L3 Informatique

Rapport

Système et Réseaux – Pegasus

Piton Léo, Semenov Egor

Novembre–Décembre 2025

Table des matières

Étudiant 1 : Egor Semenov-Tyan-Shanskiy, Egor_Semenov-Tyan-Shanskiy@etu.ube.fr
Étudiant 2 : Léo Piton, Leo_Piton@etu.ube.fr
GitHub : https://github.com/HobbitTheCat/pega_s6

1 Introduction

Ce rapport est une synthèse de notre projet et est présent pour répondre aux attentes du projet universitaire qui nous a été demandé. Cela étant dit , certaines précisions techniques ont été écrits dans un second rapport, le rapport_technique qui a été ajouté également lors du dépôt de celui si. Cela nous permet de ne pas trop détaillé d. Un lien Github sera fournit sur ce compte rendu pour la partie code du projet mais aussi les 3 rapports : le pré_rapport , le rapport et le rapport_technique.

2 Analyse du Problème et du Sujet

2.1 Contexte et Objectifs

Le projet s'inscrit dans le cadre de l'unité d'enseignement "Systèmes & Réseaux". La commande initiale consistait à illustrer les concepts théoriques vus en cours à travers le développement complet du jeu de cartes "6 qui prend".

L'objectif est concevoir une architecture système robuste capable de :

— Gérer des interactions hybrides entre joueurs humains (via terminal) et joueurs robots.
— Coordonner plusieurs processus distincts (Gestionnaire, Joueurs, Stats).
— Respecter les règles de simultanéité du jeu, où le choix des cartes est secret et révélé en même temps.

Au-delà de la simple implémentation des règles, le défi majeur imposé par le sujet était le développement de fonctionnalités avancées, notamment le jeu en réseau et la coopération en équipe, nécessitant une maîtrise fine des mécanismes de communication.

2.2 Défis Techniques et Systèmes

L'analyse du sujet a rapidement mis en lumière plusieurs défis techniques critiques que notre développement devait surmonter :

— **La Gestion de la Concurrence :** Le serveur doit être capable de gérer de nombreux clients simultanément sans blocage. Le jeu "6 qui prend" exige que le serveur attende les réponses de tous les joueurs avant de procéder à la résolution du tour, ce qui pose un problème classique de synchronisation et de gestion des délais d'attente.
— **La Fiabilité des Communications (IPC & Réseau) :** Le système requiert un flux constant d'informations entre le gestionnaire de jeu et les participants (distribution des

mains, état des rangées, résultats). Assurer que ces messages arrivent dans l'ordre et sans perte est crucial pour l'intégrité de la partie.

— **L'Isolation des Sessions :** Comme identifié lors de notre lecture initiale, le système doit pouvoir supporter plusieurs parties (sessions) en parallèle, chacune fonctionnant de manière étanche pour ne pas interférer avec les autres.

2.3 Apport Pédagogique et Développement Personnel

Ce projet a constitué un terrain d'application concret pour transformer nos connaissances théoriques en compétences d'ingénierie système. Il nous a obligés à :

— **Structurer une architecture complexe :** Passer de petits exercices isolés à une application complète nécessite une vision globale (design patterns, modularité).

— **Maîtriser le bas niveau :** L'utilisation imposée du C et des scripts Shell/Awk nous a poussés à gérer manuellement la mémoire, les sockets et les signaux, renforçant notre compréhension du fonctionnement intime du système d'exploitation.

— **Collaborer et Documenter :** Le travail en binôme et la production de rapports intermédiaires ont simulé un environnement professionnel où la clarté du code et la communication sont aussi importantes que la solution technique elle-même.

2.4 Lien avec l'Analyse Préliminaire (Rapport Pega_s)

Dès notre rapport préliminaire rendu en novembre, nous avions identifié que la clé de la réussite résidait dans une architecture orientée événements ("Event-Driven"). Dans le document *Pega_s*, nous avions anticipé plusieurs besoins qui se sont confirmés lors du développement final :

— La nécessité de séparer le thread de gestion réseau (I/O) des threads gérant la logique des sessions pour éviter les goulots d'étranglement.

— L'importance cruciale de l'utilisation de sockets non-bloquants pour optimiser les performances et éviter les interblocages.

— Le choix de structures de communication "lock-free" (comme les buffers SPSC - Single Producer Single Consumer) pour minimiser les coûts de synchronisation entre les threads.

