03/05/2023

COMPTE RENDU DE PROJET

Simulation d'un bar électronique

Christoph Samuel - Mouche Valentin

COMPTE RENDU DE PROJET

Simulation d'un bar électronique

Table des matières

Introduction

Dans ce projet, nous avons implémenté **une simulation d'un bar électronique,** composé d'un **barman**, de **deux tireuses à bière** (une de blonde et une ambrée), et d'un ou **plusieurs clients**. Le barman, le fournisseur et les clients sont tous des programmes informatiques exécutés sur **différentes machines** en réseau les uns avec les autres. Le but de cette simulation est de **gérer les demandes de boissons des clients,** de préparer les boissons avec les tireuses, **de gérer les stocks de bières** et de passer commande auprès d'un **fournisseur** en cas de rupture de stock.

Architecture et implémentation

Nous avons divisé notre implémentation en **plusieurs modules** pour faciliter la conception et la **maintenance du code.** Voici un bref aperçu des modules que nous avons développés.

Module Barman

Il s'agit du **processus principal** qui gère **les demandes des clients** via communication.c qui permet la **communication entre clients.c et main.c** . Le fichier communication.c communique avec le ou les clients via socket TCP, lorsque **le client fait choix** d'afficher les informations disponibles sur les bières (choix 1 dans le menu côté client) ou le choix 2 (commander une bière) , son choix est envoyé **via socket TCP** au fichier communication.c, ce choix est **enfin transféré via pipe** (tube nommé) au main.c pour être analyser et traiter, en plus du choix, communication.c écrit aussi l'identifiant du client (id qu'il crée **pour pouvoir reconnaître les différents clients** et les différencier), le type de bière qu'il veut et la quantité (tout 2 égal à 0 si le client veut seulement les informations sur les bières). Au niveau du main.c on va lire le pipe **pour récupérer les informations sur le client** et savoir ce que l'on va faire. Pour le choix 1 on va simplement réécrire sur le pipe les informations de nos bières, les concaténer dans un char et renvoyer ce char au fichier communication.c via pipe, qui se chargera de le renvoyer **via la socket TCP** au client.c, une fois reçu le client se chargera de récupérer le char et de l'afficher.

```
+----------------------------------------+
|           Contenu des tireuses         |
+----------------------------------------+

            Tireuse 1 - Blonde
               Nom : Goudale
               Degré : 10°
               Quantité : 5000

            Tireuse 2 - Ambrée
               Nom : Kwak
               Degré : 5°
               Quantité : 5000

+----------------------------------------+
```

Lorsque le choix est égal à 2 c'est un peu plus complexe, on va **récupérer la commande du client** (type de bière et quantité) ainsi que son identifiant. On va par la suite stocker ces données **dans une pile via un ordonnanceur pour exécuter en priorité le premier arrivé** (FCFS). Une fois l'ordre de passage établie, le premier client est servi via un appel à **la fonction servir()** qui prend en paramètre la quantité (en ml) et le **type de bière.**

Une fois la bière servie, **un message de confirmation** est envoyé au client pour l'informer que sa bière est prête (ou non selon l'état des futs) et les futs de bière via les tireuses sont **mises à jour pour supprimer la quantité servie à la quantité restante du fût.** Le barman vérifie aussi l'état des fûts et envoie **des commandes au fournisseur** en cas de besoin.

Module Tireuses

Nous avons créé des **classes pour les deux types de tireuses** (blonde et ambrée), qui sont associées à des fûts de 5 litres, convertis en 5000mL pour être **plus pratiques.** Chaque tireuse est représentée par une **entrée dans la mémoire partagée (SHM)** qui indique le **type et le nom** de la bière, ainsi que la quantité disponible dans le fût.

Module Fournisseur

Le fournisseur est représenté par un programme Java qui communique avec **le processus commande du barman via Java RMI**.

Il est accompagné de la classe Biere.java et de l'interface IBiere.java qui servent a définir les ***différents types et structures d'une bière***. Nous avons également développé **une interface utilisateur (IHM)** pour permettre la **manipulation du barman et des clients**, ainsi que le suivi de la simulation en temps réel pour la barman (états des futs).

```
+-------------------------------------------------------------------------+
|         Liste des bières disponibles chez le fournisseur:               |
+-------------------------------------------------------------------------+

-Blondes:
 - Paix Dieu (10°)
 - Goudale (7°)
 - Delirium Tremens (8°)
-Ambrées:
 - Kwak (8°)
 - Mousse Ta Shuc (6°)
 - Queue de Charrue (5°)
```

Gestion des erreurs et qualité du code

Nous avons porté une attention particulière à **la gestion des erreurs dans notre implémentation**. Par exemple, nous avons pris en compte les cas où **un fût est vide**, où une **demande de bière ne peut être satisfaite**, ou encore où la communication entre les différents processus est interrompue…

La gestion des signaux est aussi présente

Notre code est **bien structuré et organisé,** avec une séparation claire des responsabilités entre les **différents modules.** En ce qui concerne **les choix de conceptions** et d'implémentations, l'ensemble de nos codes est fournis de **commentaire** afin, nous avons de ne pas devoir tout détailler sur ce compte rendu **pour faciliter la compréhension et la maintenance du code**.

