TD
Socket Avancé
Réalisation d'une Application de Chat basique

SANNA Thomas

October 3, 2024

1 Introduction

Ce rapport détaille la réalisation d'une application de chat basique permettant à plusieurs utilisateurs de se connecter simultanément. Le serveur utilise un modèle non-bloquant pour gérer les connexions et les messages. Le projet est divisé en trois parties principales:

1. Encodage et décodage des messages.
2. Implémentation du serveur.
3. Implémentation du client.

2 Exercice 1 — Réalisation du Chat

2.1 Objectifs

1. **Structuration des messages**: Chaque message transmis contient un en-tête (un entier de 32 bits en big-endian) indiquant sa taille. Le récepteur doit d'abord récupérer cette valeur avant de lire le message complet.
2. **Fonctions d'encodage et de décodage**:
   - `encode_message(mess: str)`: Encode le message en bytes, ajoute un timestamp et un en-tête contenant la taille du message.
   - `decode_message(encoded_mess: bytes)`: Décode les bytes en extrayant l'en-tête et le message.

2.2 Implémentation

2.2.1 Fichier `utils.py`

Ce fichier contient les fonctions d'encodage et de décodage des messages.

Fonction `encode_message` :

```python
import time
import struct

def encode_message(message: str) -> bytes:
    timestamp = time.strftime('%Y-%m-%d %H:%M',
        ↪ time.gmtime())
    full_message = f"{timestamp} {message}"
    message_bytes = full_message.encode('utf-8')
    header = struct.pack('>I', len(message_bytes))  # >I
        ↪ signifie que le format est big-endian unsigned int
    return header + message_bytes
```

Listing 1: encode_message dans utils.py

Explications

- `encode_message`: Ajoute un timestamp au message, encode le message en bytes et ajoute un en-tête contenant la taille du message.

Fonction `decode_message` :

```python
def decode_message(data: bytes) -> str:
    if len(data) < 4:
        raise ValueError("Data is too short to contain a
            ↪ valid message header")

    message_length = struct.unpack('>I', data[:4])[0]
    if len(data) < 4 + message_length:
        raise ValueError("Data is too short to contain the
            ↪ full message")

    full_message = data[4:4 +
        ↪ message_length].decode('utf-8')
    yearMonthDay, hourMinSec, message =
        ↪ full_message.split(' ', 2)
    return message
```

Listing 2: decode_message dans utils.py

Explications

- `decode_message`: Décode les bytes en extrayant l'en-tête et le message. Vérifie que les données sont suffisamment longues pour contenir un message valide.

3 Exercice 2 — Implémentation du Serveur

3.1 Objectifs

1. Sauvegarder les sockets acceptés dans une liste.
2. Afficher le message reçu dans la console avec l'adresse du socket source.
3. Transmettre le message à tous les autres clients connectés, en incluant l'heure et l'adresse de l'émetteur dans l'en-tête du message.

3.2 Implémentation

3.2.1 Fichier `serv.py`

Ce fichier contient l'implémentation du serveur.

Fonction `accept` :

```python
import selectors
import socket
import utils

# Create a default selector
sel = selectors.DefaultSelector()

# List to store accepted sockets
clients = []

def accept(sock, mask):
    conn, addr = sock.accept()  # Should be ready
    print("Accepted connection from", addr)
    conn.setblocking(False)
    sel.register(conn, selectors.EVENT_READ, read)
    clients.append((conn, addr))  # Add the new connection
        ↪ to the clients list
```

Listing 3: accept dans serv.py

Explications

- `accept`: Accepte une nouvelle connexion, l'ajoute à la liste `clients` et l'enregistre pour surveiller les événements de lecture.

Fonction `read` :

```python
def read(conn, mask):
    data = conn.recv(1000)  # Should be ready
    addr = conn.getpeername()
    if data:
        message = utils.decode_message(data)
        print(f"Received message from {addr}: {message}")

        # Create the message to be sent to other clients
        encoded_message = utils.encode_message(message)

        # Send the message to all other clients
        for client, client_addr in clients:
            if client != conn:
                client.send(encoded_message)
    else:
        print("Closing connection to", conn)
        sel.unregister(conn)
        conn.close()
        clients.remove((conn, addr))  # Remove the
            ↪ connection from the clients list
```

Listing 4: read dans serv.py

Explications

- `read`: Lit les données du socket, affiche le message reçu avec l'adresse du socket source, et transmet le message à tous les autres clients connectés.

Si `data` n'existe pas (c'est-à-dire que `recv` retourne une chaîne vide), cela signifie que le client a fermé la connexion. Dans ce cas, le serveur ferme la connexion avec le client, désenregistre le socket du sélecteur et le retire de la liste des clients.

Configuration du Serveur et Boucle d'Événements :

```
# Create and bind the socket
sock = socket.socket()
sock.bind(("localhost", 12345))  # Change port to 12345 to
    ↪ match the client
sock.listen(100)
sock.setblocking(False)
sel.register(sock, selectors.EVENT_READ, accept)

print("Server started on port 12345")

# Event loop
while True:
    events = sel.select()
    for key, mask in events:
        callback = key.data
        callback(key.fileobj, mask)
```

Listing 5: Configuration du serveur et boucle d'événements dans serv.py

Explications

- **Configuration du Serveur**: Crée et lie le socket, le configure pour le mode non-bloquant, et l'enregistre pour surveiller les événements de lecture.
- **Boucle d'Événements**: Sélectionne les événements prêts et appelle la fonction de rappel appropriée pour chaque événement.

4 Exercice 3 — Implémentation du Client

4.1 Objectifs

1. Créer une application client permettant de se connecter au serveur.
2. Utiliser des threads pour gérer l'envoi et la réception des messages en parallèle. Cette fonctionnalité est utile car l'envoi et la réception de messages sont normalement des opérations bloquantes; Il est donc nécessaire d'utiliser des threads pour les exécuter simultanément.

4.2 Implémentation

4.2.1 Fichier `client.py`

Ce fichier contient l'implémentation du client.

Fonction `send_loop` :

```python
import threading
import socket
import utils

# Define and connect the socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect(('localhost', 12345))

def send_loop():
    while True:
        message = input()
        sock.send(utils.encode_message(message))
```

Listing 6: send_loop dans client.py

Explications

- `send_loop`: Lit les messages de l'utilisateur et les envoie au serveur.

Fonction `receive_loop` :

```python
def receive_loop():
    while True:
        message = sock.recv(1000)
        print("Donnée envoyée par le serveur: ",
            ↪ utils.decode_message(message))
```

Listing 7: receive_loop dans client.py

Explications

- `receive_loop`: Reçoit les messages du serveur et les affiche.

Utilisation des Threads :

```
thread_send = threading.Thread(target=send_loop)
thread_send.start()

thread_receive = threading.Thread(target=receive_loop)
thread_receive.start()

thread_send.join()
thread_receive.join()
```

Listing 8: Utilisation des threads dans client.py

Explications

- **Threads**: Utilise des threads pour exécuter `send_loop` et `receive_loop` simultanément.