CANT'ISMIN : RAPPORT TECHNIQUE

PROGRAMMATION SYSTÈME

SEMESTRE 6

Auteur :

Émilie HOR
Youssef ENNOURI

Professeur :

SERPAGGI Xavier

Date :

Juin 2024

SOMMAIRE

INTRODUCTION

PRÉSENTATION DU PROJET

Cant'Ismin est une application Client/Serveur multi-threadé en C utilisant des sockets TCP/IP pour simuler une cantine universitaire. Notre application permet aux élèves de passer des commandes à la cantine depuis un compte étudiant en se connectant au serveur, mais également aux fournisseurs de modifier les stocks disponibles en se connectant via un compte fournisseur.

Le projet fonctionne de la manière suivante : L'utilisateur a le choix de se connecter soit en tant que client soit en tant que fournisseur.

Les élèves se connectent au serveur en tant que client à la cantine et passent une commande d'un ou plusieurs plats de leur choix selon plusieurs possibilités. Les commandes sont ensuite envoyées au serveur qui valide la commande selon la limite de stock et un message de validation est renvoyé au client. Le client a le choix de commander d'autres plats par la suite. A la fin de chaque session, un récapitulatif des plats commandés est envoyé au client.

Lorsque le client se connecte en tant que fournisseur, il peut se connecter à la cantine et modifier les stocks des plats qu'il souhaite.

BIBLIOTHÈQUES UTILISÉES

Pour éviter plusieurs erreurs, nous avons décidé de ne pas inclure toutes les bibliothèques de « `pse.h` ». En effet, si nous incluons tous les fichiers, nous allons augmenter le temps de compilation et la taille du fichier, mais également introduire des dépendances non souhaitées dans le code.

- `#include <stdio.h>`
- `#include <stdlib.h>`
- `#include <string.h>`
- `#include <unistd.h>`
- `#include <sys/types.h>`
- `#include "erreur.h"`
- `#include "resolv.h"`
- `#include "dataspec.h"`
- `#include <sys/socket.h>`
- `#include <netinet/in.h>`
- `#include <arpa/inet.h>`
- `#include <pthread.h>`
- `#include <semaphore.h>`
- `#include "ligne.h"`
- `#include "datathread.h"`
- `#include <netdb.h>`

DÉCOUPAGE DU CODE

Le code est découpé en 2 programmes principaux, `serveur.c` et `client.c`.

Fichiers Sources
- `serveur.c`
- `client.c`

Fichiers d'en-tête
- `pse.h`

Le serveur est l'élément central du projet, il gère la connexion des clients, traite les commandes, valide les stocks et communique avec les fournisseurs pour modifier les stocks. En effet, le serveur centralise toutes les données et les protocoles, il permet l'authentification des utilisateurs (clients ou fournisseurs), la gestion des commandes et la gestion des modifications de stock. Il permet également à plusieurs clients/fournisseurs de se connecter simultanément.

Le code client permet quant à la lui à une session de se connecter au serveur. Il garantit également l'affichage des informations au sein de la session et l'envoi de messages à traiter au serveur.

SPÉCIFICATIONS DU PROJET

CONFIGURATION DU SERVEUR ET DES SOCKETS

Nous avons fait le choix de créer un serveur TCP/IP multithread utilisant un modèle statique. Nous avons fixé un nombre de 100 workers, donc 100 connexions clients maximum. L'avantage de ce modèle est que l'on sait dès le démarrage si le serveur peut assurer le service mais il y a aussi l'inconvénient qu'il y aura sûrement plus de threads que nécessaire et qu'il faudra gérer l'état de sommeil et le réveil des workers.

