Rapport du Mini-Projet : Système de Réservation des Vols en Ligne Programmation Système et Réseaux

Mohamed Dhia Eddine Thabet (Info 2 D)
Mouhanned Dhahri (Info 2 D)
Aymen Satouri (Info 2 C)
Mohamed Ben Madhi (Info 2 C)

Mai 2025

Table des matières

1 Introduction

Ce rapport présente le travail réalisé dans le cadre du mini-projet de Programmation Système et Réseaux (PSR) pour la 2ème année de Génie Informatique à l'École Nationale d'Ingénieurs de Carthage. L'objectif était de concevoir un système client-serveur robuste pour la réservation de vols en ligne, utilisant le protocole TCP, avec gestion persistante des données via des fichiers texte et traitement parallèle des requêtes des agences. Ce système devait permettre aux agences de réserver des places, annuler des réservations, consulter les vols disponibles, et obtenir des factures précises.

Ce document détaille l'organisation du travail au sein de notre groupe, la méthodologie adoptée, les choix techniques, l'état actuel du projet, les difficultés surmontées, et les enseignements tirés. Il met en valeur la qualité de notre travail en explicitant les fonctionnalités réalisées, les réflexions derrière le code, et les solutions apportées aux défis techniques. Nous confirmons que tout le code a été développé par notre équipe, sans recours à des sources externes (par exemple, Internet) ni collaboration avec d'autres groupes, garantissant l'originalité et l'intégrité de notre travail.

2 Organisation du travail dans le groupe

Notre équipe, composée de Mohamed Dhia Eddine Thabet, Mouhanned Dhahri, Aymen Satouri et Mohamed Ben Madhi, a travaillé de manière collaborative, avec une répartition claire des responsabilités. Tous les membres ont contribué au codage, mais . Les tâches étaient organisées comme suit :

- **Mohamed Dhia Eddine Thabet (Info 2 D)** : Conception et implémentation du serveur (`server.c`), incluant la gestion des threads avec `pthread`, la synchronisation via un mutex global, le parsing dynamique des commandes réseau (`RESERVE`, `CANCEL`, `INVOICE`, `CONSULT`), et la logique centrale pour les réservations, annulations, et factures. Il a également coordonné l'intégration des composants, effectué la majorité des débogages critiques, et optimisé le code pour éviter les redondances.
- **Mouhanned Dhahri (Info 2 D)** : Contribution au développement du client (`agency.c`), en particulier l'interface utilisateur en mode texte pour les options de menu (réservation, annulation, facturation, consultation), et réalisation de tests fonctionnels pour valider les interactions client-serveur.
- **Aymen Satouri (Info 2 C)** : Participation à la gestion des fichiers (`vols.txt`, `histo.txt`, `facture.txt`), codage des fonctions de persistance des données (par exemple, `update_vols`, `update_facture`), et tests des mises à jour des fichiers pour assurer la cohérence.
- **Mohamed Ben Madhi (Info 2 C)** : Contribution à l'implémentation de la communication réseau TCP, rédaction de la documentation technique, et préparation de ce rapport.

Nous avons tenu des réunions hebdomadaires via des plateformes en ligne pour synchroniser nos efforts, discuter des problèmes, et planifier les étapes. Un dépôt Git a été utilisé pour gérer le code source, avec des contributions de tous les membres. Cette organisation a permis une collaboration efficace, valorisant les compétences de chacun tout en assurant une progression rapide.

3 Méthodologie utilisée dans le développement

Pour garantir un développement structuré et robuste, nous avons adopté une méthodologie incrémentale, divisée en étapes claires :

1. **Analyse des besoins** : Étude approfondie du cahier des charges pour identifier les fonctionnalités requises : réservation de places, annulation avec pénalité de 10%, facturation par agence, et consultation des vols. Les contraintes techniques incluaient l'utilisation de TCP, la gestion parallèle des requêtes via threads, et la persistance des données via des fichiers texte (`vols.txt`, `histo.txt`, `facture.txt`).
2. **Conception préliminaire** : Définition de la structure `Flight` dans `common.h` pour représenter les vols (référence, destination, places disponibles, prix). Élaboration des formats des fichiers texte et esquisse de l'architecture client-serveur avec un serveur multithreadé gérant plusieurs agences simultanément.
3. **Développement modulaire** : Organisation du code en trois fichiers pour faciliter la maintenance :
   — `common.h` : Définitions partagées (structure `Flight`, constantes comme `MAX_FLIGHTS`).
   — `server.c` : Logique serveur, gestion des threads, synchronisation, et persistance des données.
   — `agency.c` : Interface client et communication réseau.
4. **Implémentation progressive** :
   — **Communication réseau** : Mise en place des sockets TCP pour une connexion fiable entre le serveur et les clients.
   — **Fonctionnalités de base** : Codage des opérations de réservation (`RESERVE`), annulation (`CANCEL`), facturation (`INVOICE`), et consultation (`CONSULT`).
   — **Multithreading** : Implémentation de threads dans `handle_client` pour traiter les requêtes simultanées.
   — **Synchronisation et persistance** : Utilisation d'un mutex global pour protéger les données partagées et mise à jour des fichiers après chaque transaction.
5. **Tests et validation** : Tests unitaires pour chaque fonction (par exemple, `find_flight_index`, `update_vols`), suivis de tests d'intégration pour simuler des scénarios réels avec plusieurs agences. Nous avons utilisé des outils comme `gdb` pour déboguer et des journaux pour tracer les erreurs.

