Sidi Mohammed AIDARA
Nabil AISSAT
Abdelnour SASSI
Noureddine ZIANI

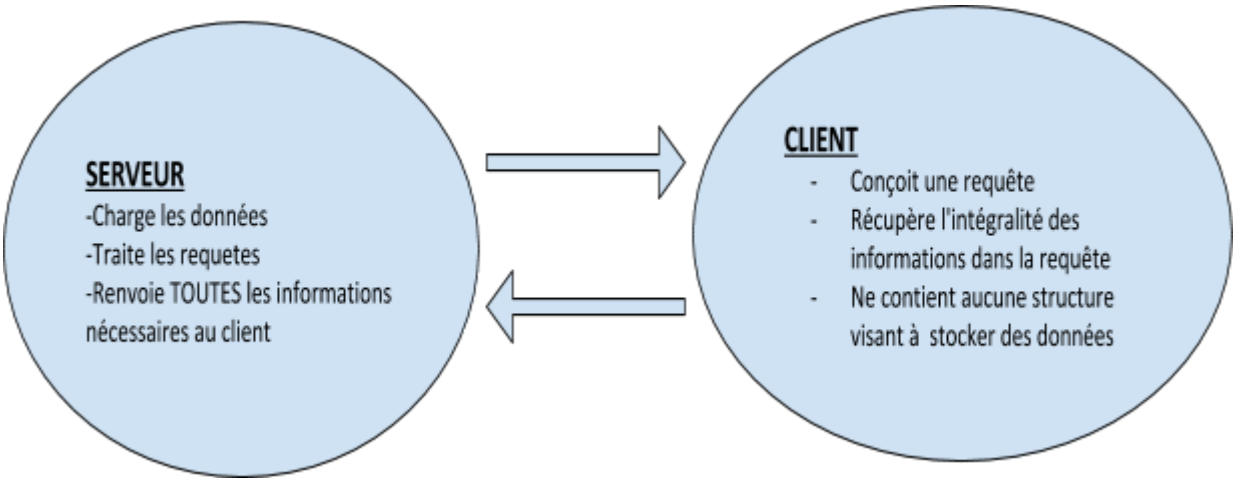
PROJET SYSTÈME RÉSEAUX

Laurence PIERRE

Table des matières :

INTRODUCTION

Nous avons conçu ce projet réseau avec beaucoup de passion, ainsi nous avons essayé de concevoir un programme solide avec une vision modulaire des différentes fonctions et structures. Pour ce qui est du partage des tâches, nous avons dans un premier temps réfléchi à l'architecture de la partie client puis celle de la partie serveur. Nous sommes arrivés à la conclusion d'avoir une partie cliente assez hermétique (ne contient pas de structure, ou autre) et une partie serveur qui contient toutes les données.



Pour chaque composant (notamment le serveur et le client) du programme il y a un dossier du même nom qui contient les dossiers suivants :

- bin -> dossier contenant les exécutables compilés.
- headers-> fichiers headers contenant table.h pour les composants qui l'utilisent.
- sources -> fichiers de source c et table.c pour les composants qui l'utilisent.

ORGANISATION DU TRAVAIL

VENDREDI	**Mise en place du serveur/client TCP IP (PARTIE I)**
LUNDI	**Mise en place du serveur/client TCP IP (PARTIE II) / Listing des fonctions / Format des données / Autres améliorations**
MARDI	**Début des travaux / Etablissement de toutes les fonctions serveur / Test**
MERCREDI	**Etablissement de toutes les fonctions client (IHM etc.)/ Test / Debut de création du Makefile / Début d'intégration**
JEUDI	**Finalisation de l'intégration / Derniers tests / Début de rédaction du rapport**

CAHIER DES CHARGES

Pour ce qui est des données on a choisit 2 structures. Une qui contient toutes les données d'un train , et l'autre qui contient un tableau de trains ainsi que sa taille.

```
typedef struct unTrain
{
	char ville_Depart[TAILLE_BUFFER];
	char ville_Arrivee[TAILLE_BUFFER];
	char heure_Depart;
	char heure_Arrivee;
	char prix;
	char option[TAILLE_BUFFER];
	char prix_option[TAILLE_BUFFER];
}train;
typedef struct trains
{
	struct unTrain train[TAILLE_LISTE];
	int taille;
}struct_train;
```

On remarque que toutes les informations sont stockées sous forme de chaînes de caractères pour des soucis de standardisation et de facilité d'affichage, les opérations qui nécessitent ainsi des nombres vont faire appel à des fonctions telles que atof() atoi() entre autres.

