POLYTECH
GRENOBLE

Project Report

Sign2Speech

Edwin NIOGRET, Matthieu NOGUERON, Reatha TITH

Table of contents

I. Introduction

With **Sign2Speech**, we would like to offer a new means of communication between a signing person and someone who doesn't know the French language of signs. LSF is a language in its own, that doesn't only require hands. It actually requires the whole body, facial expression and personal interpretation.

This project is not aiming at translating the LSF as we just described it - the time and the technology that we currently have (**Intel's Real Sense camera**) are not efficient enough to allow us to aim for such a goal. Instead of that, we chose to implement an application able to recognize and translate basic words from the LSF, using a chat working with the WebRTC technology.

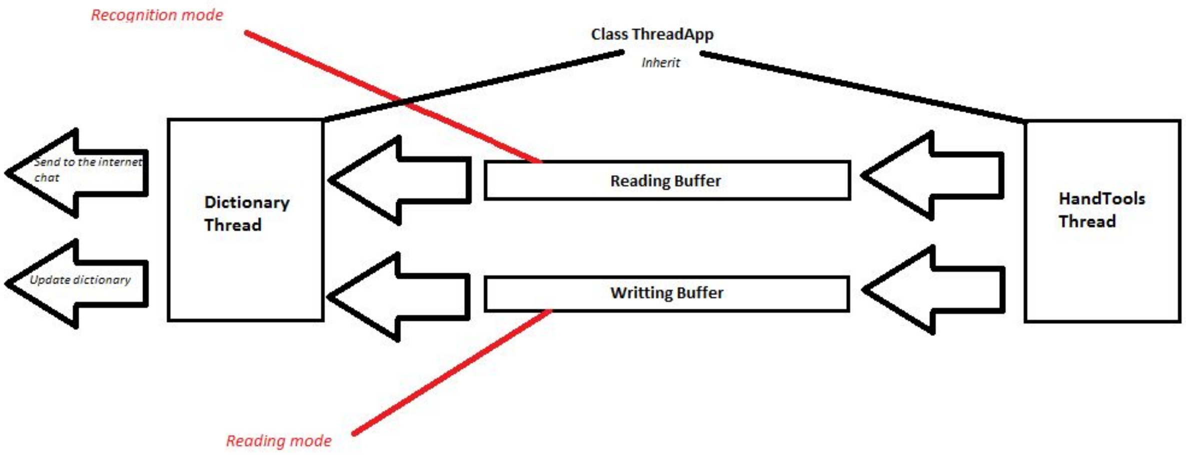
II. Gesture recognition

A. Application structure and overview

1. *Program flow*

The recognition software works with two threads:

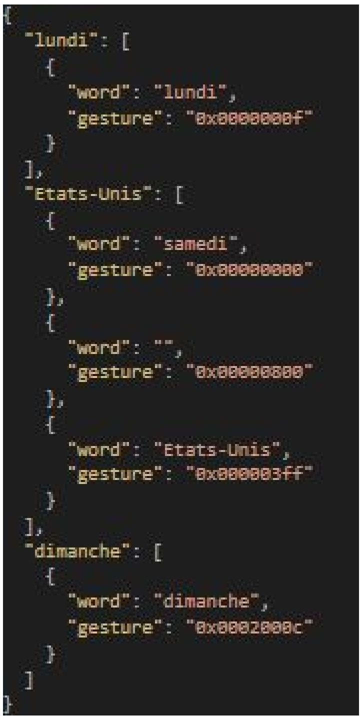
- The **"Dictionary" thread** is managing the connection between the data base and the internet chat/camera. It also updates the Dictionary when it is needed.
- The **"Handtools" thread** is the thread that works with the camera and that tries to recognize which gesture the user is currently doing and then sends it to the Dictionary thread.



When the application starts, it will open a console window and ask the user if he wants to add words to the Dictionary (it is the "**Learning Mode**") or if he wants to be in the "**Recognition Mode**". If the user chooses the learning mode, he will just have to follow instructions that will be printed in the console. Otherwise, for the Recognition Mode choice, the user will just have to do his gestures in front of the camera.