Dans notre serveur, on initialise d'abord nos workers.La fonction `creerCohorteWorkers`() initialise les structures de données pour les threads workers et crée ces threads. Chaque worker a un sémaphore (`sem`) pour signaler quand il est libre ou occupé, un identifiant de thread (`tid`), et un mutex (`mutexCanal`) pour synchroniser l'accès à son canal.

```
void creerCohorteWorkers(void) {
    for (int i = 0; i < NB_WORKERS; i++) {
        dataSpec[i].canal = -1;
        dataSpec[i].tid = i;
        sem_init(&dataSpec[i].sem, 0, 0);
        pthread_create(&dataSpec[i].id, NULL, threadWorker, &dataSpec[i]);
        pthread_mutex_init(&mutexCanal[i], NULL);
    }
}
```

Le sémaphore `semWorkersLibres` est initialisé pour suivre le nombre de workers libres disponibles.

```
ret = sem_init(&semWorkersLibres, 0, NB_WORKERS);
if (ret == -1)
    erreur_IO("init sem workers libres");
```

On peut ensuite effectuer la configuration de la socket d'écoute. Le serveur crée une socket avec `socket(AF_INET, SOCK_STREAM, 0)`. L'adresse de la socket est configurée avec la structure `sockaddr_in`, en utilisant l'adresse IP `INADDR_ANY` et le port spécifié. La socket est liée à l'adresse configurée avec `bind()`.La socket est mise en écoute avec `listen()` pour accepter jusqu'à 5 connexions en file d'attente.

```
ecoute = socket(AF_INET, SOCK_STREAM, 0);
adrEcoute.sin_family = AF_INET;
adrEcoute.sin_addr.s_addr = INADDR_ANY;
adrEcoute.sin_port = htons(port);
bind(ecoute, (struct sockaddr *)&adrEcoute, sizeof(adrEcoute));
listen(ecoute, 5);
```

GESTION DES CONNEXIONS ET DES THREADS

Le serveur entre dans une boucle infinie où il accepte les connexions entrantes avec `accept()`. Pour chaque connexion acceptée, il attend qu'un worker soit libre en utilisant `sem_wait(&semWorkersLibres)`. Une fois un worker libre trouvé, le serveur assigne le canal de communication (`canal`) au worker et réveille ce dernier en postant sur son sémaphore.

```
while (VRAI) {
    canal = accept(ecoute, (struct sockaddr *)&adrClient, &lgAdrClient);
    if (canal < 0)
        erreur_IO("accept");

    ret = sem_wait(&semWorkersLibres);
    if (ret == -1)
        erreur_IO("wait sem workers libres");

    numWorkerLibre = chercherWorkerLibre();
    dataSpec[numWorkerLibre].canal = canal;
    sem_post(&dataSpec[numWorkerLibre].sem);
}
```

On cherche ensuite un worker libre, la fonction `chercherWorkerLibre()` parcourt la liste des workers pour trouver un worker dont le canal est `-1` (indiquant qu'il est libre).

```c
int chercherWorkerLibre(void) {
    for (int i = 0; i < NB_WORKERS; i++) {
        lockMutexCanal(i);
        if (dataSpec[i].canal == -1) {
            unlockMutexCanal(i);
            return i;
        }
        unlockMutexCanal(i);
    }
    return -1;
}
```

La fonction principale du worker s'exécute ensuite. Chaque worker exécute la fonction `threadWorker()`.Le worker attend d'être réveillé par le sémaphore (`sem_wait(&dataTh->sem)`). Une fois réveillé, il lit le message de bienvenue et traite la réponse du client pour démarrer une session client ou fournisseur.

```c
void *threadWorker(void *arg) {
    DataSpec *dataTh = (DataSpec *)arg;

    while (VRAI) {
        sem_wait(&dataTh->sem);

        ...

        lockMutexCanal(dataTh->tid);
        dataTh->canal = -1;
        unlockMutexCanal(dataTh->tid);

        sem_post(&semWorkersLibres);
    }

    pthread_exit(NULL);
}
```

GESTION DES MESSAGES

La gestion des messages se fait au niveau de client.c et est particulière dans notre application. En effet, le serveur est susceptible d'envoyer un ou plusieurs messages par communication. Le client doit donc être capable de lire tous les messages qui sont envoyés quel que soit leur nombre. Pour résoudre ce problème nous avons opté pour la détection d'une chaine de caractère spécial qui permet de dire au client que c'est la fin de l'envoie. Cette chaine de caractère est « `\n\n` » et est envoyé après chaque message de la sorte :

```c
if (write(canal, "\n\n", 2) < 0)
                erreur_IO("write");
```

