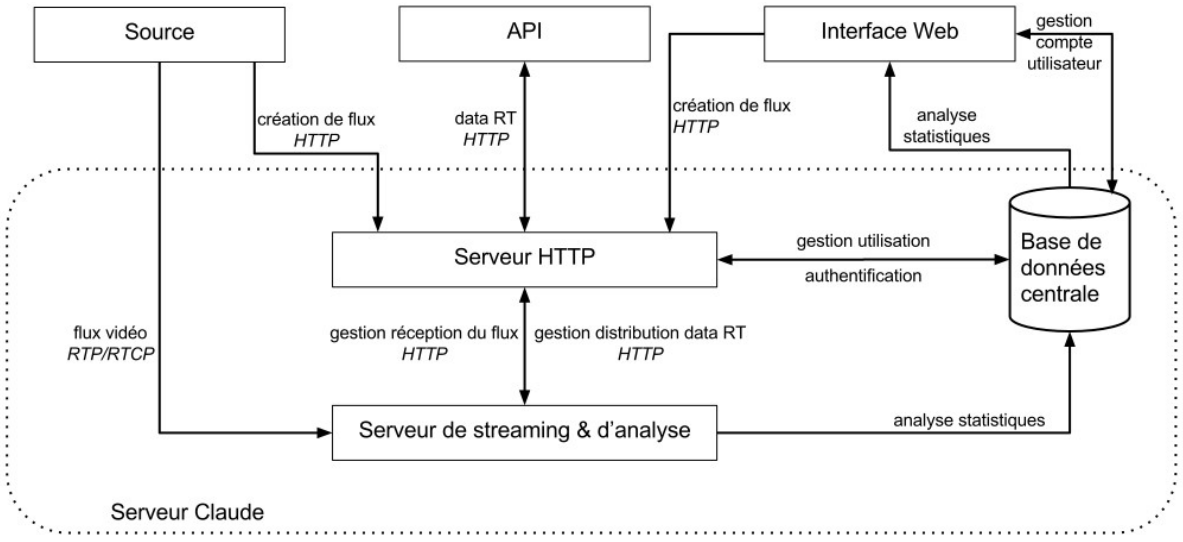


Figure 3.1: presentation of the *Claude* project structure

.

My contribution on the project was the source, and the specification of the communications between HTTP gateway and streaming & analysis server. Gabriel have worked on the video & analysis server, and Maud on the rest (her internship was longer).

4. Methodology and alternatives

4.1 Organisation

During all my internship, I had two small meeting per week (called *point claude*) with Steven Durand and Loïc Lecerf. The others intern of the *Claude* project (Maude and Gabriel) were also there. During those meeting, we presented the work we had done since the last time, and then we decide what to do next. Since none of the employees of Smart Me Up have look our code during our internship (as least for the *Claude* project), it was their only way to know what we were doing.

Our general methodology was to quickly have something that work somewhere, and then improve it by small iteration to make it work everywhere in all conditions. As a consequence, we first tested something on our machine, then using multiple machines in local network, and finally on the cloud.

4.2 Technologies used

The main challenge of that internship was to found the right technology to achieve the goal that I have been entrusted.

4.2.1 Network protocol

Firstly, we (Gabriel and me) have worked together to find the appropriate protocol for video streaming. We have concluded that **rtp** is a good choice. That was motivated because **rtp** can be easily replaced by **rtsp** that work similarly, but with encryption. Furthermore, it can be use in tandem with **rtcp**, witch carry information on the stream (like framerate, lag, and others useful information to analyze the quality of the stream).

Select the right protocol is of course not enough, and since we have planed to support multiples platform, we have search implementations that match this requirement. **ffmpeg** (and his fork **avconv**), **vlc** and **gstreamer** where good candidates. **vlc** was quickly avoided, because it has too much code that we do not need. It was nevertheless useful for testing purpose. In July, we have used **ffmpeg**/**avconv**, but we finally found that **gstreamer** as better performance results.

4.2.2 Video encoding

Having the right network protocol is still not enough. We had to choose an encoding that fit our needs. What we want is something fast to encode, because the client program can run on small hardware, and because low latency was unavoidable. The resulted stream has also to be lightweight, because we could not assume that SMART ME UP's client will have a fast internet connection.

All our tests have been first made on LAN, so **mjpeg** was a very good candidate. It's just a sequence of **jpeg** images, so it is really, really fast to encode it, even on small hardware. As a direct consequence, the delay was excellent. However, when we have made the firsts real tests using a server somewhere on the cloud, we have discovered that the amount of data needed for the transmission was too high. So we had to choose something else less eager in bandwith. We have tested **h264** and **vp8** encoding, and we have chosen the later.

