université
PARIS-SACLAY

FACULTÉ
DES SCIENCES
D'ORSAY

RAPPORT DE PROJET

PROGRAMMATION RÉSEAU AVANCÉ

YILMAZ Mikail

CELAYIR Hilmi

MEFTAH Ouassim

&

BERTRAND Quentin

Janvier 2022 à Avril 2022

Référent de matière : Monsieur MARTIN <steven.martin@lri.fr >
Chargée de TD : Madame TOUATI <h.h.h.touati@gmail.com>

Table des matières

Tables des figures

Introduction

Le but de ce projet est de réaliser une application en réseau sur un *RaspberryPI*. Cette application devra utiliser la technologie du *Wi-Fi AD HOC* avec un adressage via *DHCP* et utiliser une relation entre client et server. *Raspberry* ne supportant que quelques langages de programmation, nous nous sommes tournés vers *Python*. Une fois le langage fixé, il reste néanmoins un problème de taille, qu'elle application créer. L'idée nous est apparue assez rapidement : une messagerie en temps réel. On va donc devoir mettre un *RaspberryPi* en mode *AD HOC*, qui gèrera le côté server et sur lequel les autres *RaspberryPI* se connecteront en tant que clients. Une fois cette architecture réseau établie, nous allons pouvoir passer à la réalisation de notre application.

Analyse Globale

Ce projet sera donc réalisé en *Python*, notre code comportera au moins deux fichiers : un pour la gestion du server et un pour la gestion d'un client, qui pourra lui être exécuté sur différentes machines en simultané pour avoir plusieurs clients. Pour implémenter notre application, nous allons déjà devoir faire une étape de recherche afin de trouver comment réaliser ce que l'on souhaite (méthodes, *packages*, interfaces etc.).

Plan de Développement

Nous allons énoncer toutes les fonctionnalités qui vont être mises en place dans ce projet, la répartition de ces fonctionnalités se fait en deux catégories distinctes, d'un coté les fonctionnalités dites "nécessaires" au bon fonctionnement du projet et de l'autre coté les fonctionnalités dites "complémentaires" ou "optionnelles" qui peuvent être vues comme des *features* améliorant le projet. Il sera également mentionné la difficulté générale liée à l'analyse, la conception, le développement et les tests de chacune de ces fonctionnalités.

Fonctionnalités nécessaires :

- Mettre un *RaspberryPi* en mode *ADHOC*, difficulté moyenne
- Gestion de l'adressage des clients via *DHCP*, difficulté moyenne
- Coté client et un côté server, difficulté moyenne
- Gérer les messages qui arrivent et les renvoyer aux clients, difficulté moyenne
- Afficher les messages

Fonctionnalités complémentaires :

- Affichage graphique pour la messagerie, difficulté moyenne
- Un système avec des administrateurs et mot de passe de connexion, difficulté moyenne
- Permettre aux clients d'envoyer des *emojis*, difficulté moyenne
- Attirer l'attention des autres clients via un son, difficulté facile
- Implémenter un système de sauvegarde des sessions de messages, difficulté difficile
- Un système de commandes pour effectuer différentes tâches et gestion de ces taches, difficulté difficile

Conception Générale

Ce projet a été réalisé en respectant le fait d'avoir un côté server et un côté client, le server pouvant gérer plusieurs clients. L'utilisation du *WI-FI Ad Hoc* permet de créer un réseau local sur lequel nous pourrons lancer notre server et sur lequel nos clients vont se connecter. Notre projet contient 6 fichiers *python* : *BackupMessage*, *Client*, *Imports*, *Packages*, *Run* et *Server.*

Les fichiers *Packages* et *Run* ne servent pas directement à faire fonctionner notre application, ils jouent pour autant un rôle important :

- *Packages* contient la liste des packages utiles pour le projet et passant par une boucle *For Each* et utilisant la fonction *run* du *package Subprocess* afin d'installer tous les *packages* nécessaires pour le bon fonctionnement de notre application.
- *Run* contient aussi la fonction *run* du *package Subprocess* afin de lancer les fichiers *Server* et *BackupMessage* sans avoir à les lancer un par un à chaque fois.

Le fichier *Imports* gère, comme son nom l'indique, toutes les importations de *packages* mais également toutes les constantes et listes communes aux autres fichiers.

Les trois fichiers restants, *BackupMessage*, *Client* et *Server*, gèrent l'application.

- *BackupMessage* comme son nom l'indique, gère tout ce qui est lié à la sauvegarde des messages dans un fichier *.txt.*
- *Client* gère le coté client, c'est-à-dire la partie graphique (connexion, mot de passe si admin et affichage du chat) et l'interprétation des commandes si nécessaire.
- *Server* gère la partie server, la connexion des clients, la reconnaissance de commande dans les messages, l'interactions entre les clients, la gestion de ces derniers ainsi que la gestion de la *backup*.

