Projet réseau - Chat Python

Loïc Lachiver Romain Pigret-Cadou

30 avril 2019

Résumé

`https://github.com/picachoc/IRC_python`
Projet UE Réseau L2
Université de Bordeaux
Groupe TMA12_13
Année 2019

Table des matières

1 Introduction

Dans le cadre de l'UE Réseau de la L2 informatique de l'Université de Bordeaux, nous devions, pour nous initier aux sockets, réaliser un chat textuel en Python. En s'inspirant fortement du protocole IRC, les fonctionnalités du chat nous étaient imposées en 2 versions (0 et 1) via le cahier des charges disponible en Annexe.

2 Fonctionnement général

Deux scripts python sont donc nécessaires, le **client** et le **serveur** (respectivement client.py et server.py). Python a été préféré à d'autres langages pour sa simplicité d'implémentation, il est vrai qu'il a été simple de mettre en place certaines structures.

Le Serveur a pour rôle d'orchestrer les différentes demandes des clients, transmises par sockets. Il stocke aussi toutes les informations nécessaires comme les canaux de discussion, les noms d'utilisateurs, qui est situé où, qui est admin, ... Ces informations sont manipulées sous la forme de chaînes de caractères. Pour communiquer avec le serveur il faut lui envoyer des commandes spécifiques, de même il vous transmet ses informations via des commandes aussi (souvent le même intitulé). En conséquence, pour se mettre d'accord sur ces commandes nous devions créer notre propre protocole de communication *(client <-> serveur)*.

Le Client lui peut prendre diverse formes, que ce soit une application graphique ou dans un terminal. L'intérêt du client est d'interpréter les commandes du protocole serveur et de convertir en protocole serveur les ordres donnés par l'utilisateur, que se soit par un clic sur un bouton ou en tapant "/commande". En effet le serveur est conçu pour fonctionner parfaitement avec son protocole, d'ailleurs nous le testions avec la commande linux *nc*. Mais le client permet de rajouter une surcouche ergonomique, en traduisant les codes d'erreurs, en parlant dans la langue utilisateur (français) et de manière plus claire que le protocole.

La conception du protocole serveur était totalement libre, seule la syntaxe et la fonctionnalité des commandes à taper dans le terminal client étaient imposées.

3 Serveur

Notre serveur est basé sur le protocole PICROM, intitulé par concaténation des 3 premières lettres de nos pseudonymes respectifs.

3.1 Documentation

Le serveur peut-être lancé UNIQUEMENT depuis un terminal linux avec la commande : *python3 server.py*
ou *python3 server.py port* , sinon le port par défaut est 1459

Protocole Client -> Serveur :

Légende :
✸ : Nécessite les droits administrateurs
● : Le client ne peut l'utiliser que dans un *channel* (canal de discussion)
■ : Utilisable uniquement en dehors de tout *channel*, dans le *HUB*
▲ : Utilisable de n'importe où

- —**LIST** :▲ le client demande la liste des *channels* ouverts
- —**JOIN <channelName>** :▲ le client veut rejoindre le canal "channelName"
- —**LEAVE** :● le client veut quitter le *channel*
- —**WHO** :● le client veut la liste des *nicks*(pseudonymes) des clients sur le *channel* courant
- —**MSG <message>** :● le client veut envoyer un message sur le *channel*
- —**PRV_MSG N <nick1> ... <nickN> <message>** :● le client veut envoyer un message privé à N utilisateurs identifiés par leurs *nicks*
- —**BYE** :■ le client veut se déconnecter du serveur
- —**KICK <nick>** ✸● l'administrateur veut *kicker* nick du *channel* : lui forcer le LEAVE
- —**REN <newChannelName>** ✸● l'administrateur veut renommer le *channel*
- —**CURRENT <newChannel |>** ▲ le client veut définir un autre *channel* préalablement JOIN comme son *channel* courant | ou simplement demander son *channel* courant

—**NICK <newNick>** ▲ le client veut changer son pseudo

—**GRANT <newAdminNick>** ✷● l'administrateur veut rendre admin un client sur le *channel*

—**REVOKE <oldAdminNick>** ✷● l'administrateur veut destituer un admin sur le *channel*

—**SEND <nick> <file_name>** ● demande d'envoi de fichier au serveur, à destination de nick

—**SENDF <package>** | **void** ● envoi d'une partie du fichier | SENDF seul pour indiquer la fin de la transmission

—**RECV** ● demande de réception d'un fichier qui nous est destiné.