Cette analyse initiale a servi de boussole tout au long du projet, nous permettant de nous concentrer sur l'optimisation d'une structure saine plutôt que de devoir refondre l'architecture à mi-parcours.

3 Évolution de l'Architecture

Cette section retrace le cheminement technique qui nous a conduits de la solution la plus immédiate vers l'architecture finale optimisée.

3.1 L'Approche Naïve : Un Thread par Client

Notre première intuition, qualifiée d'approche "naïve", reposait sur un modèle classique de programmation réseau synchrone. Le principe est le suivant : pour chaque nouveau client se

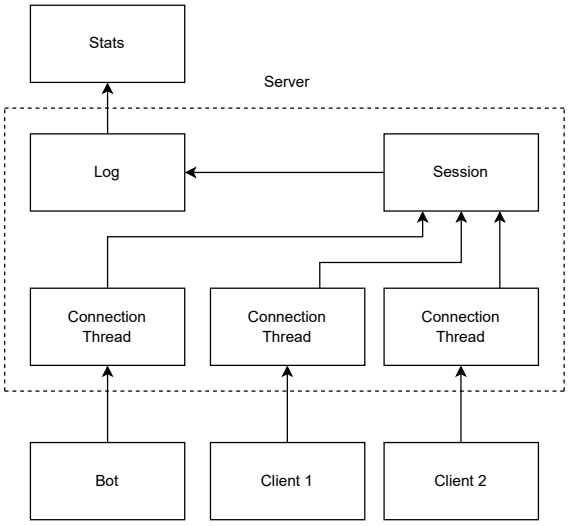

FIGURE 1 – Approche Naïve

connectant via un socket, le serveur instancie un nouveau thread dédié (modèle *Thread-for-Client*). De plus, cette architecture initiale ne prévoyait la gestion que d'une seule session de jeu active à la fois.

3.1.1 Avantages de l'approche

Bien que limitée, cette solution présente des atouts indéniables en début de projet :

— **Simplicité de compréhension et de mise en œuvre :** Le flux d'exécution est linéaire. On utilise des opérations d'entrée/sortie (I/O) bloquantes, ce qui rend le code séquentiel et facile à lire (ex : `read()` attend simplement que des données arrivent).

— **Gestion contextuelle simplifiée :** Chaque thread possédant sa propre pile d'exécution, il est aisé de conserver l'état du joueur (cartes en main, score) directement dans des variables locales au thread sans recourir à des structures de gestion d'état complexes.

3.1.2 Limites et Problèmes identifiés

Rapidement, cette architecture a montré ses limites face aux exigences de performance et de robustesse du sujet :

— **Surcharge du système (Overhead de Commutation) :** À mesure que le nombre de joueurs augmente, le nombre de threads explose. Le système d'exploitation (le *scheduler* Linux) doit constamment interrompre un thread pour en lancer un autre. Lorsque le nombre de threads devient critique, le temps passé par le processeur à effectuer ces "commutations de contexte" (*context switching*) dépasse le temps utile alloué au traitement des requêtes.

— **Risque de Déni de Service (DoS) :** La création d'un thread consomme des ressources mémoire (pile) et des descripteurs de fichiers. Une architecture où chaque connexion entraîne une allocation de ressources lourdes expose le serveur à une saturation rapide,

créant un risque de Déni de Service (DDoS) involontaire ou malveillant dès que le trafic augmente.

— **Problèmes de Concurrence et Synchronisation :** Dans un jeu comme "6 qui prend", les états sont partagés (le plateau de jeu est commun). Avec un thread par joueur, l'accès simultané aux ressources partagées nécessite la mise en place de verrous (*mutex*, sémaphores). Cela engendre :
  — Des risques de conditions de course (*race conditions*) si la synchronisation est mal gérée.
  — Des ralentissements dus à la contention (les threads passent leur temps à attendre qu'un verrou se libère).

— **Monolithisme (Session Unique) :** Cette structure lie fortement la logique réseau à la logique du jeu. Il devient très complexe d'implémenter le multi-session (plusieurs parties en parallèle) car le serveur est bloqué par l'état de la partie en cours.

Face à ces constats, notamment la difficulté de faire cohabiter la couche réseau bloquante avec la logique de jeu, nous avons décidé de migrer vers une architecture asynchrone non-bloquante.

