Yann Simon

11 avril 2023



DÉPARTEMENT DE MATHÉMATIQUES, D'INFORMATIQUE ET DE GÉNIE

**INF26207 – Téléinformatique**

**Hiver 2023**

**Devoir #2**

Table des matières

[I. Introduction](#_heading=h.gjdgxs) **3**

[II. Matériel et méthodes](#_heading=h.30j0zll) **3**

[III. Résultats](#_heading=h.1fob9te) **4**

[IV. Analyse des résultats](#_heading=h.3znysh7) **6**

[V. Conclusion](#_heading=h.2et92p0) **6**

[**Biographie**](#_heading=h.yytsy4js2d1l) **6**

1. **Introduction**

Tout d’abord, qu’est ce qu’un socket ? Sur Linux, c’est un ‘file descriptor’ associé à un fichier, la libc disposent de quelques appels systèmes qui permettent d’interagir avec les sockets, comme la fonction ‘bind’ ou ‘ accept’. Les langages de programmations comme le rust ou le python ont construit leur socket en utilisant la libc.

1. **Objectifs**

Ce TP à pour objectif de créer un client et un serveur FTP basique en utilisant des sockets. Cela nous permet de comprendre comment les sockets fonctionnent et comment le protocole TCP et UDP sont construits. Nous devons donc créer nos propres en-têtes TCP et UDP ainsi qu’un protocole de communication pour le FTP.

1. **Résultats**

J’ai choisi d’utiliser le langage Rust et d’avoir un seul binaire pour le client et le serveur, la commande ‘ftp -c’ permet de lancer le client et ‘ftp -s’ le serveur, le serveur devant bien sûr être lancée avant le client.

Le serveur utilise un thread par client, il peut donc y avoir plusieurs clients connectés en simultané. Lorsque le client est lancé, il va automatiquement se connecter au serveur sur le port 22222.

Le client dispose de trois commandes:

* “put <path relatif du fichier>” -> ex: “put ./mon\_fichier.txt”
* “get <nom du fichier à download> <path relatif du fichier local>” -> ex: “get mon\_fichier.txt ../download.txt”
* “exit”

1. **Conception**

J’ai créé des classes Udp et Tcp afin de simplifier l’utilisation des sockets, j’ai également créé une fonction fn “receive\_file(file: &mut File, udp: &mut Udp) -> std::io::Result<()>” qui est utilisé par la fonction “put” côté serveur et “get” côté client, ainsi qu’une fonction “fn send\_file(file: &mut File, udp: &mut Udp) -> std::io::Result<()>” est utilisé par la fonction put côté client et get côté serveur.

J’utilise différentes struct pour la communication TCP:

* CommandPacket: permet au client d’envoyer la commande qu’il souhaite exécuter.
* FileInfoPacket: permet de transmettre les infos du fichiers comme le nom et la taille.
* ResponsePacket: permet d’envoyer une réponse avec un statut (Ok, Err) et un éventuel message d’erreur.

Ainsi que pour l’UDP:

* FilePacket: permet d’envoyer le numéro de la séquence envoyé, la taille des données envoyés et les données.
* ResponseFilePacket: permet d’envoyer un accusé de réception avec le numéro de séquence.

Pour faciliter la sérialisation et la désérialisation, j’utilise les crate serde et bincode qui me permettent de transformer un tableau de bytes en struct et inversement.

Le taille des blocs envoyé est définie avec la constante FILE\_BLOCK\_SIZE à 1024, lors de l’envoie d’un fichier, la fonction “send\_file”, lit le fichier avec un offset égale à FILE\_BLOCK\_SIZE, le place dans le FilePacket et l’envoie. Le serveur read le FilePacket et le write directement. La transmission de fichier fonctionne de la même manière pour le “get ” et le “put”, seule l’étape de récupération du nom du fichier va changer.

1. **Conclusion**

Ce TP m’a permis de mieux comprendre comment les sockets, les protocoles TCP et UDP et la logique de l’envoie d’un fichier