Sujet du Mini-Projet (TD) du cours Programmation Système et Réseaux sous UNIX

Présenté Par : Mohamed Yassine Ahmed Ali
& Mohamed Dhia Hamam

1- Contexte du projet :

Ce projet s'inscrit dans le cadre du cours de Programmation Système et Réseaux sous UNIX, qui met l'accent sur la compréhension et la mise en œuvre des concepts fondamentaux des sockets et des communications réseau en environnement Unix/Linux. Le projet vise à développer deux applications client/serveur en utilisant les protocoles UDP (mode non connecté) et TCP (mode connecté) qui doit être réalisée en quatre étapes :

- Monoclient/monoserveur
- Multiclient/mmnoserveur
- Multiclent/multiserveur

2- Mode non-connecté (UDP) :

- **ClientUDP.c :**
  Les étapes principales du code `clientudp.c` sont :
  1. Vérifier les arguments (hôte et port).
  2. Créer un socket UDP.
  3. Configurer l'adresse du serveur.
  4. Générer un nombre aléatoire et préparer un message.
  5. Envoyer et recevoir le message au serveur.
  6. Afficher les valeurs reçues.
  7. Fermer le socket.

- **ServeurUDP.c :**
  Les étapes principales du code `ServeurUDP.c` sont :
  1. Vérifier les arguments (port uniquement).
  2. Créer un socket UDP.
  3. Configurer et lier l'adresse du serveur.
  4. Recevoir un message du client.
  5. Générer une séquence aléatoire en fonction de la requête.
  6. Envoyer la réponse au client.
  7. Fermer le socket.

> logger.c : Fournit des fonctionnalités pour enregistrer, afficher et gérer des logs dans un programme.
>
> make.sh : Script shell qui sert à compiler les fichiers sources en générant les exécutables correspondants.

3- Mode connecté (TCP) :

3.1- Monoclient/monoserveur :

- **ClientTCP.c :**
  Les étapes principales du code `clientTcp.c` sont :
  1. Connexion au serveur.
  2. Authentification.
  3. Envoi de commandes via une boucle.
  4. Réception des résultats et affichage.
  5. Fin de session avec l'opération 5.
  6. Fermer le socket.

- **ServeurTCP.c :**
  Les étapes principales du code `ServeurTCP.c` sont :
  1. Initialisation du serveur.
  2. Acceptation des connexions client.
  3. Authentification sécurisée.
  4. Envoi du menu d'options.
  5. Traitement des requêtes client (services variés).
  6. Fermeture des connexions.

3.2- Multiclient/monoserveur :

- **ServeurTCP.c :**
  - Le serveur utilise des threads pour gérer plusieurs clients simultanément.
  - Chaque client est traité dans un thread distinct, permettant un traitement parallèle.
  - Le serveur accepte plusieurs connexions avec safe_accept dans une boucle infinie.
  - Pour chaque connexion, un thread est créé avec pthread_create pour gérer le client.
  - Les threads doivent partager certaines ressources (comme le compteur active_clients).
  - Synchronisation nécessaire pour éviter les conflits :
    Utilisation de mutex avec `pthread_mutex_lock` et pthread_mutex_unlock.
    Exemple : Mise à jour du compteur active_clients pour suivre les connexions actives.

3.2- Multiclient/multiserveur :

- **Proxy.c:**

1. Centralisation des connexions :
   Le proxy reçoit toutes les requêtes des clients et décide à quel serveur spécifique (authentification, date, liste de fichiers, etc.) rediriger chaque requête. Il agit comme un intermédiaire entre les clients et les serveurs de services.
2. Routage des requêtes :
   Basé sur l'opération demandée (message.op), le proxy redirige les requêtes vers le serveur approprié. Cela permet de découpler les clients des serveurs de services, rendant l'architecture plus modulaire et maintenable.
3. Gestion des connexions multi-clients :

Utilise des threads pour gérer plusieurs clients en parallèle, chaque client ayant son propre thread. Cela permet une gestion simultanée de plusieurs connexions.

4. Abstraction de la logique métier côté client :
   Les clients n'ont pas besoin de savoir quel serveur gère une opération spécifique. Ils communiquent simplement avec le proxy, qui se charge de rediriger la requête vers le bon serveur.

> Cette classe sert donc de cœur de l'architecture client-proxy-serveurs et illustre bien les principes de modularité, de découplage et de gestion réseau centralisée.