—**RECVF** ● paquet bien reçu, demande du paquet suivant

—**HISTORY** ● demande de transmission de l'historique du channel

Protocole Serveur -> Client :

Légende :
<R> : rang du client, 1 pour administrateur, 0 sinon
● : Commande envoyée au client
■ : Commande envoyée aux autres clients du *channel*
▲ : Commande envoyée à tout les autres clients du serveur
✖ : Commande envoyée aussi au client"
✷ : Commande envoyée au(x) client(s) cible(s)

—**ERR <code>** ● envoie d'un code d'erreur

—**CONNECT <firstNick>** ▲✖ confirme la connexion d'un nouveau client

—**LIST <channel1> ... <channelN>** ● envoie la liste des *channels* ouverts, séparés par des espaces

—**JOIN <channel> <R> <newCommerNick>** ■✖ indique l'entrée d'un nouveau client sur le *channel*, si votre rang est 1 alors vous venez d'ouvrir le *channel*

—**LEAVE <channel> 0 <nick>** | **<channel> 1 <nick> <newAdmin>** ■✖ indique qu'un client a quitté le *channel* | indique un nouvel admin s'il était le dernier.

—**WHO <R> <client1> ... <R> <clientN>** ● envoie la liste des clients sur le channel

—**MSG <channel> <R> <nick> <message>** ■ nick a envoyé un message sur le channel

—**PRV_MSG <channel> <R> <nick> <message>** ✸ nick vous a envoyé un message privé

—**BYE <nick>** ▲ indique qu'un client s'est déconnecté du serveur

—**KICK <channel> <adminNick> <R> <nick>** ■✖ indique que nick va être immédiatement *kick* par l'admin du *channel*

—**REN <channel> <adminNick> <newName>** ■✖ le *channel* a été renommé par l'admin

—**CURRENT <currentChannel>** ● indique au client son *channel* courant ou celui qu'il vient de définir

—**GRANT <channel> <adminNick> <newAdmin>** ■✖ newAdmin a été promu admin par adminNick dans le *channel*

—**REVOKE <channel> <adminNick> <oldAdmin>** ■✖ oldAdmin n'est plus administrateur

—**SEND <0 | 1>** ● Attente du premier paquet | confirmation de la fin du transfert

—**SENDF** ● réception du paquet ok, en attente du prochain

—**RECV <sender_nick | >** ✸ |● Indique au destinataire qu'un fichier l'attend | demande du paquet suivant

—**RECVF <file_package | >** : ● envoie un paquet | téléchargement terminé vous avez tout reçu

—**HISTORY <0 | 1>** : ● début de transmission de l'historique | fin de transmission de l'historique

Protocole Entrée serveur :

A saisir dans le terminal serveur :

—**GLOBAL <message>** ▲ envoie un message à tout les clients sur le serveur

—**KILL <nick>** ▲ pour expulser un client du serveur

—**BAN <nick>** ▲ pour expulser un client et blacklister son adresse IP

Protocole erreurs :

ERR 0 Commande inconnue

ERR 1 Commande non autorisée, requiert les privilèges administrateurs

ERR 2 Vous essayez d'appliquer sur vous même une de ces commandes : KICK, PRV_MSG, GRANT ou REVOKE

ERR 3 Ce pseudo est déjà utilisé

ERR 4 Le client sélectionné n'est pas sur le *channel*

ERR 5 Commande non autorisée ici, rejoignez un *channel* ou quittez les tous.

ERR 6 Vous n'avez pas rejoins ce *channel*, impossible d'en faire le *channel* courant

ERR 7 Vous devez d'abord choisir un pseudo avant toute opération sur le serveur

ERR 8 Ce nom de *channel* est déjà pris

ERR 9 Arguments mal saisis

ERR 10 Vous essayez de JOIN ou CURRENT le *channel* spécial "HUB"

ERR 11 Vous essayez de JOIN un *channel* déjà rejoint, ou de CURRENT votre *channel* courant

ERR 12 Vous voulez GRANT un admin ou REVOKE un non-admin

ERR 13 Aucun fichier ne vous est destiné

ERR 14 Un fichier est déjà en cours d'acheminement pour ce destinataire

ERR 15 Cette IP est bannie et ne peut se connecter

3.2 Implémentation

Fonctionnement général :
Au lancement du serveur, son socket est initialisé, ensuite le serveur répète indéfiniment les opérations suivantes :
La fonction *select*() de la librairie du même nom détecte les sockets qui essaient d'envoyer une information.
Pour chacun de ces sockets :
Si le socket est le socket serveur lui même, nous appelons la fonction pour

une nouvelle connexion client.
Sinon la donnée est récupérée et transformée en chaîne de caractères , ensuite en fonction du type de commande protocole, le programme appelle la fonction de traitement adapté qui effectue l'action.