4.2.3 Communication between devices

At this point we knew how to grab images, encode them, and send them to a server. It's cool, but we need something to ask the permission to the server. We choose to simply use http request. **curl** was a good initial choice, since it is a very powerful library. However, in the perspective of minimizing dependencies, I finally use **HappyHTTP**. The request are formatting is **json**.

The documentation can be found is the appendix A

4.3 Minimize the dependency

One of my objective was to create a software whose source code is really portable. For making that task easier, my tutor ask me to minimize the number of dependencies, and their complexities.

4.3.1 Prototyping

When I was prototyping, I have used some python and shell scripts to work faster. Since we cannot assume that those languages are installed on our client machine, my tutor ask me to use native code (to be more specific C++). Binary package are not transferable across operating system and cpu, so I had to generate binaries for all targets.

In July, I have written a software that did the work, although with too many dependencies. I had used **boost** for arguments and json parsing, and **curl** for http request. I have also used **gstreamer** for video (grab image, compress using **vp8** and send using **rtp**/**rtcp**). Only the last one was a required dependency.

4.3.2 Reducing complexity

Removing **boost** and **curl** was too complicated, so I re-write my software, but much faster than the first time, since I already know what I want to do, and how to do it.

To replace **boost**, I have written a minimal **json** parser (for the sub-part of json used for the communication between the client and the http gateway). To parse the arguments provided to the program, I have made a manual read of the arguments on `argv`. Of course my own code have fewer possibilities than **boost**, but it was less complex, and that was what I was asked for.

With the similar perspective, I have replaced **libcurl** by **HappyHTTP**. That second library is only one small source file, so it is easier to embed it. Furthermore, the employees of SMART ME UP were already using it. With some take back, I think that using **libcurl** during the prototyping was a good idea. Indeed, it had helped me to understand the errors easily when we (Gabriel and me) were creating the http protocol between the client software and the http gateway.

Despite everything my main dependency was either **gstreamer** or **ffmpeg**/**avconv**. Since they are both as complex, it was not an argument in favor of one or another. At the end, **gstreamer** (and its own dependencies) is the last remaining in my code, even if it is the most complicated.

4.4 Methodology

4.4.1 Solving problems

During my internship, I have wasted some time because at some point I was stuck, and I had not enough method to find the right solution in a short time. Making pair-programming was a good idea, but we did not use it enough for my taste.

Sometime, when I was stuck, I left my desk, to help someone else, and when I came back that person help me back. In general, that save us both a lot of time, because, it has helped us to take a more global view on what we were doing. The key point of that is that explaining to someone else our problem help us to formalize it.

4.4.2 Writing the documentation

During my internship, I have contributed to the writing of to type of documentation. The first is the documentation of the http protocol used for the communication. This one have been written gradually. An extract is presented in appendix A. The second one is the source code documentation. I have mainly written it at the end of my internship. **Doxygen**[1] have been used. It is a powerful documentation processor, commonly used.

[1] http://www.stack.nl/ dimitri/doxygen/

5. Results

5.1 Technologies used

5.1.1 Generality

I have written a software in C++ that grab image and send the stream via **rtp** to the server of Smart Me Up. The protocols used have been described previously (on part 4). In that section, I am going to describe the implementation details.

5.1.2 Build system

I have developed a build system based on **cmake**. It was chosen because it is cross platform (that was obviously required since I have to support many systems), and powerful enough to fit my need. Furthermore, **cmake** is also used on other projects in Smart Me Up.

Our initial goal was to have a cross-plateform build system. In that ideal scenario, when I want to create a new release of my software, I can create all the binaries from the same host. At the beginning of August, I had set up that type of build system. Like this, from my Ubuntu, I was able to build binaries not only for x86_64 **Linux**, but also for arm **Linux**.

However, when I have tried to statically link all the dependencies, cross-compilation has become too complicated to maintain. In effect, **gstreamer** has a lot of dependencies, including the **glib**. During the link process, everything need to be previously built for the destination architecture, so in this case, have to be previously cross-compiled. That took me too much time to maintain, and as a consequence I have stopped to develop it and I have come back to a more simple build system.

In addition to the **cmake** files, I have also written a small **python** script to call **cmake** more easily. I have developed it when I was trying to make cross compilation. So in one command, I build the whole program for all architectures. After the drop of the cross-compilation, it was just used as a helper.