De même, la fin de la session est détectée à l'aide d'une autre chaine de caractère spéciale qui est « *** ». Ainsi, après que le message de fin soit envoyé, on envoie cette chaine de caractère. Lorsque le client reçoit cette chaîne de caractères, il ferme la session et la communication avec le serveur.

```
char Fermeture[] = "***";
if (write(canal, Fermeture, strlen(Fermeture)) < 0)
        erreur_IO("write");
```

Les chaines de caractères sont détectées par le client à l'aide d'une boucle `while`, le client lit le message et l'affiche jusqu'à trouver une des chaines de caractères et agit en conséquence.

```
while ((ret = read(sock, buffer, sizeof(buffer) - 1)) > 0) {
        buffer[ret] = '\0';
        printf("%s", buffer);
        strncat(reponse, buffer, ret);
        reponse_longueur += ret;
        // Vérifier si la réponse complète a été reçue
        if (buffer[ret - 1] == '\n' && buffer[ret - 2] == '\n') {
            break;
        }
        else if (strstr(buffer, "***") != NULL) {
            fin = VRAI;
            break;
        }
    }
```

Ainsi, le client reste lire les messages jusqu'à qu'il reçoive la chaine de caractère permettant de savoir si le message ou la session est finie.

AUTHENTIFICATION DES UTILISATEURS

L'authentification des utilisateurs est une étape primordiale au bon fonctionnement du projet car elle permet l'accès aux différentes fonctionnalités du programme. La connexion se fait directement dans les fonctions `sessionClient()` et `sessionFournisseur()`.
L'authentification des fournisseurs et des clients est similaire, l'utilisateur entre son identifiant, si l'utilisateur a demandé à se connecter en tant que client, l'identifiant est recherché dans `identifiant_client.txt`, s'il a demandé à se connecter en tant que fournisseur l'identifiant est recherché dans `identifiant_fournisseur.txt`.

Si l'identifiant entré correspond à l'un des identifiant dans le fichier texte associé alors la connexion est autorisée, sinon l'entrée est invalidée et il est demandé à l'utilisateur de rentrer à nouveau l'identifiant.

Pour les clients, il y a également la possibilité de créer un compte. Les utilisateurs peuvent entrer l'identifiant qu'ils veulent, il sera alors ajouté dans à la fin du fichier texte et pourrons directement se connecter avec. La création de compte se fait à l'aide de la fonction `creerCompte()` qui permet d'ouvrir le fichier identifiant_client.txt et d'écrire à la fin de celui-ci le nouvel identifiant.

Les identifiants sont vérifier à l'aide des fonctions `vérifierIdentifiantClient()` et `vérifierIdentifiantFournisseur()` qui parcourent les fichiers textes à la recherche de l'identifiant pris en argument. Si l'identifiant est trouvé, alors la fonction retourne 1, sinon 0. La valeur de retour de la fonction est utilisée pour vérifier si on peut quitter la boucle de vérification de l'identifiant. Si elle passe à 1 alors l'identifiant a été trouver et on peut sortir du `while`.

```
while (!identifiantValide) {
    //...
    //...
    if (verifierIdentifiantClient(identifiant)) {
        identifiantValide = 1;
        //...
        //...
        }
    else{
        //identifiant invalide
        //...
        }
```

GESTION DES COMMANDES ET DES STOCKS

Pour une session client :

La gestion des commandes et des stocks pour le client se fait, elle, aussi via plusieurs fichiers. Ces opérations sont gérées par la fonction `gestionstock()`. Les plats ainsi que leurs stocks sont enregistrés dans le fichier `nourriture.txt`. À chaque plat est également associé un numéro d'identification qui sert de clé primaire rendant chaque plat exclusif.