2. Data storage types

Sign2Speech is able to work with a database of gestures. This database is saved in a **JSON file** that will be read by a Parser and stored in a **Dictionary**. Then the application will work with the Dictionary. Once the program is about to be stopped, the JSON file will be updated with the new words that have been inserted into the Dictionary during his execution (if it was executed under the “learning mode”). In fact, it will make a new JSON file.



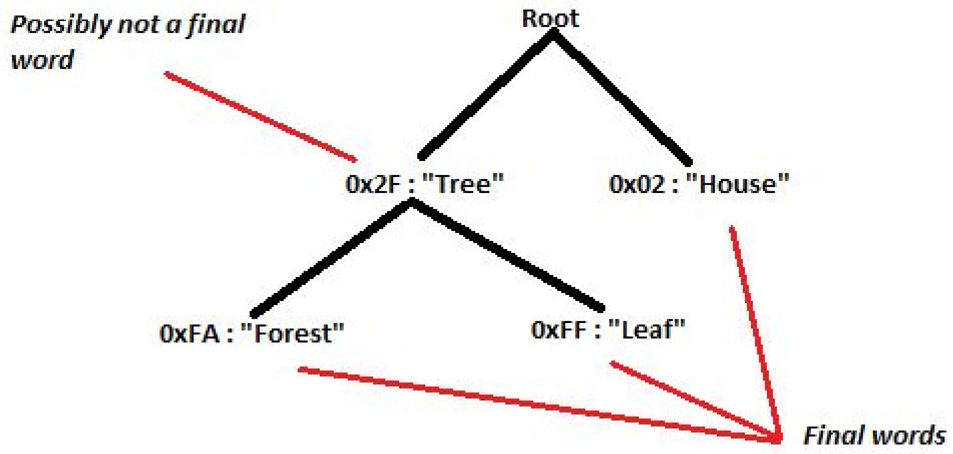
```
{
  "lundi": [
    {
      "word": "lundi",
      "gesture": "0x0000000f"
    }
  ],
  "Etats-Unis": [
    {
      "word": "samedi",
      "gesture": "0x00000000"
    },
    {
      "word": "",
      "gesture": "0x00000800"
    },
    {
      "word": "Etats-Unis",
      "gesture": "0x000003ff"
    }
  ],
  "dimanche": [
    {
      "word": "dimanche",
      "gesture": "0x0002000c"
    }
  ]
}
```

For our Dictionary, we decided to use a **n-tree structure** because it was a simple and an efficient way to read a serie of gestures.

Our Dictionary has a pointer on the current Node of its n-tree that is at this time read by the application. It means that the Dictionary loops through the n-tree each time the user does a new gesture. Our n-tree is made of a **Node Type**. Each Node have n children.

The signification of a series of gestures will be translated once the Dictionary reached **a leaf of the n-trees**, that means it reached a “**final word**”. It also can translate a serie of gestures that still have children in the n-trees if the user **stops doing gestures of if he does a gesture that is not a child of the current Node of the tree**.

<u>For</u> <u>example</u>: If you check the following image : If the user does “0x2F”, the system will not be sure that he wanted to write “Tree”. So the system will wait for his next gesture to know if he wanted to say “Forest”, “Leaf” or something else. If he does “0x02” after “0x2F”, the system is sure that he first wanted to say “Tree”. If the user doesn’t make a new gesture within 2s (reset time) after he first did “0x2F”, the system will be sure that the user also wanted to say “Tree”.