Les requêtes clients doivent respecter le protocole suivant

VilleDepart > VilleArrivée > Id_OPERATION > INFO1 > INFO 2….>

Id_OPERATION correspond à :

1 : Si le user demande d'un train à une heure précise

2 : Une liste de trains correspondant à une fourchette horaire

3: La liste de tous les trains

Les réponses du serveur se présentent sous un format assez semblables:
Nombre de trains(ou Code d'erreur) >TRAIN1 / TRAIN2 / …./TRAIN-N où TRAIN-i est :
VilleDepart > VilleArrivée > Heure_depart> heure_d'arrivée > prix >promo > prix après promo

CONCEPTION PARTIE SERVEUR

Architecture du serveur:

Le fichier server.c contient le programme principal qui va donc établir la connexion avec le client et communiquer avec lui , ainsi plusieurs étapes naturelles se font :

1) La création d'une socket , l'attachement à une structure adresse puis son écoute
2) La primitive accept en aval de la boucle while et du switch fork qui attendre une nouvelle connexion entrante.
3) Si accept() réussi on lance la création d'un processus fils qui va exécuter la fonction "***comportement_serveur()***"'.
4) La réception de la chaine de caractere se fait par la fonction "recv"
5) La fonction "**comportement_serveur()**" va donc traiter la requête du client et tourner tant que le client ne s'est pas deconnecté.
6) Si le client se déconnecte, on sort de "***comportement_serveur()***" et on tue le processus puis on attend une possible nouvelle connexion.

Dans la fonction comportement_server(int socket_client):

Nous avons créé cette fonction toujours dans cette vision modulaire , par conséquent nous pouvons imaginer d'autres comportement possible pour le même serveur, pour ce qui est du corps de la fonction nous avons procédé ainsi :

On déshabille la requête avec la fonction décomposer (cf : fonctions utilitaires pour le serveur) , puis on charge les donnés situés du le fichier Trains.txt vers une liste (une structure contenant un tableau de structure train et une taille), qui sera retourné par load_data(nom_du_fichier). Après cela on sélectionne les trains correspondant aux villes choisis par l'intermédiaire de city-opération qui va nous renvoyer une nouvelle liste avec les bons trains (si la taille nul c'est que la destination n'est pas disponible). Puis on trie cette dernière liste par horaire de départ (tri croissant).

Après toute ces étapes intermédiaires on a donc une liste trié contenant les même villes correspondant à la requête du client , on peut passer à la fonction traiter_requête qui suivant le numéro de l'opération , va appeler les fonctions operation_req1 (operation 1) operation_req2 etc… ces fonctions retourne une liste struct_train , ou seulement un train, traiter requête se chargera de de construire des messages "erreurs" (PASDETRAIN , ERREUR_SWITCH etc)

Architecture du fichier utilitaires pour serveur:

```
void decomposer(char chaine[SIZE_LIGNE], char smart[SIZE_MORCEAU][TAILLE_BUFFER], char coupeur[]);
struct_train load_data(char path_name[]);
struct_train city_operation(char requete_d[NM_REQUETE][TAILLE_BUFFER], struct_train liste);
struct_train tri_horaire(struct_train liste_recu);
train* operation_req_1(char requete_d[NM_REQUETE][TAILLE_BUFFER], struct_train liste_r_sorted);
struct_train operation_req_2(char requete_d[NM_REQUETE][TAILLE_BUFFER], struct_train liste_r_sorted);
struct_train operation_req_3(char requete[NM_REQUETE][TAILLE_BUFFER], struct_train liste_r_sorted);
char* traiter_requete(char requete[NM_REQUETE][TAILLE_BUFFER], struct_train liste);
```

La majorité des fonctions opère sur des listes de train, mais ici nous nous attarder sur la fonction décomposer , la fonction décomposer a été développé durant le projet Système précédent , cette fonction permet de decouper une chaine de caractère, avec un coupeur au choix(ex ">", ça peut même être une chaîne !) , et le stocke dans un tableau 2D , cette fonction est utilisé dans plusieurs endroit dans le projet d'où son importance.

