RAPPORT PROJET CLIENT-SERVEUR

Projet Système et réseaux

Jimmy CHEVALLIER - Mickael KOC - Jules NOEL - Gabriel NOURRIT

Table des matières

Remerciements **2**

I-Présentation du projet **2**

II-Organisation **2**

II-1) Préambule 2

II-2) Début de la réalisation 2

III-Structuration **3**

III-1) Conventions de nommages 3

III-2) Structure du projet 4

III-3) Arborescence et compilation 5

III-4) Le fonctionnement du serveur 6

III-5)Le fonctionnement du client 8

III-6)Les services 8

III-7) Compilation et exécution 9

V-Evolution **10**

VI-Conclusion **10**

VI-1) Conclusion du point de vue personnelle 10

VI-2) Conclusion du point de vue technique 10

Remerciements

Tout d'abord, nous souhaitons remercier Mme Laurence Pierre et les autres professeurs de système et réseau de nous avoir apporté les connaissances nécessaires à la réalisation de ce projet.

I-Présentation du projet

Durant cette semaine de projet du 4 Décembre au 8 Décembre 2017 inclus, nous avons dû réaliser un projet mettant en oeuvre un système de consultations d'horaires de trains en utilisant le langage C et les protocoles de réseaux vu en cours. Ce système devait être réalisé avec l'architecture client-serveur, où le serveur permettait à un ou plusieurs clients de se connecter à lui en leur donnant la possibilité de consulter les différents voyages disponible. Le client a donc trois possibilitées :

- Faire une demande avec une ville de départ, une ville d'arrivée et un horaire de départ. Le serveur lui renverra un train avec l'horaire de départ le plus proche.
- Faire une demande par tranche horaire, il rentre donc une ville de départ, une ville d'arrivée, et pour la tranche, un horaire minimum et un horaire maximum. Le serveur lui renverra une liste de train dans cette tranche horaire.
- Faire une demande pour une liste de train entre deux villes. Le client rentrera donc une ville de départ et une ville d'arrivée, et le serveur lui renverra une liste de train entre ces deux villes.

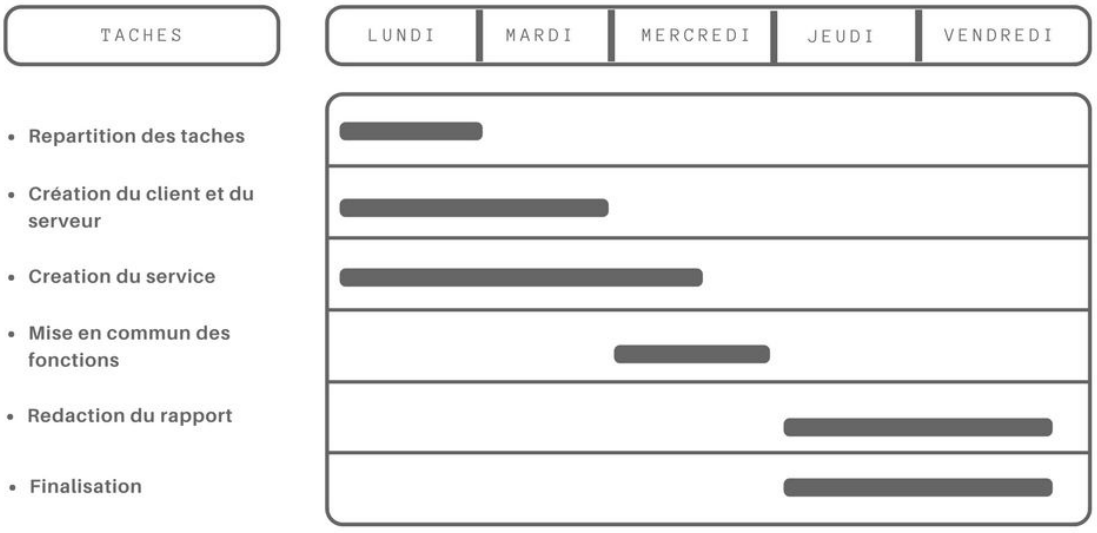
II-Organisation

II-1) Préambule

Tout d'abord, avant de commencer à coder, nous avons réalisé un TP de mise en oeuvre d'un mini client-serveur via le système de socket pour comprendre correctement le fonctionnement. Nous avons donc réussi à connecter une machine A cliente à une machine B serveur sur le même réseau, et à transmettre un message du client au serveur.