B. Features

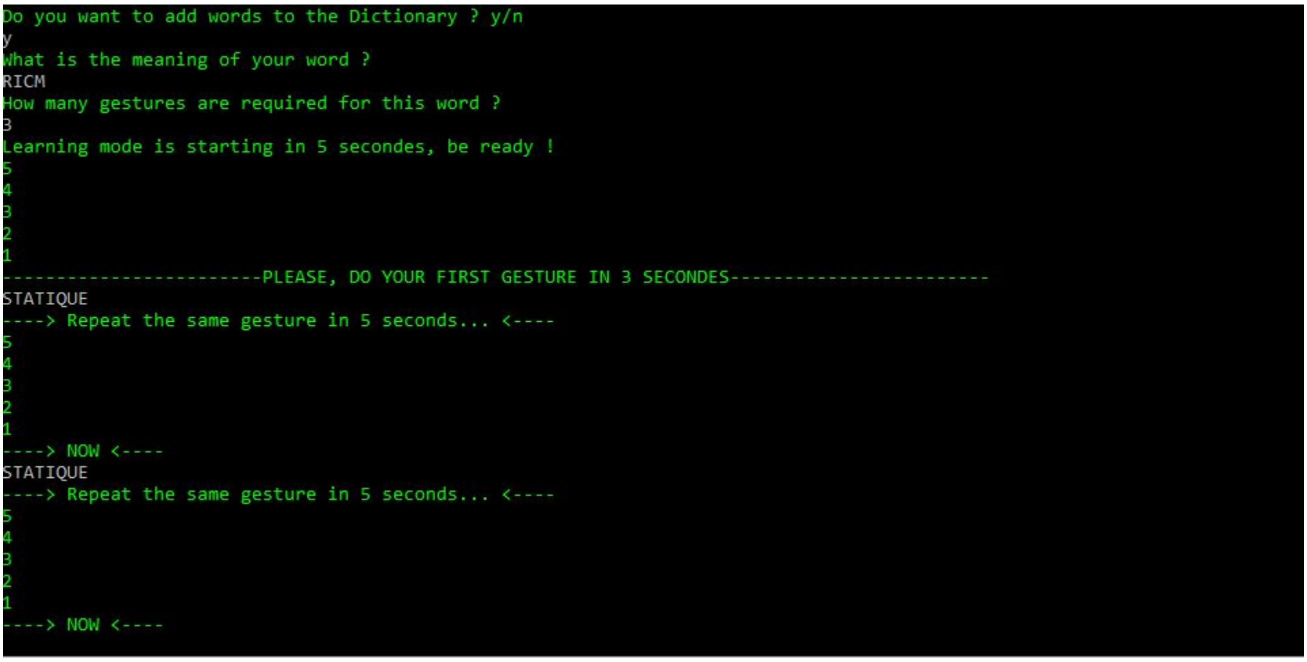
1. “Recognition mode”

This is the main mode of the software, which can recognize a set of gestures and translate it into a word. The SDK sends data to a thread which gathers all incoming frames in an array and when this array is full, it applies a treatment to reduce the error rate. During this step, an average of the frame array is calculated and encoded to compare with each words present in the dictionary. Then if the gesture is in the dictionary as a final word, the word is sent to the chat application thanks to a **WebSocket** channel. If the gesture is not a final word, it is kept in a vector, and then the next gesture will match the word which starts with all the gestures in the vector and finish with the new gesture. Each time a gesture is a part of a composed word, we move forward in the dictionary to the node associated to the current gesture, until we are not arrived to a leaf.

2. “Learning mode”

We decided to add a feature that allows the user to add new series of gestures and their significations to the Dictionary. The system will ask the user what is the signification of the serie of gestures he wants to add and of how many gestures the serie of gestures is composed of. Then, the user will have to do each gesture three times to be sure the camera sees it correctly.

In fact, because of the lack of precision of the camera, it only works with really simple gestures (like a fist). If the user does for example a gesture with two fingers, the camera might not see it. And we can’t use a correction function because we don’t know which gesture the user want to do.

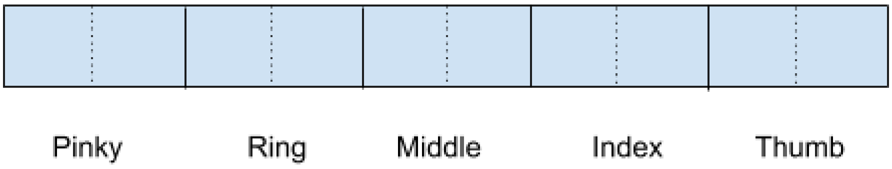


```
Do you want to add words to the Dictionary ? y/n
y
What is the meaning of your word ?
RICM
How many gestures are required for this word ?
3
Learning mode is starting in 5 secondes, be ready !
5
4
3
2
1
-----------------------PLEASE, DO YOUR FIRST GESTURE IN 3 SECONDES-----------------------
STATIQUE
----> Repeat the same gesture in 5 seconds... <----
5
4
3
2
1
----> NOW <----
STATIQUE
----> Repeat the same gesture in 5 seconds... <----
5
4
3
2
1
----> NOW <----
```