Et enfin, l'une des parties les plus importante : le *RaspberryPi*. Qui va servir soit de client soit de server. Le *RaspberryPi* client, n'aura juste qu'à se connecter sur le *RaspberryPi* et exécuter le fichier *Client*. En ce qui concerne le *RaspberryPi* server, il est paramétré en mode *Ad Hoc* et gère également l'adressage de chaque client à l'aide du *DHCP*. Nous verrons dans la partie suivante comment nous avons réalisé les paramétrages.

Conception Détaillée

Dans un premier temps, nous allons parler des paramétrages des *RaspberryPi*. Nous avons cherché pendant longtemps avant de trouver une méthode fonctionnelle et surtout, pratique et utilisable sans devoir passer par de nombreuses étapes pour gérer notre *RaspberryPi.* Au début nous avions trouvé une méthode qui consistait à modifier le fichier *Interfaces* mais ça n'a pas fonctionné, et après de nombreuses tentatives qui se sont soldées soit par un échec soit par une méthode trop compliquée pour pouvoir la garder en application, nous nous sommes tournés vers un tutoriel, nous permettant d'installer *RaspAP*, transformant notre *RaspberryPi* en point d'accès *WIFI* et de gérer l'adressage des clients à l'aide du *DHCP.* L'interface qu'offre *RaspAP* est très intuitive et facilite grandement le paramétrage de notre réseau.

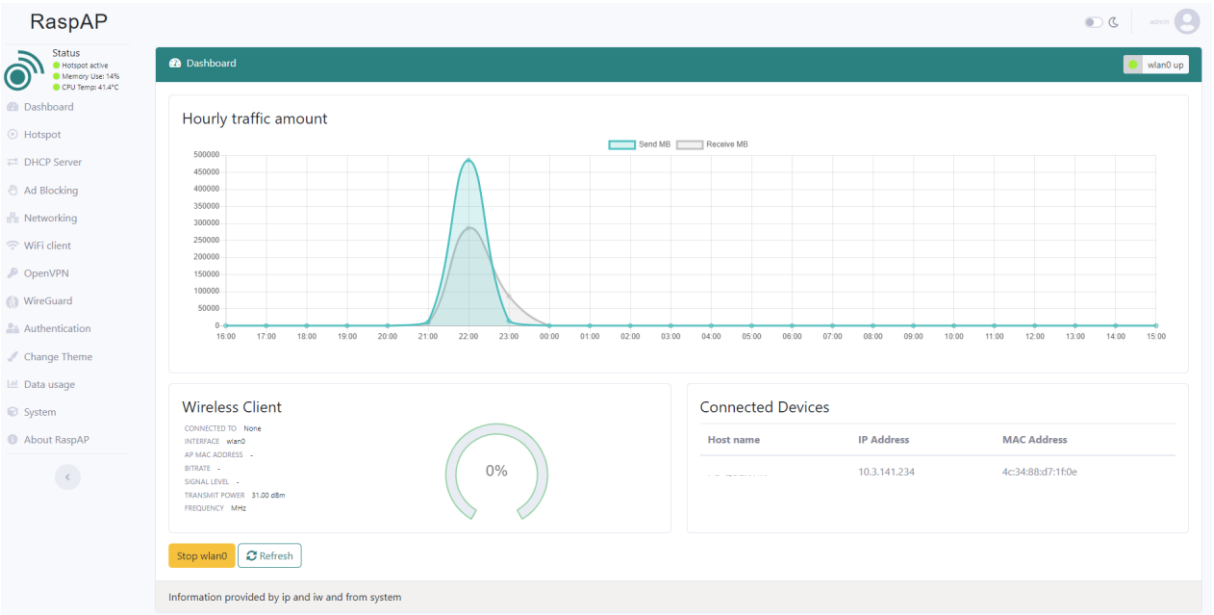

Figure 1: Interface de paramétrage RaspAP

Tout peut être géré depuis cette interface, accessible une fois connecté au réseau du *RaspberryPi* en allant sur internet puis sur 10.3.141.1. Nous avons choisi de mettre l'adresse IP de notre server en statique de façon à toujours se connecter sur la même. Simplifiant très légèrement le code.

Maintenant on va voir comment fonctionne le code des différents fichiers, tout d'abord, le server. Pour gérer les connexions des clients et celle du server, nous avons choisi d'utiliser le *package Socket*, ce dernier nous permet même de gérer le protocole que l'on veut, dans notre cas TCP/IP. Une fois que le PORT et l'adresse de connexion ont été choisis, on en fait un couple afin de réaliser une adresse sur laquelle notre server va se connecter à l'aide de la fonction *bind* de *socket*.