Structures de données :
Pour garder en mémoire les informations nécessaires à sa gestion, le serveur utilise plusieurs structures de données. La plupart sont des dictionnaires ou des *set*() pour avoir un temps d'accès instantané.
Les *channel* disponibles sont stockés dans un dictionnaire qui pour chaque nom de *channel* associe une liste de sockets clients. L'intérêt de la liste permet de trouver rapidement le prochain admin si le dernier a quitté le canal, il se trouve en tête de liste. On notera l'existence du *set*() "waiting_room" pour stocker les clients connectés mais n'ayant pas encore choisi de pseudo.
Un autre dictionnaire permet de garder les informations clients, à la clé "socket client" est associée une liste de 4 valeurs : [IP, pseudo, 1 si admin 0 sinon, *channel* courant].
L'IP permettra de bannir un client, lors d'une requête client le pseudo peut être rapidement retrouvé tout comme le *channel* d'où elle a été envoyée. L'entier permet aussi de savoir si le client est administrateur.
En plus de cet entier, un dictionnaire associe pour chaque nom de *channel* un *set*() contenant les administrateurs du *channel*, ajouté pour avoir plusieurs administrateurs on peut facilement savoir combien il y a d'admins et qui ils sont.
Pour éviter les doublons de noms de *channels* ou de pseudonymes, deux *set*() enregistrent les noms.

Fonction de traitement : Lorsque le serveur reçoit une commande client "COMMAND" par exemple, le serveur appelle la fonction "picrom_command()" qui va effectuer l'action demandée. Ces fonctions marchent de la même manière :
– Vérification des cas d'erreurs (client cible inconnu, besoin d'être admin,...)
– Modification des structures de données (traitement de l'action)
– Retour serveur au client, à la cible, ou à tous les clients du *channel* ou du serveur
Pour cette dernière étape, 3 fonctions permettent d'envoyer aux clients voulus les retours. "send()", "send_all()" et "send_channel", les 3 en profitent pour laisser une trace dans le *log* serveur, où chaque trame est stockée avec

date et heure.

Choix et optimisations : L'utilisation des dictionnaires et des *set*() permet d'éviter beaucoup de parcours de listes et rend la complexité bien meilleure. Cependant s'il est facile de récupérer le pseudo d'un client en connaissant son socket, l'inverse nécessite de parcourir les clients dans un *channel*. Nous pourrions améliorer ceci avec un autre *set*().
C'est également en ajoutant le système de *channel* courant que nous avons pensé ajouter une structure association à un client la liste des *channels* rejoint. Mais finalement le gain de complexité est difficile à calculer, en effet cette structure améliorerait la complexité de BYE et LEAVE qui ne dépendrait plus que du nombre de *channels* rejoints et pas du nombre total. Mais l'efficacité de REN qui dépendait du nombre de clients sur le *channel* dépendrait du nombre de clients total. Il faudrait savoir qu'elle valeur risque d'être la plus grande et quelle commande serait le plus utilisée en pratique.
Nous envisagerons le passage vers cette nouvelle structure quand nous ajouterons celle qui associe les sockets aux pseudos.

Echange de fichier L'échange de fichiers s'effecture grâce à deux commandes : /SEND, entrée par l'expéditeur, et /RECV, entrée par le destinataire. Nous avons opté pour des entêtes supplémentaires dans le protocole PICROM (voir SENDF et RECVF plus haut).
Pour SEND, le protocole est décrit en annexe. RECV fonctionne sur le même principe, sauf qu'une fois que le serveur a confirmé qu'il avait un fichier pour le client, le client répond "RECVF" qui correspond à l'acquittement. Cela permet de garder un système question/réponse.
Les difficultés rencontrées dans l'échange de fichiers sont détaillées dans la partie "problèmes rencontrés".