C. Gesture encoding

1. Fingers and trajectories

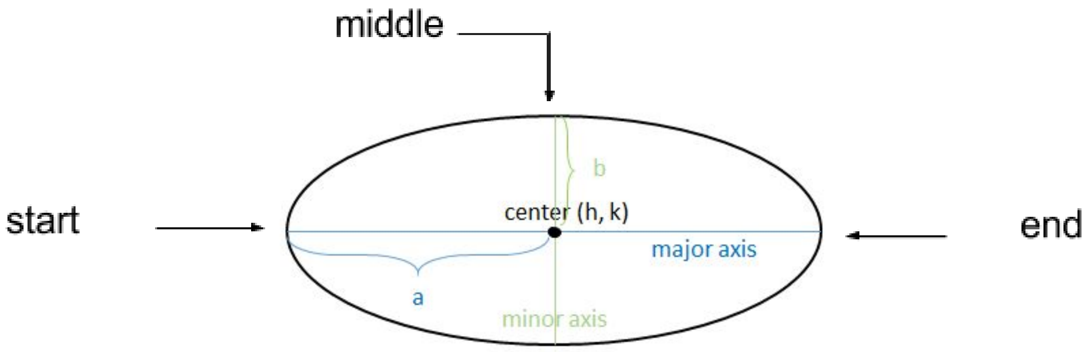
Finger and trajectory data are stored in a total of 18 bits: **2 bits for each finger of one hand + 8 bits for the trajectories**. The information used for fingers is the **foldedness**, which goes from 0 (the finger is completely closed) to 100 (the finger is wide open).



Our finger encoding is as follows: **0b11** means the foldedness is $\geq$75, **0b10** means 50$\leq$foldedness < 75, **0b01** means 25 $\leq$ foldedness < 50 and **0b00** means the foldedness is < 25. Therefore, if all the fingers are open except the pinky which is the only one that is completely closed, the encoding would be: *00 11 11 11 11*.

We then encoded three major types of trajectories: static, straight and elliptic.

- A gesture is considered as **static** if the hand moved of less than NBMETERS_STATIC = 0.01m in at least 92% of the captured frames.
- A gesture is considered as **straight** if most of the captured points respect the equation of a straight line $y = ax + b$. This condition is valid if $|y - ax + b| < ERR\_STRAIGHT$. We then check if it horizontal, vertical or "diagonal"
- A gesture is considered as **elliptic** if most of the captured points respect the equation of an ellipse $\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$. We considered two situations: the ellipse is either full or partial.



In the last case, it is pretty simple to find the values of **a**, **b** and the coordinates of the **center**. The **center** is the middle between the first and the last point. From this, we know the value of **a** as it is the distance between the first point and the center. To find **b**, we have to take the coordinates of the point in the middle of the set of captured points in order to find the distance between this point and the center. We consider that the points follow the equation only if $|\frac{x^2}{a^2} + \frac{y^2}{b^2} - 1| < ERR\_ELIPSE$.

For the trajectories, the encoding is as follows:

partial ellipse	full ellipse	↰	↱	↓	↑	←	→

For example, if the gesture is static, all the bits are 0. If the gesture is ↘, the trajectory encoding would be *00001001* (bottom + right).