Lorsqu'un client choisit un plat, son stock est diminué de 1 et ajouter à la commande de l'utilisateur. Le client a la possibilité de commander autant de plat qu'il le souhaite tant qu'il ne clôture pas sa session. Dès qu'un client commande un ou plusieurs plats, un nouveau fichier texte exclusif à chaque client est créé. Ce fichier stocke donc les plats commandés pour chaque client et permet d'afficher un récapitulatif de la commande d'un client. Le fichier porte alors le nom du client. Ainsi, si c'est le client émilie qui commande, un fichier `commande_émilie.txt` est créé.

```c
int enregistrerPlatChoisi(char *identifiant, char *plat) {
    //...
    snprintf(nomFichier, sizeof(nomFichier), "Projet/commande_%s.txt", identifiant);
    FILE *fichierCommande = fopen(nomFichier, "a");
    //...
    fprintf(fichierCommande, "%s\n", plat);
    fclose(fichierCommande);
    return 1;
}
```

Comme indiqué précédemment, à la fin de chaque session, le fichier est lu et afficher par la fonction `affichercommande()` qui montre à chaque utilisateur un récapitulatif de sa commande.

```c
int affichercommande(int canal, char *identifiant) {
    //...
    FILE *fichierCommande = fopen(nomFichier, "r");
    //...
    while (fscanf(fichierCommande, "%s", nom) != EOF) {
        int n = snprintf(message, sizeof(message), "Plat : %s\n", nom);
        if (n < 0) {
            erreur_IO("snprintf");
            return 0;
        }

        if (write(canal, message, n) < 0) {
            erreur_IO("write");
            return 0;
        }
    }

    fclose(fichierCommande);
    return 0;
```

Pour une session fournisseur :

La gestion pour le fournisseur est similaire, il peut modifier le stock de chaque plat pour en augmenter le nombre ou le diminuer. Pour modifier le stock d'un plat on utilise la fonction `gestionstock_fournisseur()` qui parcours tout le fichier `nourriture.txt` jusqu'à trouver le numéro correspondant au plat qu'on veut modifier.

```c
int gestionstock_fournisseur(int canal, char *numeroplat, char *augmentation, char *identifiant) {
    FILE *fichier = fopen("Projet/nourriture.txt", "r+");
    //...

    while (fgets(buffer, sizeof(buffer), fichier) != NULL) {
        pos = ftell(fichier);
        sscanf(buffer, "%s %d %d", nom, &id, &stock);
        if (atoi(numeroplat) == id) {
            //...
            stock += atoi(augmentation);  // Ajoute la quantité spécifiée au stock
            found = 1;
            fseek(fichier, pos - strlen(buffer), SEEK_SET);
            fprintf(fichier, "%s %d %d\n", nom, id, stock);
            fflush(fichier);
            //...
            break;
        }
    }

    //...
```

De même, à chaque fois qu'une modification sur un stock est faite par le fournisseur celle-ci est enregistrer dans un fichier incluant le fournisseur. Ainsi, si le fournisseur youssef fait une modification, un nouveau fichier `modifications_youssef.txt` sera créé et ses modifications écrites dedans.

D'autres fonctionnalités sont possibles, on peut utiliser ajouter un nouveau plat avec `ajouterPlat()`. Pour ajouter le plat qu'on veut on entre juste son nom et son stock initial, il est alors écrit à la fin du fichier `nourriture.txt` et son numéro d'identification est calculé automatiquement comme le numéro d'identification du plat au-dessus + 1.

De même, on peut retirer un plat avec `retirerPlat()` en entrant son numéro, un autre fichier nourriture.txt est alors créer où l'on copie tous les plats sauf celui que l'on veut supprimer, on obtient alors un fichier similaire sans le plat qu'on a décidé de supprimer.

Enfin, à la fin de chaque session on affiche également un récapitulatif de chaque modification.