4 Client

4.1 Documentation

Le serveur peut-être lancé UNIQUEMENT depuis un terminal linux avec la commande : *python3 client.py*
ou *python3 client.py IP port* , sinon l'IP par défaut est localhost et le port 1459

✸ : Nécessite les droits administrateurs
Commandes disponibles :

—/**HELP** affiche la liste des commandes disponibles

—/**LIST** affiche les *channels* ouverts

—/**JOIN** <**channel**> pour rejoindre (ou créer) un *channel*

—/**LEAVE** pour quitter le canal courant

—/**WHO** affiche les utilisateurs sur le canal

—<**message**> envoi un message sur le *channel*

—/**MSG** <**nick1 ;nick2 ;...**> <**message**> pour envoyer un message privé à un ou plusieurs clients

—/**BYE** pour se déconnecter du serveur

—/**KICK** <**pseudo**> ✸ éjecte un utilisateur du canal

—/**REN** <**channel**> ✸ change le nom du canal

—/**CURRENT** affiche le nom du *channel* courant

—/**CURRENT** <**channel**> défini le *channel* courant

—/**NICK** <**nouveauPseudo**> pour changer de pseudo

—/**GRANT** <**pseudo**> ✸ pour rendre un utilisateur admin

—/**REVOKE** <**pseudo**> ✸ pour destituer un admin

—/**SEND** <**pseudo**> <**nomDuFichier**> envoyer un fichier à destination d'un client

—/**RECV** <**nomDuFichier**> récupérer un fichier qui vous est destiné

—/**HISTORY** récupérer l'historique du channel

4.2 Implémentation

Au démarrage du client, on initialise tout d'abord les deux variables HOST et PORT. Si le programme est appelé sans argument, on les initialise respectivement à 127.0.0.1 (soit localhost) et 1459.

Voilà les variables et structures utilisées :

nick : le nick du client (str).

file_send file object du fichier à envoyer (avec /SEND).

file_recv file object du fichier à recevoir (avec /RECV).

cmd_list dictionnaire ayant pour clé la commande et comme valeur son nombre d'arguments. Cela permet de vérifier si la commande existe et si elle est bien tapée.

help_msg le message d'aide affiché avec /HELP.

err_msg dictionnaire ayant pour clé le numéro de l'erreur et comme valeur le message affiché côté client.

Tout d'abord, le client rentre dans une boucle qui teste si le nick entré est bien valide, que ce soit côté client (si le nick n'est pas vide) et côté serveur (s'il n'est pas déjà utilisé).
On rentre ensuite dans la boucle principale.
Afin de ne pas bloquer les retours serveur, on utilise select.select pour traiter les données du "premier qui parle" entre le socket de connexion et sys.stdin. Par contre, avec cette méthode, les retours serveurs sont synchronisés avec ce qu'il tape. Pour éviter cela, il faudrait désactiver les entrées canoniques (avec tty.setraw(sys.stdin)), mais cette solution n'est pas viable sous Windows, donc nous ne l'avons pas adoptée.
Ensuite, en fonction de l'entrée sélectionnée on exécute la fonction send (si c'est une entrée utilisateur) ou display (si c'est un retour serveur).
Pour chaque commande entrée, on teste le nombre d'arguments et on construit la trame PICROM correspondante avec la fonction send. Quant à la fonction display, elle récupère les arguments des trames PICROM et affiche les messages correspondants, avec :
- c'est à dire le channel (si nécessaire)
- les pseudos (avec affichage des admins grâce à la fonction display_rank)
- les eventuelles erreurs (avec le dictionnaire err_msg)
- les informations serveurs (quand un utilisateur join, leave, change de pseudo, kick, OP etc.)

Exemple d'affichage côté client :

#chanL2Info | Romain a rejoint le channel.
#chanL2Info | Loic a rejoint le channel.
#chanL2Info | @GBlin@ > Qu'est-ce que la feuille d'un arbre ?
#chanL2Info | Loic > En morphologie végétale, c'est l'organe spécialisé dans la photosynthèse chez les végétaux supérieurs.
#chanL2Info | Romain > C'est un noeud qui n'a pas de fils.
#chanL2Info | @GBlin@ a kické Loic !

#chanL2Info | @GBlin@ a OPé Romain.
#chanL2Info | @Romain@ > yes !

5 Problèmes rencontrés

Structures de données du serveur : Nous voulions garantir un temps d'accès constant aux données côté serveur. Ainsi, nous avons souvent changé de stratégie pour stocker les données du client pour finalement utiliser des dictionnaires dans la plupart du temps.