II-2) Début de la réalisation

Voici comment s'est déroulé notre projet :



Le premier jour, nous avons réalisé la liste des cas d'utilisation de l'application et nous nous sommes répartis les tâches convenablement. Nous avons donc commencé à développer le client et le serveur final et nous avons également commencé à développer le service permettant de traiter les requêtes des clients. Nous avons ensuite Mis en commun notre travail et corriger certaines erreurs le mercredi. Finalement, nous avons finalisé le projet en exécutant des séries de tests de l'application que nous avons développé.

III-Structuration

III-1) Conventions de nommages

Avant de commencer le code nous nous sommes mis d'accord sur une convention de nommage, d'une part parce que nous allions chacun coder de notre côté l'application et que notre code devait être compréhensible par tous les développeurs du groupe lors de la mise en commun. D'autre part parce que celle-ci permettent une meilleur compréhension du code, et une meilleur organisation de celui, ce qui va nous permettre de générer une documentation de notre application.

1. Pour les variables temporaire, local à l'application, on préfixe la variable par "tmp".
2. Pour les variables en paramètre d'une fonction, on la préfixé par "p".

3. Pour le nommages des fonctions/variable, on utilise **seulement** du francais et un nom qui décrit exactement l'objectif de la fonction en utilisant un verbe et un complément: exemple : remplir_tableau_voyage.
4. Chaques fonction est commenté pour qu'on sache quel traitement elle réalise, quels paramètres elle utilise et quel est la valeur exact de retour de la fonction

III-2) Structure du projet

Afin de répondre à l'objectif du projet nous avons décidé de mettre en place une certaine structure qui permet de traiter correctement les requêtes du client.
Nous avons donc décidé de créer plusieurs structures.

```
// Structure du voyage
typedef struct{
    int numero_train;
    char* ville_depart;
    char* ville_arrive;
    horaire heure_depart;
    horaire heure_arrive;
    double prix;
    CodePromo code_promo;
}voyage;
```

Cette structure permettra de stocker toutes les informations d'un train, c'est à dire : son numéro, la ville de départ, la ville d'arrivée, l'horaire de départ, l'horaire d'arrivée, le prix, et le code promo.

```
// Structure de l'heure de voyage
typedef struct {
    int heure;
    int minute;
}horaire;
```

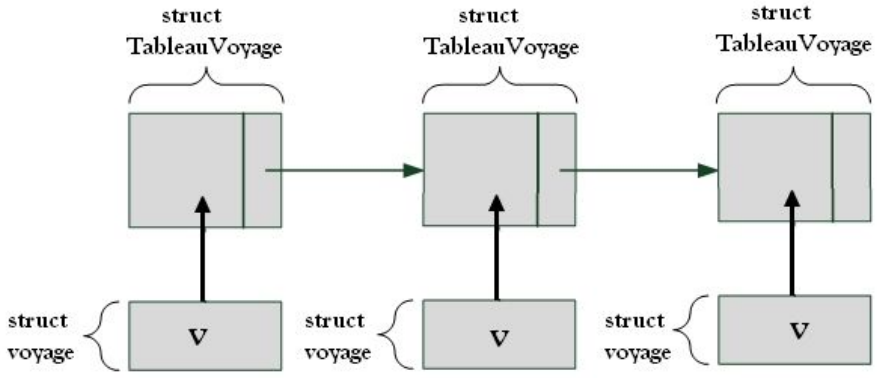
Cette structure nous permet de stocker un horaire. Elle contient les minutes et les heures de l'horaire.

```
// Structure pour les codes promos
typedef enum CodePromo CodePromo;
enum CodePromo {
    REDUC,SUPPL,DEFAULT
};
```

Cette enum nous permet de définir les codes promos qui sont REDUC (réduction de 20% sur le prix), SUPPL (supplément de 10% sur le prix), DEFAULT (le prix ne change pas).

Ensuite nous avons décidé de mettre en place une liste chaînée de voyage, qui permettra de stocker différents voyages ensemble. Et ainsi de récupérer le/les voyage(s) qui nous intéresse(nt) lors du traitement d'une requête client.