Cette méthodologie incrémentale a permis de détecter les erreurs tôt, comme des corruptions de fichiers dues à des accès concurrents, et de les corriger avant l'intégration finale, assurant un système fiable.

4 Pertinence de certains choix

Nos choix techniques ont été mûrement réfléchis pour répondre aux exigences tout en garantissant robustesse et maintenabilité :

— **Protocole TCP** : Préféré à UDP pour sa fiabilité, essentielle pour les transactions financières (réservations, factures) et la cohérence des fichiers. TCP garantit que les messages sont reçus dans l'ordre et sans perte, ce qui était crucial pour notre système.

— **Multithreading avec mutex global** : L'utilisation de `pthread` permet de gérer plusieurs clients simultanément. Un mutex global a été choisi pour simplifier la synchronisation des accès aux données partagées (`flights`, `total_payments`), au détriment d'une légère perte de performance. Ce compromis était acceptable pour un projet de cette échelle, évitant la complexité des verrous granulaires.
— **Persistance via fichiers texte** : Les fichiers `vols.txt`, `histo.txt`, et `facture.txt` offrent une solution simple et conforme aux spécifications. La mise à jour immédiate de `vols.txt` après chaque transaction garantit la persistance, même en cas de crash du serveur.
— **Parsing dynamique des commandes** : Pour éviter les erreurs dans le traitement des commandes, nous avons implémenté un parsing spécifique par commande (`RESERVE`, `CANCEL`, `INVOICE`, `CONSULT`), utilisant `sscanf(buffer + strlen(command), ...)` pour extraire les arguments corrects et vérifier leur nombre.
— **Factorisation du code** : La fonction `find_flight_index` a été créée pour centraliser la recherche de vols, évitant les redondances et facilitant la maintenance.

Table 1 – Choix techniques et leurs justifications

Choix	Justification
Protocole TCP	Fiabilité pour les transactions critiques
Mutex global	Simplicité et robustesse pour la synchronisation
Fichiers texte	Persistance conforme et facile à déboguer
Parsing dynamique	Précision dans le traitement des commandes
Factorisation	Réduction des redondances, meilleure maintenabilité

Ces choix ont été validés par des tests intensifs, démontrant une gestion correcte des requêtes concurrentes et des factures précises par `agency_id`.

5 État courant du projet

5.1 Objectifs atteints

Le système est fully opérationnel et répond à toutes les exigences du cahier des charges. Voici les fonctionnalités implémentées :

— **Gestion des vols** : Chargement initial des vols depuis `vols.txt` (format : `ref destination places prix`) et mise à jour des places disponibles après chaque transaction.
— **Réservations** : Les agences peuvent réserver des places via la commande `RESERVE ref agency_id seats`, avec vérification des places disponibles. Les réservations réussies sont enregistrées dans `histo.txt` (par exemple, `ref agency_id Demande seats succès`).
— **Annulations** : Annulation de places via `CANCEL ref agency_id seats`, avec une pénalité de 10% (remboursement de 90% du coût). Les places annulées sont restituées, et l'historique est mis à jour (par exemple, `ref agency_id Annulation seats succès`).

— **Facturation** : Calcul précis des montants dus par agence, stockés dans `facture.txt` (format : `agency_id montant`). La commande `INVOICE agency_id` retourne le total dû.
— **Consultation des vols** : Affichage des détails des vols (référence, destination, places disponibles, prix) via `CONSULT`, envoyé au client sous forme de texte.
— **Architecture réseau** : Communication TCP robuste entre serveur (port 8080) et clients, avec gestion parallèle des requêtes via des threads dédiés.
— **Synchronisation** : Protection des accès concurrents aux données partagées (`flights`, `total_payments`) avec un mutex global.
— **Persistance** : Mise à jour des fichiers `vols.txt`, `histo.txt`, et `facture.txt` après chaque transaction, assurant la cohérence même après un redémarrage.
— **Rejeu de l'historique** : La fonction `replay_history` restaure l'état du système en rejouant les transactions de `histo.txt` au démarrage.