Envoi de fichiers encodés (SEND et RECV) :
Il est facile d'écrire et d'envoyer des données de fichiers en clair (fichiers textes, par exemple). On peut les ouvrir avec la fonction open('filename','r') Par contre, les pdf, images ou autres fichiers non clairs sont ouverts avec l'argument 'rb' au lieu de 'r'. Cela change notre manière de décoder les trames côté serveur, car cela veut dire que SENDF envoie des données non claires. Les fonctions decode(), split() etc. ne peuvent pas s'appliquer aux trames dont l'entête est SENDF car ce ne sont pas de simples chaînes de caractères. Pour détecter une entête SENDF, on doit donc la comparer avec b"SENDF", qui correspond à la valeur encodée de "SENDF". Si ce n'est pas cette entête, alors on peut se permettre de décoder la trame.
Emplacement des fichiers à envoyer (SEND) : En argument de la fonction SEND, il est demandé dans le cahier des charges de spécifier <file/to/path>. Sur Windows, les chemins d'accès à un fichier sont de cette forme : C :
Users
picrom
test.py, alors que sur Linux les "
" sont remplacés par des "/". Cela complique donc la gestion des répertoires, nous avons finalement opté pour mettre le fichier à envoyer dans le même dossier que "client.py", ce qui est plus simple. De même côté serveur, le fichier sera téléchargé dans le même dossier que server.py.
Synchronisation serveur / client (SEND et RECV) : Au départ, on lisait le fichier par paquet de 1024 bits et on envoyait ces paquets avec l'entête "SENDF ". La fonction listen du socket server avait pour argument 1500. Le serveur attendait donc des trames de 1500 bits. Par conséquent, le serveur concaténait plusieurs trames en une seule et donc essayait parfois de

décoder des trames correspondantes aux données brutes des fichiers envoyés, ce qui le faisait planter. Nous avons donc synchronisé le serveur avec le client à 1024 bits de longueur de trame maximale afin d'être sûr de recevoir les bonnes trames.

6 Conclusion

"J'ai trouvé le sujet très motivant. Je pense qu'en informatique il est important de pratiquer pour progresser, et ce projet en est la preuve. J'ai progressé en python (alors que je connaissais peu ce langage), j'ai découvert ses avantages et ses inconvénients. Le fait qu'il soit interprété permet une plus grande flexibilité, en revanche il est difficile de savoir si chaque fonction marche, donc tester son programme peut vite devenir fatiguant. Nous avons été surpris par la complexité de l'envoi et réception de fichiers, mais cela nous a permis de découvrir d'autres pans de l'informatique et de s'adapter. Il était gratifiant de voir notre programme fonctionner, car le côté intéractif du chat nous a donné la sensation d'avoir travaillé pour quelque chose (exemple : échange de fichiers à distance depuis nos deux PC) Le fait de partir de zéro et de devoir créer nos propres protocoles m'a montré l'importance d'un cahier des charges (ici le Readme) bien rédigé afin de pouvoir construire notre programme sur ses bases. Au final, j'ai apprécié mener ce projet avec Romain, je regrette seulement que le rendu tombe au moment des partiels car cela nécessite tout de même beaucoup de temps. "

Loïc Lachiver

"Mon avis personnel sur le projet réseau est qu'il était intéressant à coder mais très limité à la manipulation des sockets et des chaînes de caractères. Le projet est extrêmement chronophage, m'efforçant de commit régulièrement et après de nombreux tests j'ai passé plus de temps à taper des commandes serveur qu'autre chose, le côté interprété ne facilite pas les choses. Finalement nous avons réussi à nous transmettre des fichiers depuis des PC distants et le serveur est très fonctionnel. Par ailleurs les tests automatiques et le cahier des charges parfois trop précis et pas très intuitif étaient très contraignants, pour s'y conformer nous avons parfois sacrifié l'efficacité et la propreté du code."

Romain Pigret-Cadou

7 Annexes

Cahier des charges :