The build process for **Linux**, on x86_64 and arm devices or for **Windows** is quite similar (except some few '#ifdef') thanks to **cmake**. On **Android** however, The main has to be written in java. So, I

```
string cmd { "gstrtpbin name=rtpbin autovideosrc"
        " ! videorate"
        " ! videoscale"
        " ! ffmpegcolorspace"
        " ! video/x-raw-yuv,width=" + video_width + ",height=" + video_height +
        " ! vp8enc speed=1 threads=2"
        " ! rtpvp8pay"
        " ! rtpbin.send_rtp_sink_0 rtpbin.send_rtp_src_0"
        " ! udpsink host=" + video_url + " port=" + rtp_port +
             " rtpbin.send_rtcp_src_0 "
        " ! udpsink host=" + video_url + " port=" + rtcp_port +
             " sync=false async=false udpsrc port=" + local_rtcp_port + " "
        " ! rtpbin.recv_rtcp_sink_0"};
auto pipeline = gst_parse_launch( cmd.c_str(), &error );
```

Figure 5.1: Extract of C++ code: gstreamer pipeline used for emission

.

have used **jni** to build my C++ codebase and linked it to java. Thus, I can reutilize my work across all systems.

5.1.3 Video stream

As I have explained, **gstreamer** have been chosen. It has lots of binding (including C++), but I finally choose to use the C api, since it have the best documentation. **Gstreamer** is pretty easy to use when you have understood some basics stuff. **Gstreamer** use a textual representation of the acquisition process, called a pipeline.

The listing 5.1.3 describe the command used to describe the **gstreamer** pipeline. It grabs a video using ‘autovideosrc‘. Then it set some attribute. After that the stream is encoding in **vp8** with ‘vp8enc‘ and some tuning for performance (‘speed=1 thread=2‘). And finally the stream is sent using **rtp**, and add **rtcp** information. This is an extract of the C++ code I have written. The easier way to test it, it to use the **bash** command **gst-launch** that used the same syntax.

Before using **gstreamer**, I have used **ffmpeg**/**avconv**. They work similarly (a C API, and a **bash** command). However, the syntax is not the same.

5.1.4 Android

Android software has to be in java. Since my work was done in C++, I have use **jni**, that allow me to call C++ code from java. **jni** use makefile, so I had to adapt my build process in consequence. My C++ code have to create a dynamic library, and a small C++ file make the glue between my code and the interface (in java).

The interface used on mobile has been design by Maud. A screenshot is presented on figure 5.2.

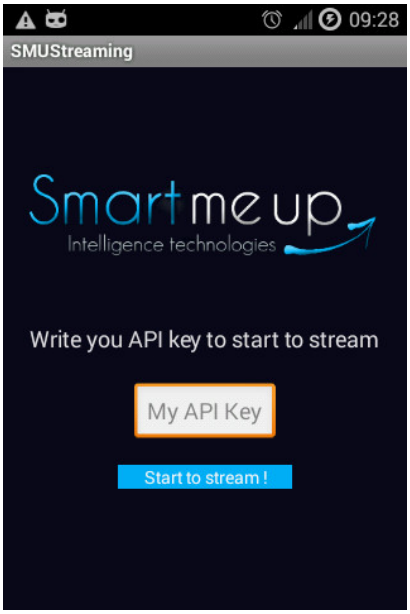


Figure 5.2: Mobile interface

I have also used the **Android** sdk and ndk as well as the gstreamer sdk. Using those software tools as allow me to easily start to port my software. However, some differences in the usage of **gstreamer** on **Android** have made the development more difficult, and I had not enough time to finish it.

5.1.5 Installer

In July, I have created an installer for linux, using **fpm**[1]. It is a software that can create **.deb** (Debian/Ubuntu package), **.rmp** (used on other Linux), **tar** package, ... I was really easy to do, and I will use it again for another project if need be.

However, in August, the objective of my internship have change a bit, and at this point, we had to create a demonstrator. Therefore, my software has to be as easy as possible to use, so removing the installation step was required. That was taking part of the clean process I have explained before (section 4.3.2).

5.2 Analyze results

5.2.1 Debugging

Debugging is an important step in software lifecycle. The main debug tool we have used with Gabriel is **logging**. We have both add a lot of debug informations, printed to **stdlog**. Each step can be identified, so we knew at any time what our program was doing. When something went wrong, we was able to watch

[1] https://github.com/jordansissel/fpm

our logs, and compared them. In this way, we was able to identify the problems and to determine if it was a client or a server side issue.