The final encoding is the trajectory encoding + the finger encoding.

partial ellipse	full ellipse	↰	↱	↓	↑	←	→	Pinky		Ring		Middle		Index		Thumb	

2. Error correction

One of the main problems we encountered is the imprecision of the camera. It is unable to recognize elaborate gestures, especially when the current position of the fingers has to be recognized. To prevent that, we decided to implement three correction mechanisms:

1: <u>Removing odd frames:</u>

Sometimes when capturing a set of frames the user can be starting or finishing a gesture. As it could lead to errors in the average of the set, when the average of all previous frame is too different from the new frame, a treatment has to be made. For that, we have decided to do two types of treatments. The first one is used when there is not a lot of recorded frames (⅓ of the maximum number of frames), and the second one is when more than ⅔ of the maximum number of thread is reached. In the first case, if the difference is too important, we remove all the previous set and we start over. In the second case, we keep all the previous set, excluding new frames.

2: <u>Removing outlier values:</u>
Each symbol is made of a number of frames recorded by the camera. Sometimes, the camera is unable to recognize a finger during some frames and gets strange positions. So, before trying to recognize a gesture, we try to remove from our dataset the outlier values of the finger's positions. You can check **"removeOutValues**" function in HandTools.cpp for more details.

3: <u>Scoring function:</u>
We are aware that the user might not do the exact gesture that was expected in the Dictionary. In this case, we try to recognize which gesture is the closest to the one the user did and if it is close enough, we will translate his gesture by this one. This is the aim of the "**ScoreGesture**" function in Node.cpp. This function calculates for each finger and for the gesture trajectory, the degree of difference with all the children of the current Node of the Dictionary. If the child with the lowest difference degree has a score lower than 10, it will be accepted as the user's gesture. For that, we decided to define three degrees of error: **Major error (eg), medium error (em), simple error (es)**. And the difference degrees are calculated with the following function:

$$E = 10 * eg + 3 * em + es + nbErrors.$$

III. WebRTC

From the beginning, setting up the **WebRTC** technology in our project was one of our main points. That is why we have decided to split our work into two parallels tasks: the WebRTC server and its User Interface, and the recognition software. This part has to set up a chat application working on WebRTC with a **real time subtitling** system based on the recognition software.

A. Introduction about WebRTC

WebRTC is a technology which saw the day in 2011 and which attempts to allow an evolution of exchanges between multiple clients over the web. This new technology introduces a **browser to browser** communication system without passing through a transitional server, origin of latencies and private life issues.
In order to retrieve necessary information during the establishment of the connection by WebRTC, a **negotiation procedure** has to be done. Every clients has to exchange their connection data on any media. The most efficient way to do it, is to make a negotiation server using the **WebSocket** technology. However there still is an issue due to the fact that a client is often hidden behind a NAT and that is why WebRTC requires to use a negotiation procedure based on the **ICE standard**. This standard allow a "NAT Override" to exchange connection data.