SESSION CLIENT

`sessionClient()` est la fonction principale qui va permettre de gérer la communication avec le client et la logique général du programme en appelant les différentes fonctions. La session Client permet les fonctionnalités ci-dessous dans l'ordre :

- Connexion ou création de compte :
  - La fonction affiche un message de bienvenue et demande au client de choisir entre se connecter ou créer un compte.
  - Selon le choix du client, il appelle creerCompte(canal) pour créer un compte ou continue avec la connexion.
  - Si la création de compte échoue, un message d'erreur est envoyé et la session se termine.
- Demande de saisie de l'identifiant :
  - La fonction demande au client de saisir son identifiant.
- Vérification de la validité de l'identifiant :
  - La validité de l'identifiant est vérifiée en appelant verifierIdentifiantClient(identifiant).
  - Si l'identifiant est valide, le client est informé de sa connexion réussie.
- Création d'un fichier de commande pour le client connecté :
  - Un fichier de commande spécifique au client est créé pour enregistrer ses choix.
- Choix d'un plat et mise à jour du stock :
  - La fonction demande au client de choisir un plat et affiche la liste des plats disponibles en appelant afficherplat(canal).
  - Le choix du client est lu et traité, puis le stock est mis à jour en appelant gestionstock(canal, identifiant, numero_plat).

- Demande au client s'il souhaite commander un autre plat :
    - La fonction demande au client s'il souhaite commander un autre plat et traite la réponse en appelant gestionreponse(canal, identifiant, reponsefinale).
- Gestion de la déconnexion du client :
    - La fonction gère la déconnexion du client en attendant qu'il envoie le message de fin de session.

Cette fonction utilise des messages pour interagir avec le client à chaque phase, un message textuel est envoyé à l'utilisateur en déclarant un char et en l'envoyant à l'utilisateur à l'aide de la fonction `write` sans oublier « `\n\n` » pour dire le message est terminé.

```
char Message[] = "Ceci est un message";
if (write(canal, Message, strlen(Message)) < 0)
    erreur_IO("write");
if (write(canal, "\n\n", 2) < 0) {
    erreur_IO("write");
}
```

La réponse est ensuite lue et traitée à l'aide de la fonction `lireLigne()` :

On remplace le dernier caractère de la chaine de caractère qui est un saut de ligne et on le remplace par « `\0` » pour dire que la chaine de caractère est fini.

```
lgLue = lireLigne(canal, Message);
if (lgLue < 0){
    erreur_IO("lireLigne");
}
Message[lgLue - 1] = '\0';
```

La réponse est alors stockée dans Message.

Nous utilisons dans notre fonction plusieurs boucle while pour vérifier que l'entrée de l'utilisateur et l'entrée attendue correspondent bien, si ce n'est pas le cas on redemande à l'utilisateur de rentrer sa réponse. Lorsqu'on reçoit la bonne réponse, on peut alors effectuer une action comme lancer une fonction.

SESSION FOURNISSEUR

`sessionFournisseur()` est la fonction principale qui va permettre de gérer la communication avec le fournisseur. Cette fonction est similaire à `sessionClient()`, mais adaptée pour les fournisseurs.

Le début de la fonction `sessionFournisseur()` permet de faire le choix entre continuer en tant que fournisseur ou en tant que client et lancer `sessionClient()`. La logique pour se choix est une boucle comme expliquer précédemment.

La `sessionFournisseur()` effectue les actions suivantes :