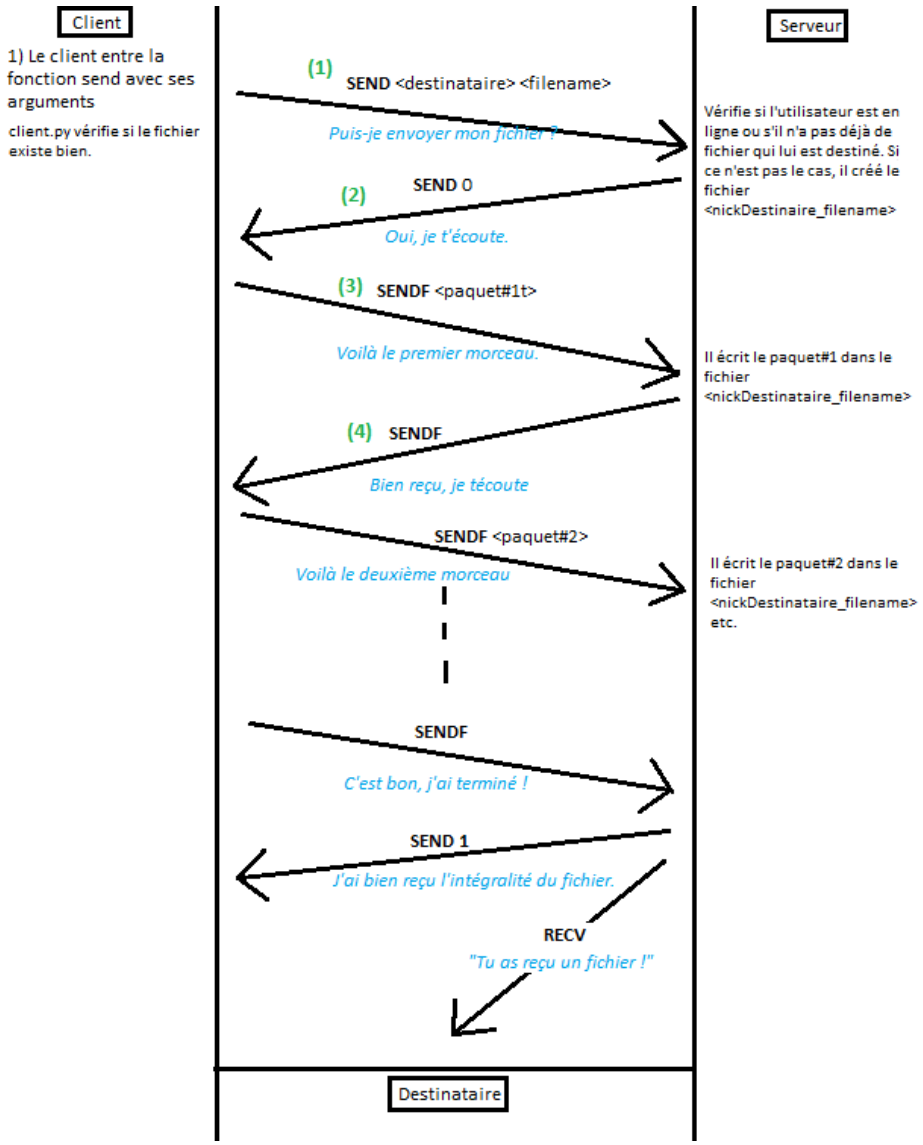
Commandes client v0 :

— /HELP : print this message
— /LIST : list all available channels on server
— /JOIN <channel> : join (or create) a channel
— /LEAVE : leave current channel
— /WHO : list users in current channel
— <message> : send a message in current channel
— /MSG <nick> <message> : send a private message in current channel
— /BYE : disconnect from server
— /KICK <nick> : kick user from current channel [admin]
— /REN <channel> : change the current channel name [admin]

Commandes client v1 :

— /CURRENT : print current channel name
— /CURRENT <channel> : set current channel
— /MSG <nick1 ;nick2 ;...> <message> : send a private message to several users in current channel
— /NICK <nick> : change user nickname on server
— /GRANT <nick> : grant admin privileges to a user [admin]
— /REVOKE <nick> : revoke admin privileges [admin]
— /SEND <nick> </path/to/file> : send a file to a remote user
— /RECV </path/to/file> : receive a file and save it locally
— /HISTORY : print history of current channel (saved by server)

Protocole SEND :


Client
1) Le client entre la fonction send avec ses arguments
client.py vérifie si le fichier existe bien.
Serveur
(1) SEND <destinataire> <filename>
Puis-je envoyer mon fichier ?
Vérifie si l'utilisateur est en ligne ou s'il n'a pas déjà de fichier qui lui est destiné. Si ce n'est pas le cas, il créé le fichier <nickDestinaire_filename>
(2) SEND 0
Oui, je t'écoute.
(3) SENDF <paquet#1t>
Voilà le premier morceau.
Il écrit le paquet#1 dans le fichier <nickDestinataire_filename>
(4) SENDF
Bien reçu, je técoute
SENDF <paquet#2>
Voilà le deuxième morceau
Il écrit le paquet#2 dans le fichier <nickDestinataire_filename> etc.
SENDF
C'est bon, j'ai terminé !
SEND 1
J'ai bien reçu l'intégralité du fichier.
RECV
"Tu as reçu un fichier !"
Destinataire