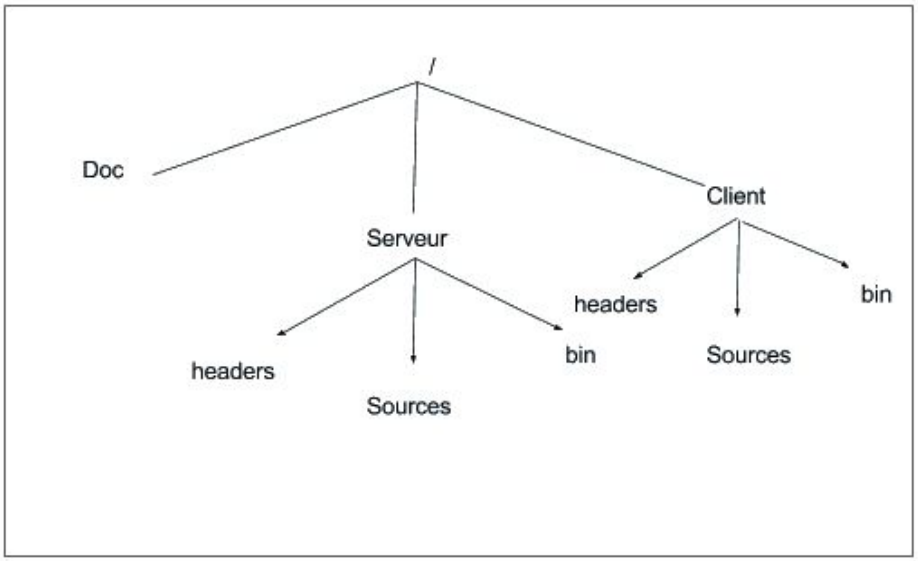
```
// Liste chainée contenant la liste des voyages
struct TableauVoyage{
    voyage v;
    struct TableauVoyage* suivant;
};
typedef struct TableauVoyage TableauVoyage;
```



Structure de la liste des voyages

III-3) Arborescence et compilation

Voici l'arborescence que nous avons utilisé pour notre projet :



Fichiers présent dans chaques dossiers :

- Racine du dossier
    - /makefile : Makefile general compilant le serveur et le client
    - /lanceur.sh : script bash appelé par makefile, il permet à l'utilisateur de rentre le nom d'hôte et le port, et lance le programme client et serveur

- Dossier serveur
    - /Serveur/bin : dossier des fichiers compilés (.o et exécutable)
    - /Serveur/header : dossier contenant les headers (.h)
    - /Serveur/Sources/ : dossier contenant les codes sources des fichiers du serveur (.c) et le fichier Trains.txt
        - fonctionRecherche.c : fichier contenant les fonctions de recherche de trains
        - main.c : fichier contenant le main du serveur
        - outil.c : fichier contenant les fonctions outils supplémentaires
        - serveur.c : fichier contenant les fonctions du serveurs
        - triModule.c : fichier contenant toutes les fonctions de tris de trains
        - voyage.c : fichier contenant toutes les fonctions réalisant un traitement sur la liste des voyages
        - Trains.txt : fichier texte contenant tous les trains
        - makefile : makefile du serveur

- Dossier client
    - /Client/bin : dossier contenant les exécutables
    - /Client/headers : dossier contenant les headers du client
    - /Client/source : dossier contenant les codes sources des fichiers du client
        - client.c : fichier contenant le client et toutes les fonctions nécessaires à la communication réseau
        - fonctionClientRequete.c : fonction contenant toutes les fonctions sur les requêtes du client
        - makefile : makefile du client

- Dossier Doc
    - /Doc/rapport.pdf : rapport de la semaine au format pdf

III-4) Le fonctionnement du serveur

Le serveur est la grosse partie de l'application puisque c'est lui qui utilise les services et retransmet les données au client. Celui ci possède donc une fonction de lancement prenant en paramètre un PORT, et un tableau de voyage contenant la liste des trains. Cette liste des trains est récupéré grâce à la méthode remplir_tableau_voyage() qui lit le fichier texte de la liste des trains et le stocke en mémoire. Nous avons choisi de stocker le tableau en mémoire car cela permet d'éviter à l'application de relire le fichier à chaque requête. L'inconvénient est que lorsque beaucoup de données de trains sont entrés, l'application va accorder une grosse partie de la mémoire au stockage du tableau, mais cette application est une retranscription minimaliste d'un système de réservation, c'est pourquoi les soucis d'optimisation mémoire ne se pose pas sur ce problème.
Le serveur va donc créé une socket qui va lui permettre d'écouter, pour savoir si un client essaye de se connecter. En cas de connection au serveur, on crée un service grâce à la fonction fork() qui va nous permettre de créer un processus fils. Ce procesus va traiter la/les requête(s) du client, tout en continuant à écouter les demandes des autres clients. C'est pourquoi, tant que l'on reçoit des requêtes de connexions, on crée des services.
Le traitement d'une requête client se fait en deux partie. Tout d'abord, une requête est une chaîne de caractère qui retranscrit une liste d'instruction séparé par le caractère "?". On utilise une fonction général "diviseur_chaine" qui nous permet de transformer une chaine de caractère en tableau via un délimiteur donné. Cette fonction est utilisé également pour transformer les lignes du fichier texte contenant la liste des trains en structure. La requête du client est donc retranscrite sous forme de tableau. Le premier élément permet d'identifier la demande du client, cette demande est le nom d'une fonction appartenant au service. Par exemple la requête :