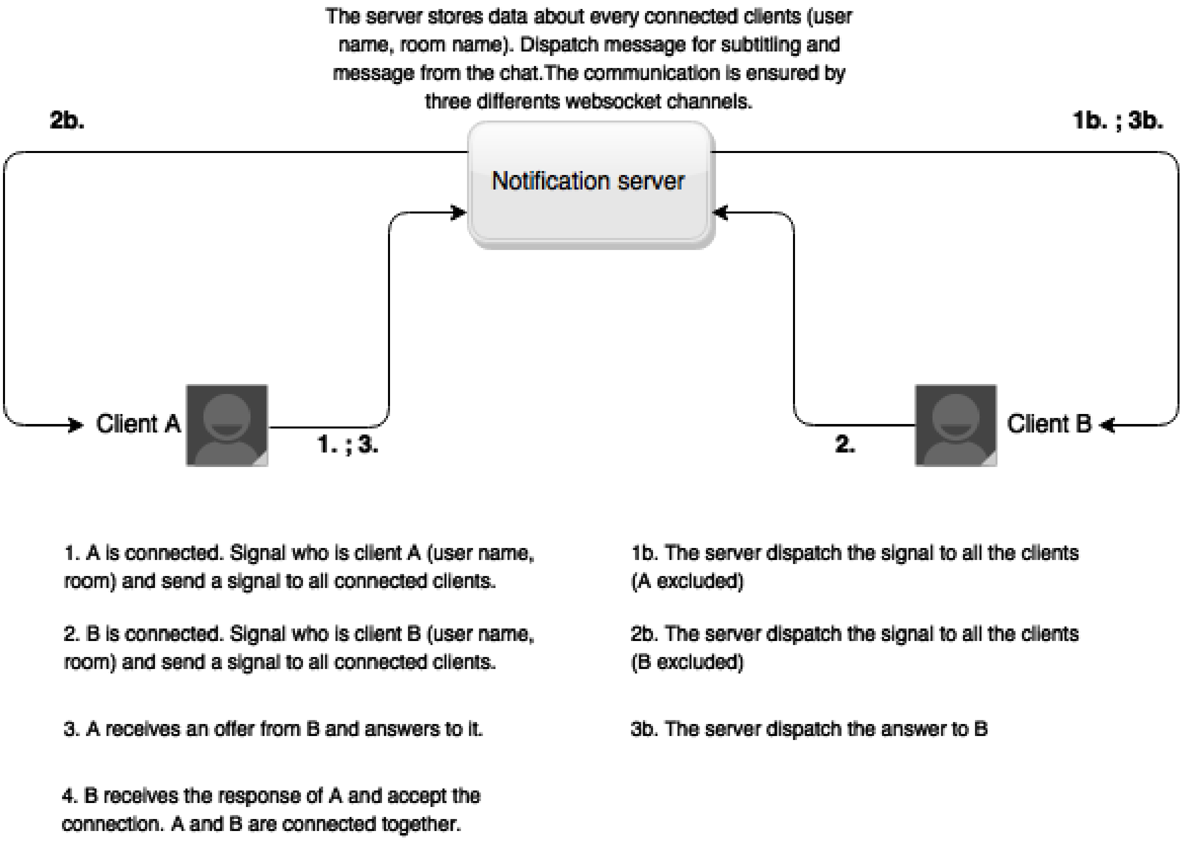
B. Negotiation server

In order to set up our negotiation server, we have decided to implement it using Java and a **OSGi framework** called **Wisdom Framework**. It allows an easy et simplified creation of a web server using an architecture based on the service component runtime **iPOJO**. As stated in the introduction, in order to share connection data, our negotiation server is, at first, a WebSocket server which will forward messages to connected clients. Its second purpose will be to serve each pages of the chat application running on WebRTC.

We have dedicated 3 differents WebSockets:

- one for the WebRTC negotiation
- one for the subtitle transmission
- one for the transmission of incoming messages in the chat

The server in its own doesn’t keep any data about each clients, apart from their **username** and the **room** where they are connected. Hence the server knows all the connected clients and can forward to them the incoming messages. The negotiation procedure rests upon this diagram.



The server stores data about every connected clients (user name, room name). Dispatch message for subtitling and message from the chat.The communication is ensured by three differents websocket channels.

1. A is connected. Signal who is client A (user name, room) and send a signal to all connected clients.

1b. The server dispatch the signal to all the clients (A excluded)

2. B is connected. Signal who is client B (user name, room) and send a signal to all connected clients.

2b. The server dispatch the signal to all the clients (B excluded)

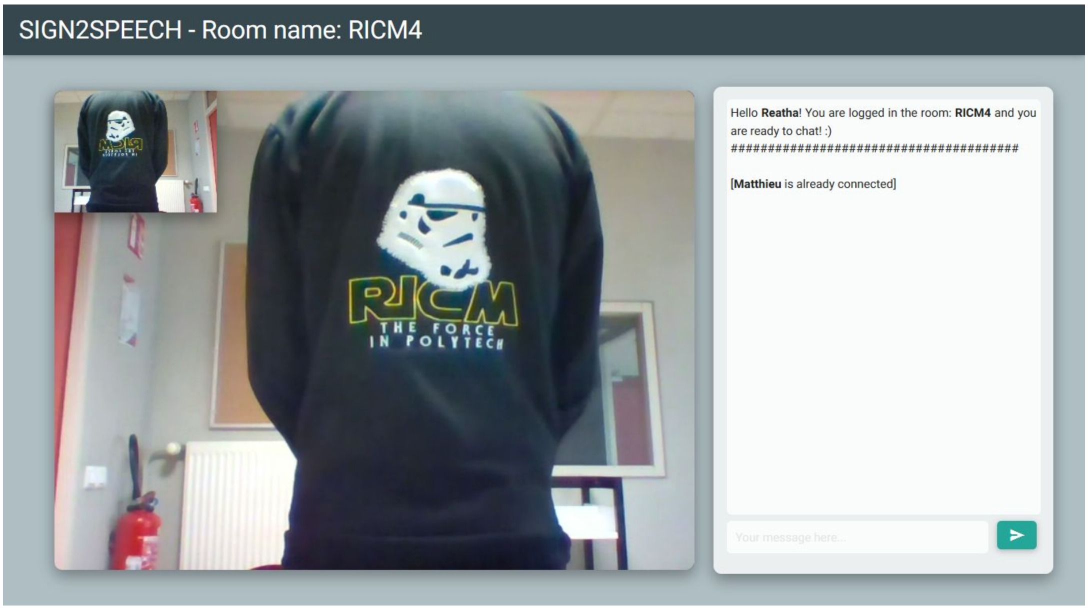
3. A receives an offer from B and answers to it.