Une fois crée, nous allons faire en sorte que notre server écoute se qu'il se passe afin de voir si un client se connecte ou non, pour cela, dans notre fonction *start()*, nous appelons *server.listen()* qui permet de récupérer le *socket* de connexion et l'adresse du client.

Une fois le client reconnu le server lui envoie un message afin de récupérer son nom, ce message sera encodé avec le format *UTF-8*. Une fois envoyé, le server va attendre une réponse, qu'il stockera dans une variable message

```
# decode the message to save the name
name = connection.recv(1024).decode(FORMAT)
```

Figure 2: Réception d'un message

Si le nom contient le mot "admin" le server enverra au client qu'il doit demander un mot de passe, que le client lui renverra dans la foulée, si ce dernier est bon, on ajoute dans un dictionnaire *clients* le *socket* de connexion et le nom du client. Dès que le client est enregistré dans le dictionnaire, on lance un *thread* qui va lancer la fonction qui va gérer ce client.

```
thread = threading.Thread(target=handle_client, args=(connection, address))
thread.start()
```

Figure 3: Thread de gestion d'un client

La fonction *handle_client* gère et interprète chaque message reçu d'un client et lui renvoie une réponse adéquate. La prise en compte des commandes de *chat* se fait dans cette fonction. C'est également cette fonction qui envoie le message reçu à tout les clients, l'envoyeur compris, car les clients n'affichent dans le chat que ce qui leur est envoyé, cet envoie groupé se fait grâce à la fonction *broadcast()*.

```
# Send the message to all users
def broadcast(message):
    for client in clients:
        client.send(message)
```

Figure 4: Fonction broadcast

Le fichier server contient plusieurs autres fonctions aux utilités simples comme *backupConnection()* qui renvoie le *socket* de connexion de la *backup*, ou encore *userConnection()* qui renvoie le *socket* de connexion d'un client via son *username.*

On va maintenant s'intéresser au fichier client. Il contient une grosse class GUI qui gère entièrement le client, ses messages et son affichage graphique. L'affichage graphique est assez basique, il est réamisé à l'aide du *package Tkinter*. Il s'agit de différentes fenêtres avec des buts différents comme la fenêtre de *login* (Figure 7: Fenêtre Login) ou encore la fenêtre de chat (Figure 8 : Fenêtre de chat). Pour la gestion de messages, c'est le même processus que pour le server, reçoi un message de la part du server et en fonction de son contenu, fait différentes actions. Le cas de base : un message d'un autre client à imprimer dans le chat, il est géré de la façon suivante :

```
# Insert message to text box
self.textCons.config(state=NORMAL)
# emojize turn :heart: into the emoji
self.textCons.insert(END, emoji.emojize(message, language='alias') + "\n\n")
self.textCons.config(state=DISABLED)
self.textCons.see(END)
```

Figure 5: Code d'insertion d'un message dans le chatbox

Le changement à *NORMAL* de la *state*, permet de le rendre modifiable, on insert ensuite notre message, avec la gestion des *emojis* avec le *package emoji*, à la fin de notre chat, on remet la *state* à *DISABLED* et on rend notre message visible dans le *chat*.

La gestion de l'envoie des messages est un peu de la même forme, si le message commence par "!", "@", "a@" ou encore est égal "!quit" il est traité comme exception et est donc envoyé tel quel au server afin de gérer ces exceptions, sinon le message est concaténé avec le nom de l'utilisateur, il ne restera plus qu'à l'afficher. Dans le client, on retrouve aussi la gestion des deux sons possibles dans le chat : le Wizz et l'Avada Kedavra. Les sons sont gérés par le *package pydub*. On *load* le son dans une variable que l'on joue ensuite à l'appel d'une fonction.

```
# Sounds used in commands
wizz_sound = AudioSegment.from_mp3("wizz_sound.mp3")
avada_kedavra = AudioSegment.from_mp3("Avada_Kedavra.mp3")

# Plays the wizz sound
def wizz():
    play(wizz_sound)

# Plays the Avada Kedavra sound
def avadaKedavra():
    play(avada_kedavra)
```

Figure 6 : Importation et lecture d'un son

On va maintenant parler du fichier *BackupMessage,* il y a quatre fonctions qui font fonctionner le système de sauvegarde des messages, la première, la fonction *startbackup()* ouvre le fichier et écrit la date du début de la session. La fonction *endBackup()* met fin à la sauvegarde et écrit dans le fichier une balise de fin de session. La fonction *addMessage()*, comme son nom l'indique elle ajoute un message dans le fichier texte. Et enfin la dernière, *clearbackup()* qui vide le fichier de sauvegarde et commence une nouvelle sauvegarde. Toutes ces fonctions sont appelées dans la fonctions *start()* qui analyse chaque message reçu et agit en conséquence.