- **Exemple dimplementation d'un serveur (serveur_auth.c) :**

1. Vérification des identifiants :
   Le serveur reçoit les identifiants (nom d'utilisateur et mot de passe) du client.Il utilise une fonction appelée authentification pour valider ces informations.
2. Filtrage des clients non autorisés :
   Si l'authentification échoue, le client est invité à réessayer.Le client ne peut pas accéder aux autres services sans une authentification réussie.
3. Point d'entrée pour les services :
   Une fois l'utilisateur authentifié, le serveur fournit un menu d'options indiquant les autres services disponibles (date, liste de fichiers, contenu d’un fichier, etc.).
4. Gestion multi-clients :
   Le serveur utilise des threads pour traiter plusieurs clients simultanément.

3.3- Compilation :

Le fichier make.sh permet de compiler facilement les différents composants du système multiclients/multiserveurs. Il prend un argument pour décider si l'utilisateur veut compiler les serveurs ou le client, simplifiant ainsi le processus de génération des exécutables.

- **Exemple de fichier make.sh pour Multiclients/multiserveurs :**

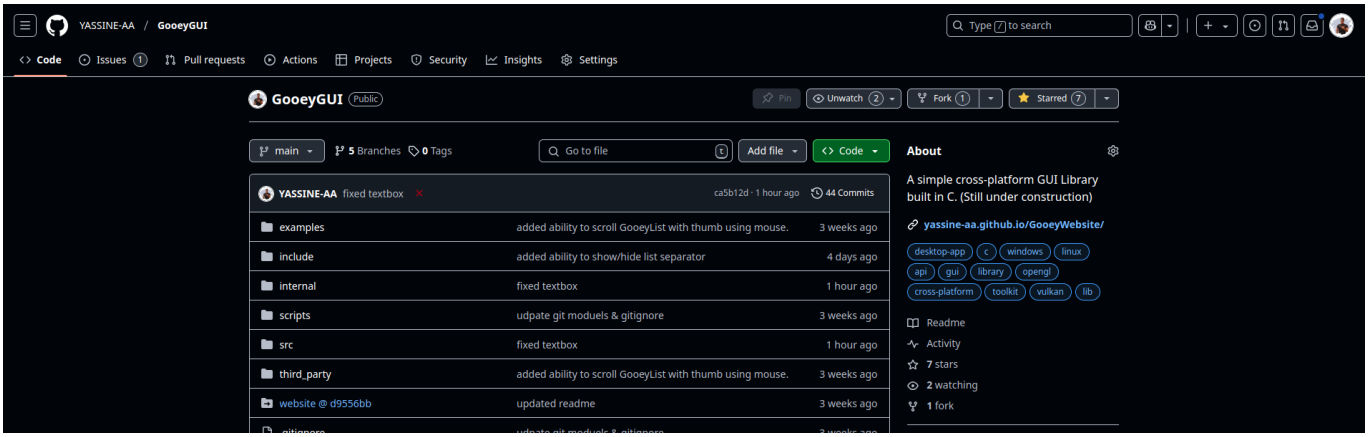
4- Bibliothèque Graphique :

4.1- Objectifs de l'intégration graphique :

L'objectif est de permettre une visualisation claire des étapes de communication réseau, et d'améliorer l'interaction avec l'utilisateur grâce à une interface simple mais fonctionnelle.

- **Affichage des messages réseau** : Recevoir et afficher en temps réel les messages envoyés et reçus via le réseau.
- **Authentification** : Offrir une interface d'authentification intuitive.

4.2- Technologie utilisée (GooeyGUI) :



Dans ce projet, nous utiliserons notre propre bibliothèque graphique developpée par **Mohamed Yassine Ahmed Ali** et **Mokhtar Sellami**, compatible avec le système Unix/Linux, Windows et MacOS. Cet bibliothèque permet de créer des fenêtres, des boutons et d'afficher des messages à l'écran, tout en étant facile à intégrer dans un projet C.

Exemple de code utilisant Gooey pour l'interface graphique :

```
    #include "gooey.h"
    #include <stdio.h>

    bool state = 0;
    GooeyWindow msgBox;

    void messageBoxCallback(int option)
    {
        LOG_INFO("MessageBox option: %d", option);
    }

    void onButtonClick()
    {
        state = !state;
        GooeyWindow_MakeVisible(&msgBox, state);
    }

    int main()
    {
        set_logging_enabled(true);
        set_minimum_log_level(DEBUG_LEVEL_INFO);

        Gooey_Init(GLFW);

        GooeyWindow win = GooeyWindow_Create("Hello World", 400, 300, 1);

        msgBox = GooeyMessageBox_Create("Hello", "Welcome to Gooey!",
MSGBOX_INFO, messageBoxCallback);
```

```
        GooeyMessageBox_Show(&msgBox);

        GooeyButton_Add(&win, "Click Me", 150, 100, 100, 40,
onButtonClick);

        GooeyWindow_Run(2, &win, &msgBox);

        GooeyWindow_Cleanup(2, &win, &msgBox);

        return 0;
    }
```