When my software was compiled for **Android**, I have used **adb** to print **stdlog** on my terminal. I was really easy to debug using the log. Thus, I was able to confirm each step one by one, and adapt my code quickly.

5.2.2 Performance

To measure the performance of our solution, Gabriel has written tools, that extract the following informations:

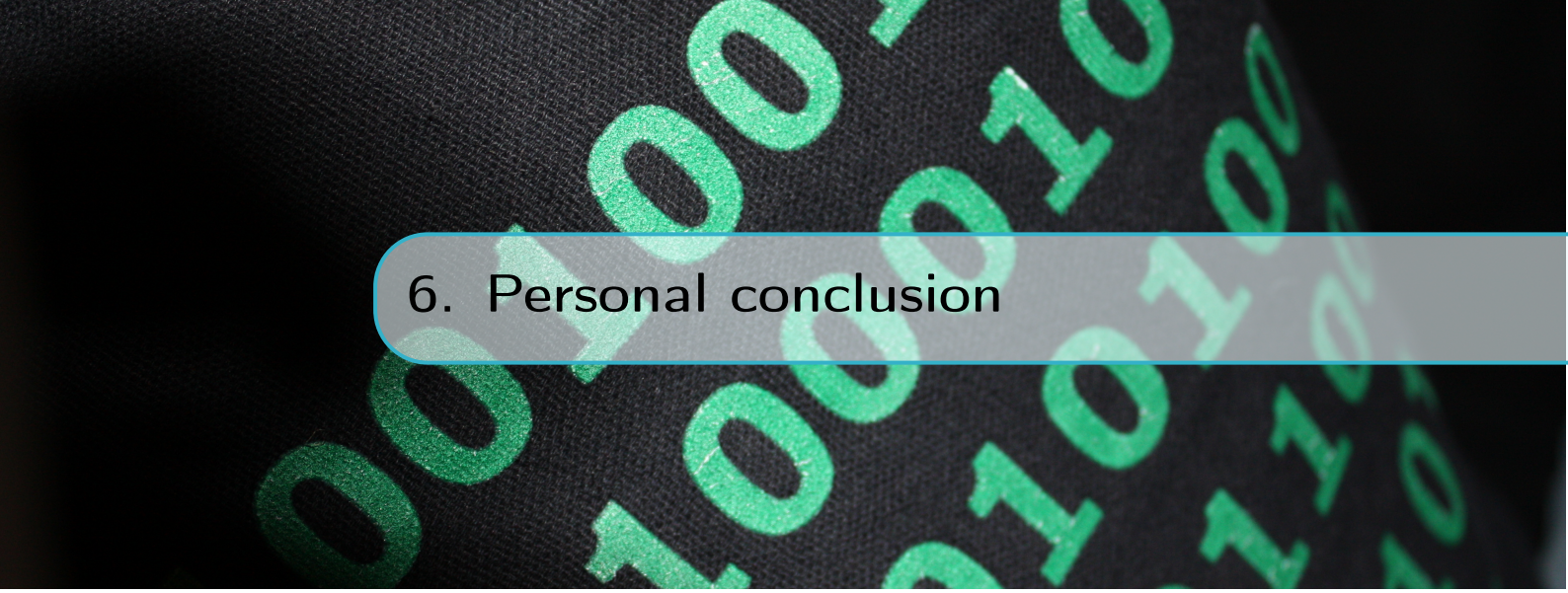
- number of image analyzed per second
- delay between the capture and the end of analyze
- many statistics that came from the result of the analysis
- if the stream is currently registered/active/analyzed
- ...

5.3 Conclusion

To sum up, at the end of my internship, I have written a software that can compile and run on multiple platforms : **Linux** (both x86_64 and arm) and **Windows**. A lot of work have also been done for **Android**, but not everything have been finished.

My program is able to ask the **htpp gateway** the authorization to start the stream. Then it grabs images, compress them using **vp8** encoding, and send them using a **rtp**/**rtcp** stream. All that is done using a gstreamer back end. The program I wrote still need more tuning to be faster, especially on embedded hardware.

All the code is documented using **Doxygen**, as well as the http protocol. All the important steps of the program are **logged** to stdlog to help to debug.