Le dernier fichier, *Imports*, contient tous les *imports* nécessaires au projet ainsi que toutes les constantes et les listes contenants les commandes existantes.

Résultats

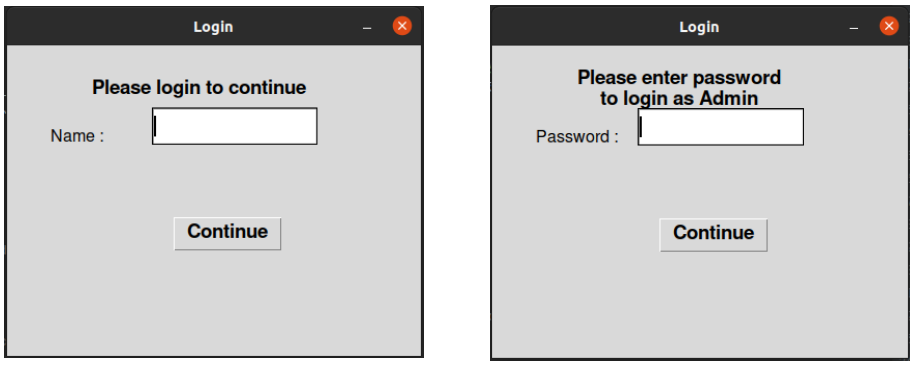

Figure 7: Fenêtre Login

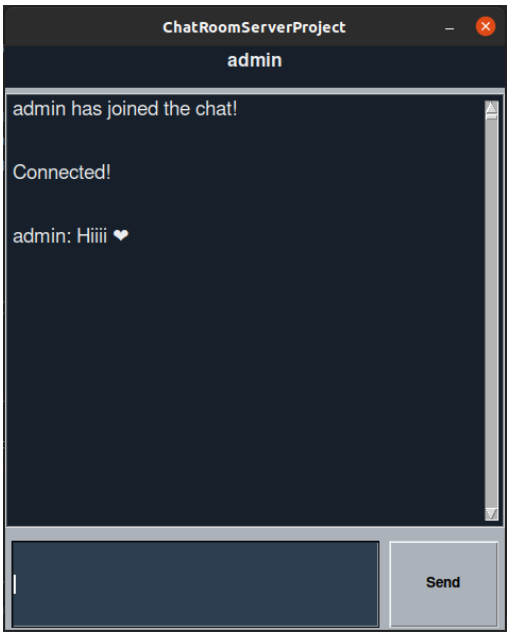

Figure 8 : Fenêtre de chat

Ici, le visuel final de notre projet, on peut y voir la fenêtre de connexion aux côtés de celle demandant le mot de passe si l'utilisateur à demandé à se connecter en admin. La figure du bas représente le visuel de la messagerie avec un aperçu des messages.

Documentation

Le projet a été réalisé en Python 3.8, mais au minimum la version Python 3 est nécessaire pour être sûr de pouvoir avoir accès à toutes les fonctionnalités. Le projet est accompagné d'un *script*, *Packages.py* qui, une fois exécuté, va installer tous les *packages* dont le projet à besoin.

Une fois que tout est en place, sur un *RaspberryPI* mit en mode *Ad Hoc*, si ce n'est pas le cas, vous pouvez suivre le tutoriel cité plus haut. Ensuite sur le premier *RaspberryPi* lancer *Run.py* qui lancera le server ainsi que la *backup*, une fois que le message disant que le server est lancé et que la sauvegarde est activée, connecter les autres *RaspberryPy* au réseau *WIFI* proposé par le 1er *RaspberryPi*. Dès que c'est fait, vous avez juste à lancer le client et le chat va se lancer.

Conclusion et Perspectives

Nous avons réussi à implémenter de nombreuses fonctionnalités, nécessaire et optionnelles, toutefois il reste de nombreuses choses à faire pour faire une application de *chat* bien aboutie. Nous avons rencontré plusieurs difficultés comme la mise en réseau *Ad Hoc* de notre *RaspberryPy* qui allait servir de server ou encore la gestion d'un affichage qui autorise les emojis en couleurs, ce que nous n'avons pas réussi à fixer en vu du délai et du travail qu'il nous restait à faire. Il nous reste toutefois de nombreuses idées comme une commande pour changer la couleur de l'interface, un système de paramètres ou encore avoir une interface plus moderne.

Pour tous les membres du groupe, la gestion de réseau était quelque chose de nouveau, Nous sortons de ce projet avec de nouvelles connaissances qui nous serons utiles à l'avenir.