5. Compilation

```
#!/bin/sh

if [ "$#" -ne 1 ]; then
    echo "Invalid arguments"
    exit 1
fi

if [ "$1" = "server" ]; then
    gcc services/services.c utils/utils.c logger.c serveurs/serveur_auth.c
-o serveur_auth -I./  -Wall -Wextra -Wno-unused-variable -Wno-unused-
parameter -g3 -fsanitize=address,undefined -lpthread
    gcc services/services.c utils/utils.c logger.c serveurs/serveur_cat.c -
o serveur_cat -I./  -Wall -Wextra -Wno-unused-variable -Wno-unused-
parameter -g3 -fsanitize=address,undefined -lpthread
    gcc services/services.c utils/utils.c logger.c serveurs/serveur_date.c
-o serveur_date -I./  -Wall -Wextra -Wno-unused-variable -Wno-unused-
parameter -g3 -fsanitize=address,undefined -lpthread
    gcc services/services.c utils/utils.c logger.c serveurs/serveur_duree.c
-o serveur_duree -I./  -Wall -Wextra -Wno-unused-variable -Wno-unused-
parameter -g3 -fsanitize=address,undefined -lpthread
    gcc services/services.c utils/utils.c logger.c serveurs/serveur_ls.c -o
serveur_ls -I./  -Wall -Wextra -Wno-unused-variable -Wno-unused-parameter -
g3 -fsanitize=address,undefined -lpthread
    gcc services/services.c utils/utils.c logger.c proxy.c -o proxy -I./  -
Wall -Wextra -Wno-unused-variable -Wno-unused-parameter -g3 -
fsanitize=address,undefined -lpthread

    echo "Target server built."
    exit 0
elif [ "$1" = "client" ]; then
    gcc utils/utils.c logger.c clientTCP.c -o client -I./  -Wall -Wextra -
Wno-unused-variable -Wno-unused-parameter -g3 -fsanitize=address,undefined
    echo "Target client built."
    exit 0
```

```
else
    echo "Invalid target"
    exit 1
fi
```

6- Comparaisons :

Aspect	Monoclient/Monoserveur	Multiclients/Monoserveur	Multiclients/Multiservices via Proxy
Architecture	Simple client-serveur unique.	Serveur gérant plusieurs clients simultanément.	Proxy central coordonnant plusieurs services indépendants.
Gestion des clients	Un seul client connecté à la fois.	Gestion multi-clients via des threads.	Chaque client est authentifié via le proxy avant d'accéder aux services.
Synchronisation	Non nécessaire (un seul client).	Utilisation de mutex pour éviter des conflits entre threads.	Mutex utilisé pour gérer les clients et coordonner les services.
Communication	Directe entre le client et le serveur.	Directe entre chaque client et le serveur principal.	Indirecte (le proxy agit comme un intermédiaire).
Modularité	Aucune (serveur unique).	Faible (serveur gérant tout).	Très haute (services spécialisés indépendants).
Différence dans `services.c`	Gère directement les opérations du client sans modularité.	Regroupe toutes les opérations dans un serveur centralisé.	Implémente des services spécialisés (auth, date, ls, etc.), chacun dans un serveur distinct.
Avantages	- Simplicité - Faible coût de développement.	- Support multi-clients - Parallélisme natif.	- Haute scalabilité - Séparation des responsabilités - Sécurité renforcée.
Inconvénients	- Non performant - Non scalable.	- Pas d'isolation des services - Complexité accrue.	- Nécessite une coordination entre proxy et services - Maintenance plus difficile.

> L'utilisation de threads au lieu de fork pour gérer les clients dans un environnement multiclients présente plusieurs avantages significatifs, notamment en termes de performance, de gestion des ressources, et de simplicité.

Aspect	Threads	Fork
Performance	Léger et rapide	Plus lourd et lent
Mémoire	Partagée, consommation minimale	Copie indépendante, plus gourmande
Communication	Directe et simple	Nécessite des mécanismes IPC
Gestion	Simple avec des bibliothèques (Pthreads)	Plus complexe à synchroniser
Scalabilité	Gère mieux un grand nombre de clients	Limité par les ressources système