Pour conclure cette partie , on a un fichier serveur.c qui communique avec le client et utilise tout un listing de fonction développés dans le fichier util_server.c

CONCEPTION PARTIE CLIENT

- **Architecture du client:**

Le fichier client.c contient la fonction qui permet d'établir une communication avec notre serveur. En pratique, le client effectue un certain nombre d'étapes, notamment:

1. Création d'une socket et binding.
2. Demande de connexion grâce à une primitive ***connect();***
3. Préparation de la requête grâce à la fonction ***interaction();***
4. Envoi de la requête au serveur via la primitive ***send();***
5. Réception de la réponse du serveur avec ***receive();***
6. Traitement de la réponse grâce à une fonction ***affichage_Resultat();***
7. Préparation d'une requête pour un tri sur le dernier résultat grâce à une fonctoin ***preference_User();***
8. Rappel de la fonction ***affichage_Resultat()*** sur la deuxième réponse du serveur.

PS: Etant donné que le tri est possible selon les opérations effectuées(possible avec opéartions 2 et 3), un cocktail de test est effectué sur les retours de notre fonction affichage_Resultat() et de la valeur d'une variable flagPreference afin d'assurer une bonne logique dans notre application.

- **Présentation de la fonction affichage_Resultat(char *reponseServer)**

Cette fonction prend en paramètre une réponse du serveur et fait appel à **decomposer();** Après décomposition sur ligne grâce au séparateur "***/***" pour chaque TRAIN et ensuite sur un séparateur "**>**" pour chaque donnée du train, on joue ensuite sur l'affichage pour mettre les informations reçues du serveur à la disposition du client de manière claire et bien détaillée. Pour rappel, toujours dans cette logique d'optimisation, nous nous sommes organisés à ne pas avoir de structure de données côté client. Cette fonction ne se limite donc qu'à un simple affichage amélioré et poli d'une réponse du serveur.

- **Présentation de la fonction interaction(char *ville_D, char *ville_A): char ***

C'est la fonction qui assure la réelle utilisation de notre application côté client. L'objectif de cette fonction est de, grosso-modo, polir au maximum la saisie des informations sur une recherche de train pour respecter le protocole que nous avons établi entre notre serveur et notre client (format de données, modèle requête entre autres cf **PARTIE CAHIER DES CHARGES**).
Elle prend en paramètres une ville de départ et une ville d'arrivée, elle la transmet ensuite à des sous-fonctions de traitement (les fonctions d'opérations cf annexe) qui correspondent chacune à une opération du cahier de charges qui nous a été remis lors de l'entame de ce projet.
Cette fonction retourne une chaîne de caractère: **la requête du client**.

CONCLUSION

Durant toute la période qui nous a été dédiée pour atteindre cet objectif de communication SERVER - CLIENT en mode TCP, nous pensons fermement que l'aspect le plus excitant a été notre manière de travailler en groupe. Notre projet respecte la quasi-totalité du cahier de charges à quelques erreurs près (indépendantes de notre volonté mais plutôt relatives aux systèmes des machines de l'école); à noter aussi que ces erreurs sont extrêmement rares.

S'il y a des améliorations à apporter, la première et probablement la plus importante dans le court terme est de permettre à l'utilisateur de lui même décider de quitter la connexion (boucler plus techniquement).

PS: Pour lancer le makefile et donc compiler tout notre projet, il suffit de se positionner sur le dossier racine du projet et de taper la commande: make compile.

Toujours dans le dossier racine du projet:

1. **Pour lancer le serveur, taper: make run_server n°PORT**
2. **Pour lancer le client, taper: make run_client n°PORT**

ANNEXE

Ci-dessous, un bref listing de nos fonctions réparties selon les composants de notre projet dans la phase de Preparation.

PARTIE CLIENT

- **fonction connexion (server, port): //déjà prête**
- **fonction Interaction ():void fonction qui fait tout. Elle appelle toutes les autres fonctions**
- **fonction char operation_1 (D,liste): gère la première opération.**
- **fonction char operation_2(D,liste): gère la deuxième opération.**
- **fonction char operation_3 (D,liste): gère la troisième opération.**
- **send()**
- **affichage_Des_Résultats: reçoit la réponse du serveur probablement utiliser decomposer.**
- **fonction interaction(): int propose différentes sorties (revenir au module 2, main menu etc.)**

PARTIE SERVEUR

- **fonction connexion (server, port):**
- **fonction load_Text(filePath,struct liste):** //tab3D charge le texte dans un tableau à 3 dimensions.
- **fonction decomposer(chaine 1D, chainedecoupé2D, coupeur)**
- **fonction traiter_requete(requete, liste): char** //traite le tableau en 2D avec des switch etc.
- **send()** //