trainParHoraireDepart**?**Valence**?**Grenoble**?**6:15**?**7:31

Va exécuter la fonction trainParHoraireDepart avec les paramètre Valence, Grenoble 6:15 et 7:31.

Par la suite, le résultat est renvoyé au client via la méthode envoie_requete_resultat(). Le résultat du service renvoie un tableau qui est ensuite retranscrit sous forme de chaîne de caractère grâce à la méthode transformer_tableau_en_reponse().

La réponse au client est séparé en 3 parties. Elle est séparée en deux parties par un **!**, la première partie est le résultat de la requête, la deuxième partie est le nombre de résultat trouvé. Ensuite sépare chaque ligne du résultat par un **@.** Enfin les informations de chaque ligne est découpée par un **?**.
La réponse envoyé au client est de la forme :
Train numéro 17524?Ville Départ : Grenoble?Ville Arrivée : Valence?Heure Départ : 16:30?Heure Arrivée : 17:45?Prix : 17.60€?@Train numéro 17525?Ville Départ : Grenoble?Ville Arrivée : Valence?Heure Départ : 16:55?Heure Arrivée : 17:55?Prix : 19.36€?@!2!

L’envoie est donc écrit dans un write et récupéré par le client qui traite la réponse et l’affiche. Voici donc le principe général du serveur.

De plus, nous avons organisé la structure du serveur en plusieurs fichiers. Nous avons choisi de créer un fichier voyage.c exclusivement pour le service CRUD (create/read/update/delete) des voyages. Ce fichier permet donc de gérer indépendamment n'importe quel voyage facilement.

D’autre part, nous avons également décidé de séparer les fonctions générales de traitements comme la fonction diviseur_chaine utilisé précédemment et d’autres fonctions d’outils dans un fichier outils.c.

III-5)Le fonctionnement du client

Le client s'exécute en prenant en paramètre un hôte (adresse IP de la machine du serveur, ou son nom d’hôte) et un port d’écoute sur lequel transitent nos paquets. La fonction principale du client est donc basée sur un simple système d’envoie de requête grâce à la méthode requete_client().
Cette méthode crée une socket pour nous permettre d’écrire un message au serveur et envoie une demande de connection à celui-ci qui va accepter notre connexion. Pour pouvoir correctement communiquer avec le serveur, nous devons former une requête à partir de la demande formé par le client. On utilise la fonction creer_requete (eventuellement creer_requete_filtre pour utiliser les filtres) qui va demander au client des informations et va automatiquement former la requête sous le même format de type :

trainParHoraireDepart**?**Valence**?**Grenoble**?**6:15**?**7:31

Cette requête va être transmise au serveur grâce à la méthode envoie_requete_serveur(). Cette méthode transmet d’une part une pré-requête au serveur en lui fournissant le nombre d’octet qu’elle va lui envoyer dans la requêtes suivante. Le serveur va donc se préparer en mémoire à recevoir une requête de X bits et va réserver la place nécessaire dans sa mémoire correspondant.
Après l’envoie le client s’attend à recevoir la réponse du serveur. Il va donc read sur un socket d’écoute jusqu'à ce qu’il va recevoir des données. Après avoir récupéré la réponse le client va la parser. Il va commencer par la parser avec les !, il va donc récupéré le nombre de résultats. Ensuite il va reparser la première partie de la réponse avec des @. Enfin il va parcourir chaque ligne grâce à la taille envoyée par le serveur, en parsant avec les ?, puis il affiche les informations sur le terminal.

III-6)Les services

Les services sont les fonctionnalités côtés serveur qui permettent de traiter les requêtes des clients. Les services utilisés sont :

1. rechercherTrainParHoraireDepart() : La fonction recherche un voyage parmis les voyages contenu dans le tableau

2. rechercherTrainsParDestionation() : La fonction recherche tout voyage d'un point a un autre parmis les voyages contenu dans le tableau

3. rechercherTrainsParTrancheHoraires(): La fonction recherche tout voyage d'un point à un autre parmis les voyages contenu dans le tableau.