- **Demande de saisie de l'identifiant :**
  - La fonction demande au fournisseur de saisir son identifiant et vérifie sa validité avec verifierIdentifiantFournisseur(identifiant).
  - Si l'identifiant est incorrect, le fournisseur est invité à réessayer jusqu'à ce qu'un identifiant valide soit fourni.
- **Menu des actions disponibles :**
  - Une fois l'identifiant validé, un menu est affiché pour permettre au fournisseur de choisir entre :
    - Modifier le stock d'un plat.
    - Ajouter un plat.
    - Retirer un plat.
- **Modification du stock d'un plat :**
  - La fonction demande au fournisseur le numéro du plat dont il souhaite modifier le stock.
  - Ensuite, elle demande la quantité à ajouter au stock.
  - Le stock est mis à jour en appelant gestionstock_fournisseur(canal, numero_plat, augmentation, identifiant).
  - Le fournisseur est ensuite invité à modifier le stock d'un autre plat, et le processus se répète jusqu'à ce qu'il réponde "non".
- **Ajout d'un plat :**
  - La fonction demande le nom du plat à ajouter et le stock initial.
  - Le plat est ajouté en appelant ajouterPlat(canal, identifiant).
- **Retrait d'un plat :**
  - La fonction demande l'identifiant du plat à retirer.
  - Le plat est retiré en appelant retirerPlat(canal, identifiant).
- **Gestion de la déconnexion du fournisseur :**
  - La fonction gère la déconnexion du fournisseur en attendant qu'il envoie le message de fin de session ("fin").

L’ensemble des fonctions pour vérifier les identifiants, gérer les stocks et envoyer les messages ont déjà été expliqués précédemment et ne nécessite pas d’explications supplémentaires.

MAKEFILE

Le fichier Makefile permet simplement de compiler le projet en utilisant la commande make. Il est décomposable en cinq parties :

1. Variables de configuration :

    - CC : Spécifie le compilateur à utiliser, ici gcc.
    - CFLAGS : Définit les options de compilation, telles que l'inclusion des avertissements (-Wall, -Wextra) et la spécification des chemins pour les fichiers d'en-tête (-I./Librairie).
    - LDFLAGS : Définit les options pour lier les bibliothèques, en indiquant où les trouver (-L./Librairie).
    - LIBS : Indique qu'on doit lier la bibliothèque mathématique (-lm).

2. Chemins des dossiers :

    - LIB_DIR : Dossier contenant les bibliothèques (Librairie).
    - MOD_DIR : Dossier contenant les modules (fichiers .c) (Modules).
    - SRC_DIR : Dossier contenant les fichiers source principaux (Projet).
    - BIN_DIR : Dossier où les exécutables seront créés (le répertoire courant .).

3. Fichiers objets des modules :

    - MOD_OBJS : Génère une liste de fichiers objets (.o) pour tous les fichiers source (.c) présents dans le dossier des modules (Modules). Cette liste est créée en utilisant la fonction patsubst et la commande wildcard pour rechercher tous les fichiers .c et les convertir en .o.

4. Cibles à générer :

    - TARGETS : Définit la liste des programmes à compiler, à savoir serveur et client. Ces cibles spécifient les exécutables finaux du projet.

5. Règles de compilation :

    - all : La cible par défaut qui compile tous les exécutables définis dans TARGETS. Lorsque vous exécutez make, cette règle est invoquée.
    - $(BIN_DIR)/serveur et $(BIN_DIR)/client : Spécifient comment compiler les programmes serveur et client en utilisant les fichiers objets nécessaires. Les fichiers objets sont compilés et liés pour créer les exécutables dans le dossier BIN_DIR.

- Règles pour les fichiers objets : Indiquent comment compiler les fichiers source (.c) en fichiers objets (.o) pour les répertoires SRC_DIR et MOD_DIR.
- clean : Supprime tous les fichiers objets et les exécutables générés, permettant de nettoyer le projet pour repartir d'une compilation propre

DIFFICULTÉS RENCONTRÉES

Nos principales difficultés ont principalement résulté en de nombreux bugs rencontrés dans la gestion des threads et du multithreading. En particulier, le choix des sessions a été une source majeure de problèmes et nous avons eu énormément de mal à débuguer cette partie du code. La gestion des messages a également posé de nombreux problèmes : il a fallu trouver un moyen efficace de gérer l'envoi de tous les messages et de déterminer comment envoyer un message pour fermer correctement la session. De plus, La manipulation des fichiers et des chaînes de caractères a souvent causé des erreurs imprévues. En conséquence, nous avons eu beaucoup de difficultés à faire fonctionner ce projet correctement, sachant que nous avons passé environ 80 % du temps à débuguer le code.