Compilation et exécution du projet

Afin de rendre possible la **simulation de notre projet,** nous avons réuni l'ensemble des **commandes de compilations** à effectuer dans un fichier. On adapte donc **les commandes** suivante de sorte à les utiliser sur **plusieurs machines** (make all, make rmi, make fournisseur) pour le fournisseur, (make all, make bar) pour le bar et (make all, make clients) pour les clients.

make all : permet la compilation de l'ensemble des fichiers utiles a notre projet,

make rmi ouvre un terminal afin d'exécuter un rmiregistry,

make fournisseur: lance le fournisseur afin de rendre possible la commande de bière,

```
ifconfig
eno1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.1.13.180  netmask 255.255.255.0  broadcast 10.1.13.255
        inet6 fe80::a6bb:6dff:fe5a:8323  prefixlen 64  scopeid 0x20<link>
        ether a4:bb:6d:5a:83:23  txqueuelen 1000  (Ethernet)
        RX packets 888561  bytes 1275396618 (1.2 GB)
        RX errors 0  dropped 912  overruns 0  frame 0
        TX packets 367081  bytes 203761410 (203.7 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
        device interrupt 20  memory 0xa5200000-a5220000

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Boucle locale)
        RX packets 236  bytes 21440 (21.4 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 236  bytes 21440 (21.4 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

attention au niveau du fichier fournisseur afin que la bonne adresse IP soit analysée il faut faire attention que le paramètre d'adressage soit le bon voir exemple ci-dessous dans notre cas il est adapté pour la salle B220 du réseau de l'UPPA, **ici eno1**.

```java
public static void main(String[] args) {
    Fournisseur fournisseur = null;
    try {
        NetworkInterface ni = NetworkInterface.getByName(name:"eno1");
        Enumeration<InetAddress> inetAddresses = ni.getInetAddresses();
        InetAddress localAddress = null;
        while (inetAddresses.hasMoreElements()) {
            InetAddress ia = inetAddresses.nextElement();
            if (ia instanceof Inet4Address && !ia.isLinkLocalAddress()) {
                localAddress = ia;
                break;
            }
        }
```

```
# Variables pour les adresses IP
HOST_BAR = localhost
HOST_FOURNISSEUR = localhost
```

Avant de poursuivre, on **met à jour nos variables dans le Makefile** de sorte pour avoir une compilation beaucoup dynamique, de plus sans cela la simulation ne fonctionnera pas…

make bar : exécute le bar et permet aux clients de s'y connecter,

make clients: permet de connecter un clients si le bar est ouvert,

Répartition du travail

En ce qui concerne la **répartition du travail** Valentin a pris en charge la **communication TCP entre les clients et le processus de communication** via la socket, ainsi que la communication entre **le processus de communication et le processus principal via les pipes**. Il a également géré l'ordonnanceur dans le fichier main.c.

De son côté, Samuel a mis en place la **connexion UDP entre les processus de commande et de contrôle,** ainsi que la **composante Java RMI avec le fournisseur**. Il a également contribué de manière significative à la **fonctionnalité de la mémoire partagée,** que Valentin a utilisée de **manière honteuse**. Lorsqu'ils ont combiné leurs contributions respectives, ils ont rencontré quelques problèmes **résolut avec rigueur.** De plus, Samuel a mis en **place l'affichage pour le ni-bar** et le **processus de remplissage de la bière.**

Conclusion

Notre implémentation de la **simulation d'un bar électronique** répond a la quasi-totalité des **exigences du projet** et fonctionne de manière efficace. Grâce à une architecture **modulaire** et **une gestion des erreurs,** notre code est facile à comprendre et à maintenir. Nous sommes satisfaits de notre travail et pensons qu'il répond aux attentes en **matière de fonctionnement nominal,** de gestion des erreurs et de qualité du code, même si avec un peut plus d'organisations nous aurions pu **régler quelques détails.**

Nous avons apprécié travailler sur ce projet même si complexe, il nous a beaucoup appris sur la **gestion des ressources partagées,** les communications réseau et les **systèmes distribués.** Nous sommes impatients de présenter notre travail et d'expliquer plus en détail **nos choix de conception et d'implémentation.**