3b. The server dispatch the answer to B

4. B receives the response of A and accept the connection. A and B are connected together.

C. The chat application

The web application sets up a system of **room** where only two clients can be connected simultaneously. The Javascript program dialogs with the server so that the two clients can be connected by WebRTC. At the same time, the application opens communication channels with the server for subtitles and messages from the chat.



SIGN2SPEECH - Room name: RICM4

Hello **Reatha**! You are logged in the room: **RICM4** and you are ready to chat! :)
#######################################

[**Matthieu** is already connected]

Chat window

IV. Limits of the project

A. Optimal conditions of use

During our tests, we noticed that the camera often lost in precision and recognition can become very difficult. We have reported different factors that have a negative consequence on the tracking process:

- The user must not wear bracelets or rings
- The user should, as much as possible, use the camera under natural light, rather than artificial light sources
- The user must wear a monochrome top that contrasts with the color of the skin

These elements can reduce the errors and make the tracking better, but it still won't be perfect because of the imprecisions of the camera themselves.

B. Possible improvements

- "Real-time" windows that could show a representation of the hand which the camera is currently analyzing. It could allow the user to know if the camera is able to correctly recognize his hand. It could be done with QT Creator. Our application is not at this time really "friendly-user".
- "2 hands" symbols are currently not implemented in our application
- Improvements of trajectories recognition
- Language Model
- A more accurate camera