6. Personal conclusion

To sum up, the keys points of my internship were:

- design a software architecture
- define witch protocols will be used and how
- create a robust build system
- understand the link process
- how to build for multiple target (**Linux** arm and x86_64, **Windows**, **Android**)
- try to have cross compilation

The part that motivate me the most was designing the communication protocol. When designing a protocol, we have to take care of making it both expressive and easy to parse/understand. I found that the result is pretty clean, and it was a good experience to work with Gabriel and Maud under the direction of Steven Durand and Loïc Lecerf.

The major amount of work was the build process and its fine details. Building for one architecture is really easy, however when we want to support many more system, it becomes more complicated. **cmake** was really useful to simplify that task. What was too complicated for me was cross-compiling with many dependencies at the same time, using static link. Despite everything, I have learned a lot of that process.

To summarize, that internship was really instructive, and I feel that I have a finer understanding of the C++ build process.

Appendix

A. HTTP protocol documentation

This appendix is extracted from the documentation wrote by all the members of the *Claude* project.

source documentation

This page describes the HTTP requests sent by a video source to the HTTP server and their respective responses.

In all requests presented here, errors may occur - in this case, the response will be a JSON object of the following form:

```
{
    "error_code": ERROR_CODE,
    "error_txt": ERROR_TXT
}
```

where ERROR_CODE is an integer constant specified for each kind of error. For each request, we give the possible ERROR_CODEs.

Identification

get an utilization id

The **source must** ask the **http server** to get an utilization id:

```
{
    "action": "new_utilization_id",
    "key": API_KEY
}
```

The response will be:

```
{
    "utilization_id" : UTILIZATION_ID
}
```

Possible error codes: 4011

Stream creation

First the **source must** notify the **http server** that it wants to start a stream.

```
{
    "action": "start_stream",
    "key": API_KEY,
    "utilization_id": UTILIZATION_ID,
    "width": VIDEO_WIDTH,
    "height": VIDEO_HEIGHT
}
```

Then, the **http server must** respond with the following informations:

```
{
    "url": STREAMING_SERVER_URL,
    "rtp_port": RTP_STREAMING_PORT,
    "rtcp_port": RTCP_STREAMING_PORT
}
```

Possible error codes: 4013, 4091, 5031

At this point, the stream has to be started. When it is done, the **source has to** verify that the stream has been established:

```
{
    "action": "validate_stream",
    "utilization_id": UTILIZATION_ID
}
```

The response is :

```
{
    "validation_code": VALIDATION_CODE,
    "validation_txt": EXPLANATION_OF_STREAM_STATUS
}
```

Where the `VALIDATION_CODE` is 0 if the stream is successfully captured. A value of `VALIDATION_CODE` different from 0 means that the server failed to capture the stream.

Possible error codes: 5031

Stopping a stream

The source must send to the **http server** the following request:

```
{
    "action": "stop_stream"
    "utilization_id": UTILIZATION_ID,
}
```

The response is

```
{
    "ok": "stream stopped successfully"
}
```

Possible error codes: 4092, 5005, 5031

Error Message Formating

In case of failure, the http status code **must be** any of the error status, like `400: Bad Request`. The json **must** contain an `error_code` field to add more information on the failure:

```
{
    "error_code": INTEGER
}
```

If needed, the json **could** have a `error_txt` field to add text informations:

```
{
    "error_code": INTEGER
    "error_txt": MESSAGE
}
```

Error code

The error code are custom codes used to indicate the exact type of error. They are linked to the code status of the message with the first 3 digits.

Value	Description
4001	Missing arguments in request : no action argument
4002	Missing arguments in request : no key argument
4003	Missing arguments in request : no utilization_id argument
4004	Missing arguments in request : missing a video stream descriptor (width, height,key)
4005	Missing arguments in request : no data parameter
4006	Missing arguments in request: missing admin credentials or request arguments
4011	Identification Error : non valid API key
4012	Identification Error : non valid utilization identifier
4013	Identification Error : utilization identifier and API key not compatible
4051	Forbidden : unconfigured API key (missing expression)
4052	Forbidden : unconfigured utilization (missing compatible API key)
4053	Forbidden : invalid administrator credentials
4081	Request Timeout : stream not received before timeout
4091	Conflict : stream already active
4092	Conflict : stream not running
4093	Conflict : stream not analysed
4151	Unsupported Media Type : the codec of the stream is not supported
5001	Internal Server Error : cannot initialize stream listener
5002	Internal Server Error : cannot capture the stream (possible timeout)
5003	Internal Server Error : cannot process 'get_data'
5004	Internal Server Error : cannot update the stream status
5011	Not Implemented : unknown action
5031	Service Unavailable : cannot access to the video server