3.2 Concepts Structurants et Ajouts Majeurs

Afin de dépasser les limites de l'approche naïve et de garantir la pérennité de l'application jusqu'à sa version finale, nous avons intégré deux piliers fondamentaux à notre architecture : un protocole standardisé et une logique événementielle.

3.2.1 Le Protocole : Vers un Format Universel

La gestion de multiples types de messages (connexion, jeu) nécessitait de sortir du traitement ad-hoc pour aller vers une standardisation. Nous avons défini un besoin critique : l'utilisation d'un **format de données universel**.

Ce choix architectural nous permet de :

— **Uniformiser le traitement :** Peu importe la nature de la requête (action de jeu ou simple "ping"), le serveur réceptionne et désérialise les paquets de manière identique avant de les router.

— **Garantir l'extensibilité :** Ce format générique est conçu pour être facilement étendu. Il permet d'ajouter de nouvelles fonctionnalités (comme les extensions de règles ou la gestion d'équipe) sans remettre en cause la structure du moteur réseau.

Les détails spécifiques de l'encodage et de la structure des paquets sont décrits dans notre *rapport technique*.

3.2.2 L'Architecture Orientée Événements (Event-Driven Architecture)

Parallèlement au protocole, nous avons radicalement changé notre approche du flux d'exécution. Au lieu de suivre un script linéaire, nous avons adopté une **Architecture Orientée Événements (EDA)**.

Dans ce modèle :

— Les échanges entre le Client et le Serveur ne sont plus vus comme des conversations continues, mais comme une succession d'**événements isolés** et indépendants.
— Le serveur agit comme un réacteur : il attend un événement (arrivée d'un paquet, expiration d'un timer), le traite, met à jour l'état, puis retourne en attente.

Cette philosophie, couplée à notre protocole universel, constitue le cœur stable de notre projet final. Elle permet de découpler totalement la réception réseau de la logique métier, offrant ainsi la fluidité et la résistance à la charge recherchées (cf. implémentation détaillée dans le *rapport technique*).

3.3 Première Tentative d'Architecture Modulaire : Le SessionManager Centralisé

Pour pallier le problème du "tout dans le thread", nous avons introduit une première séparation des responsabilités via un composant dédié : le **SessionManager**.

3.3.1 L'Objectif : Extraire la Logique du Serveur

L'idée directrice était de purifier le code du Serveur pour le ramener à sa fonction essentielle : la gestion des connexions.

— **Le Serveur :** Son rôle est réduit à la réception des paquets bruts et à leur transfert immédiat. Il ne possède "zéro logique" de jeu.
— **Le SessionManager :** Il concentre toute l'intelligence applicative. Il est responsable de

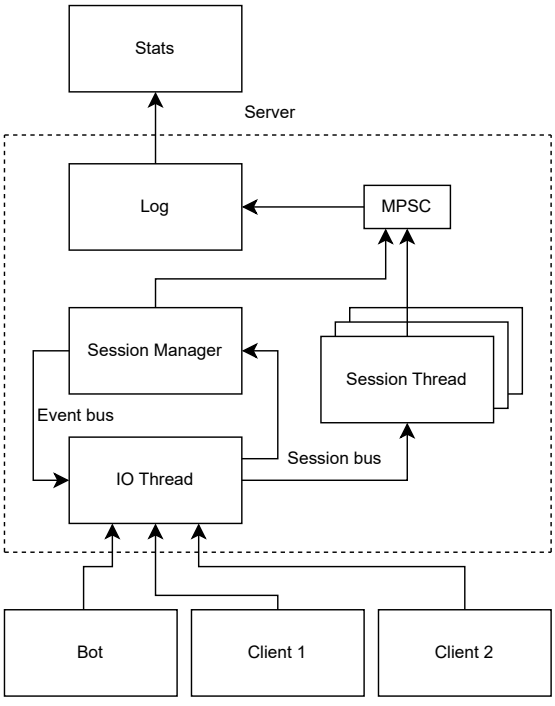

FIGURE 2 – Première Approche

la création des sessions, de l'attribution des joueurs aux parties et du maintien de l'état global.

3.3.2 Problèmes de cette Architecture Intermédiaire

Bien que plus propre conceptuellement que l'approche naïve, cette structure a révélé de nouveaux défauts majeurs lors de l'analyse :

— **Goulot d'Étranglement (Bottleneck) :** Le `SessionManager` est devenu un point de passage obligé unique. Toute communication entre le module réseau (IO) et une session de jeu spécifique devait transiter par ce manager. Cela centralise le trafic et limite les performances globales.