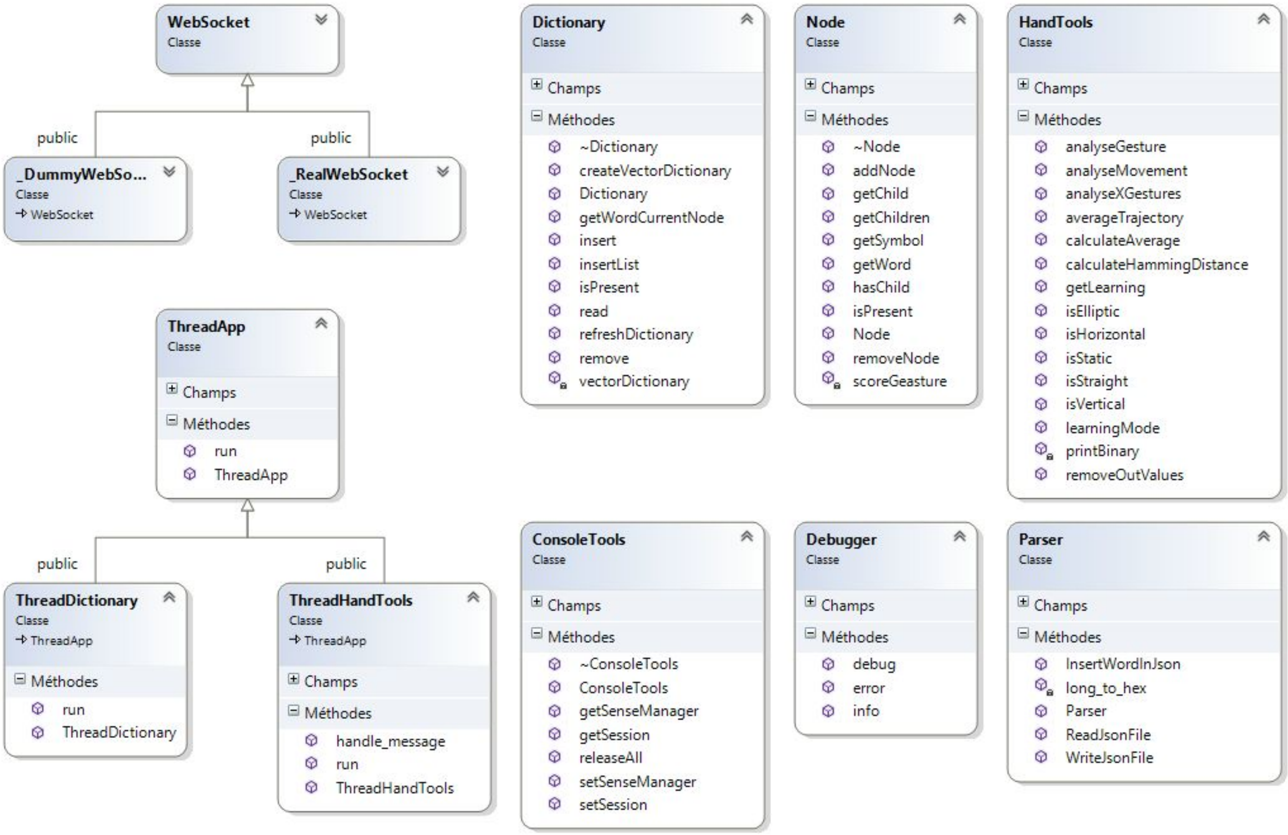
V. Conclusion

The biggest issue we had while working on Sign2Speech was the lack of precision of the camera. That is why the biggest part of the project was to implement scoring and average functions to try to correct the camera's errors. We designed a working "prototype" that, we hope, will be improved in the future, as the technology also improves. We will also need to develop a language model because you can't translate sign language by translating word by word. The translation depends also on previous words, the context etc… It is a real language.

This project taught us how to work as a team, with specific and time constraints. It brought us a lot of team mechanisms as project mechanisms and programming skills. We discovered C++ but mainly, we learnt how to discover a new technology, how to understand it and how to use it (Real Sense SDK and WebRTC). Finally, this project brought us a better sign language knowledge and its problems and we are proud to contribute a bit to solve them.

VI. Appendices

A. Class diagram


WebSocket
Classe
public
public
_DummyWebSo...
Classe
WebSocket
_RealWebSocket
Classe
WebSocket
ThreadApp
Classe
Champs
Méthodes
run
ThreadApp
public
public
ThreadDictionary
Classe
ThreadApp
Méthodes
run
ThreadDictionary
ThreadHandTools
Classe
ThreadApp
Champs
Méthodes
handle_message
run
ThreadHandTools
Dictionary
Classe
Champs
Méthodes
~Dictionary
createVectorDictionary
Dictionary
getWordCurrentNode
insert
insertList
isPresent
read
refreshDictionary
remove
vectorDictionary
Node
Classe
Champs
Méthodes
~Node
addNode
getChild
getChildren
getSymbol
getWord
hasChild
isPresent
Node
removeNode
scoreGeasture
HandTools
Classe
Champs
Méthodes
analyseGesture
analyseMovement
analyseXGestures
averageTrajectory
calculateAverage
calculateHammingDistance
getLearning
isElliptic
isHorizontal
isStatic
isStraight
isVertical
learningMode
printBinary
removeOutValues
ConsoleTools
Classe
Champs
Méthodes
~ConsoleTools
ConsoleTools
getSenseManager
getSession
releaseAll
setSenseManager
setSession
Debugger
Classe
Champs
Méthodes
debug
error
info
Parser
Classe
Champs
Méthodes
InsertWordInJson
long_to_hex
Parser
ReadJsonFile
WriteJsonFile