En reprenant l'exemple de la requête vu précédemment :

"trainParHoraireDepart**?**Valence**?**Grenoble**?**6:15**?**7:31"

La fonction exécutée va renvoyer un tableau en fonction de la ville de départ et d'arrivé et des horaires renseignés.

III-7) Compilation et exécution

Afin de gérer la compilation du projet nous avons décidé de mettre en place trois makefile. Le premier makefile est un makefile général, appelant les deux autres makefile étant respectivement dans les dossiers /serveur/sources et /client/sources. Ces deux makefile permettent la compilation des applications du serveur et du client.
Pour faciliter l'exécution nous avons créé un script bash qui demande à l'utilisateur s'il souhaite exécuter le client, le serveur, ou les deux. Ensuite il demande à l'utilisateur de choisir un numéro de port et le nom du serveur. Ensuite ce script va appeler les deux makefiles client et serveur avec les paramètres donnés par l'utilisateur. Ainsi ce script va ouvrir un terminal avec le serveur et un avec le client. Le programme est maintenant prêt à l'emploi.

Lors de l'exécution sur le client nous demandons au client quelle type de requête il veut faire (recherche d'un train avec une horaire, recherche par tranche horaire, recherche par villes), ensuite en fonction de la réponse du client, nous lui demandons :

- requête par horaire : la ville de départ, la ville d'arrivée, l'horaire de départ
- requête par tranche horaire : la ville de départ, la ville d'arrivée, l'horaire minimum de la tranche, l'horaire maximum de la tranche
- requête par villes : la ville de départ, la ville d'arrivée

Ensuite en fonction du résultat nous demandons au client si il veut filtrer les requêtes par prix ou par durée. Le client reçoit ensuite le résultat. Enfin on demande au client s'il veut continuer ou non à consulter les trains.

Pour la saisie des informations par le client nous avons essayé de gérer tous les cas. Pour la ville de départ et d'arrivée il est seulement possible de mettre des lettres et des tirets (les tirets doivent être placés entre des lettres), et une ville de moins de 30 caractères. Pour les horaires il est seulement possible d'écrire une horaire au format hh:mm. Gérer les erreurs de saisies du côté client, permet d'éviter de potentielles erreurs dans le traitement des requêtes par le serveur.

V-Conclusion

V-1) Conclusion du point de vue personnel

Ce projet, avait un objectif simple et plutôt clair. En soit, nous n'avons pas vraiment rencontré de difficultés à le comprendre. Cependant, nous en avons quand même rencontrés certaines.
La mise en place de solutions s'est faite de manière rigoureuse. Le fer de lance de notre processus fût la documentation. De tous types, du man Linux à des cours disponibles sur internet en passant par des forums. Il nous fallait comprendre de manière quasi-exhaustive afin d'être efficace et de ne pas reproduire les mêmes erreurs. Notre partage de connaissances et de réflexion ainsi que la recherche de différents points de vus de personnes d'autres groupes fût également une force. Notre volonté et notre capacité à nous relayer le projet, ont été un axe majeur de notre réussite.

Du point de vue personnelle ce projet a été pour nous une très bonne expérience dans la gestion de projet. En effet, il nous a fallu réaliser des méthodes de communications et de partage du code afin de pouvoir être le plus productif possible. Ainsi grâce à ce projet nous avons pu apprendre la gestion d'un git, mais aussi la coopération. Grâce à la définition des tâches de chacun à l'avance, la prise de note de ce que l'on avait fait et ce qu'il nous restait à faire nous a permis une bonne gestion du projet.

V-2) Conclusion du point de vue technique

Du point de vue technique nous avons pu utiliser toutes les connaissances que nous avions acquis ce semestre dans les matières de réseau et de système. Nous avons également pu améliorer grandement notre maîtrise du langage C, ainsi que de l'utilisation des sockets.
Même si ce projet n'a pas toujours été facile, notre documentation nous a permis de réaliser le projet dans le temps, tout en prenant soin du code et de sa structure.
Enfin pour conclure nous pouvons dire que ce projet est abouti, en effet nous avons bien une architecture client serveur, ainsi qu'une communication entre ces deux entités. Du côté du client nous pouvons, comme énoncé dans l'objectif de départ consulter les trains, avec des fonctions de recherches et de filtre.