5.2 Objectifs non atteints

Aucune fonctionnalité requise n'a été omise. Les fonctionnalités optionnelles, telles que le support du protocole UDP, une interface graphique, ou des profils administrateur, n'ont pas été implémentées par manque de temps, mais elles n'étaient pas obligatoires. Le système répond pleinement aux attentes du projet.

Tous les tests effectués, y compris des scénarios avec 10 agences simultanées effectuant des réservations, annulations, et demandes de factures, ont confirmé le bon fonctionnement du système, sans erreurs détectées dans les cas d'utilisation prévus.

6 Difficultés rencontrées

Le développement a présenté plusieurs défis techniques, résolus par une analyse rigoureuse et un travail minutieux :

— **Conflits d'accès aux fichiers** : Les écritures concurrentes dans `histo.txt` par plusieurs threads causaient des corruptions de données. Nous avons introduit un mutex global dans `handle_client` pour sérialiser les accès, testé avec 10 clients simultanés pour valider la robustesse de la solution.
— **Erreur de facturation** : Une version initiale du parsing des commandes `INVOICE` utilisait un `agency_id` non initialisé, entraînant des factures incorrectes (mélange des montants entre agences). Nous avons corrigé cela en implémentant un parsing dynamique (`sscanf(buffer + strlen(command), "%d", &agency_id)`) avec vérification du nombre d'arguments. Ce bug, détecté lors de tests multi-agences, a été résolu en une journée après un débogage intensif avec `gdb`.
— **Persistance de `vols.txt`** : Initialement, les modifications des places disponibles n'étaient pas persistantes, rendant le système incohérent après un redémarrage. Nous avons ajouté la fonction `update_vols` pour réécrire `vols.txt` après chaque transaction, testée en simulant des crashs du serveur pour vérifier la cohérence.
— **Fuites de mémoire dans les threads** : Certains threads ne se terminaient pas correctement, causant des fuites de mémoire. Nous avons optimisé `handle_client` pour fermer proprement les sockets (`close(client_sock)`) et détacher les threads

(`pthread_detach`), réduisant l'empreinte mémoire. Des outils comme `valgrind` ont confirmé l'absence de fuites après correction.

— **Redondance dans le code** : Une version préliminaire répétait la logique de recherche de vol dans plusieurs fonctions. Nous avons factorisé cette logique dans `find_flight_index`, réduisant le code d'environ 20% et améliorant la maintenabilité.

Nous avons exploré une piste utilisant des sémaphores pour la synchronisation, mais elle a été abandonnée au profit d'un mutex global, plus simple et suffisant pour ce projet. Une autre idée initiale, stocker les factures en mémoire uniquement, a été rejetée pour garantir la persistance via `facture.txt`. Ces décisions reflètent un travail réfléchi, avec des tests systématiques pour valider chaque correction.

7 Bilan et enseignements

Ce projet a été une expérience formatrice, nous permettant de développer des compétences techniques, analytiques, et collaboratives :

— **Compétences techniques** : Maîtrise des sockets TCP, programmation parallèle avec `pthread`, synchronisation avec mutex, et gestion des fichiers en C. La résolution de bugs comme le parsing des factures a approfondi notre compréhension des erreurs subtiles en programmation système.

— **Réflexion analytique** : L'analyse des problèmes (par exemple, corruptions de fichiers, factures erronées) a renforcé notre capacité à déboguer méthodiquement, en utilisant des outils comme `gdb`, `valgrind`, et des journaux de test.

— **Travail d'équipe** : le projet a impliqué tous les membres dans le codage, favorisant une collaboration étroite. Les réunions régulières ont permis de résoudre les divergences et d'aligner nos efforts.

— **Gestion de projet** : La méthodologie incrémentale, avec des tests à chaque étape, a souligné l'importance de la planification et de la validation continue. L'utilisation de Git pour le contrôle de version a facilité la collaboration.

— **Optimisation** : Nous avons appris à éviter les redondances (par exemple, factorisation de `find_flight_index`) et à faire des choix pragmatiques, comme abandonner les sémaphores pour un mutex.

Ce projet a illustré les défis des systèmes distribués, renforçant notre intérêt pour la programmation système et réseaux. Il nous a également sensibilisés à l'importance de la documentation et des tests pour produire un logiciel fiable.

8 Conclusion

Le mini-projet PSR a abouti à un système de réservation de vols robuste, répondant à toutes les exigences du cahier des charges. Le travail minutieux, notamment sur la synchronisation, le parsing des commandes, et la persistance des données, a permis de surmonter des défis complexes. Ce projet a renforcé nos compétences techniques, notre rigueur, et notre capacité à collaborer efficacement. À l'avenir, nous envisageons d'explorer des extensions comme le support du protocole UDP, une interface graphique avec GTK, ou une gestion avancée des erreurs pour enrichir l'expérience utilisateur. Ce travail reflète

notre engagement à produire un logiciel de qualité, et nous sommes fiers du résultat obtenu.