— **Couplage Fort et Latence :** La séparation n'était pas assez franche. Le couplage excessif entre le thread IO et le `SessionManager` a introduit une complexité inutile dans le chemin critique des données. L'ajout de cette couche intermédiaire ("middleware") monolithique a augmenté la latence de traitement à cause des coûts de transfert de données entre les modules, sans offrir le parallélisme espéré.

— **Problème d'Abstraction :** Le `SessionManager` mélangeait des tâches de trop de niveaux différents : gestion de haut niveau (créer une partie) et routage de bas niveau (passer un paquet à une partie).

Cette étape nous a appris qu'il ne suffisait pas de déplacer la logique dans un autre module, mais qu'il fallait décentraliser l'exécution des sessions elles-mêmes (d'où l'architecture finale avec des threads autonomes par session).

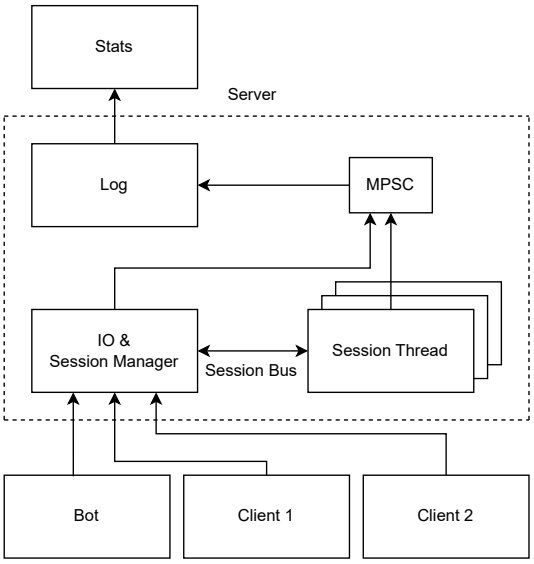

FIGURE 3 – Second Approach

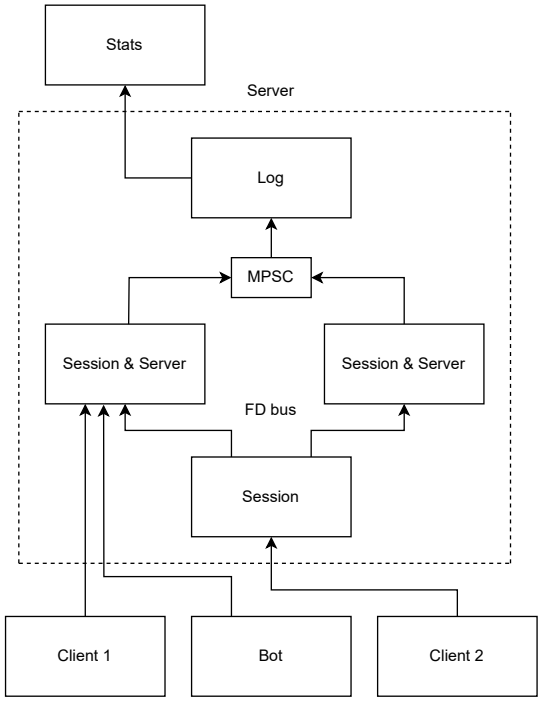

FIGURE 4 – Deuxième approche alternative

3.4 Deuxième Itération : Restructuration de l'Interface IO-Session

Forts des constats précédents, nous avons entrepris une refonte de la communication entre le module d'Entrées/Sorties (IO) et le `SessionManager`. L'objectif n'était plus seulement de séparer les modules, mais de redéfinir la manière dont ils se lient pour éliminer le "couplage fort" qui pénalisait la première approche.

Cette nouvelle structure nous a apporté plusieurs gains significatifs :

— **Réduction de la Latence :** En assouplissant le lien entre la réception réseau et la gestion de session, nous avons fluidifié le traitement des paquets.

— **Enrichissement des Structures Utilisateurs :** La suppression des contraintes de couplage nous a permis de complexifier les structures de données décrivant les utilisateurs. Nous avons pu y intégrer davantage de métadonnées sans alourdir le thread réseau.

— **Simplification de l'Enregistrement :** La procédure de "Login/Register" est devenue plus atomique et plus simple à gérer.

— **Sécurité Accrue :** En compartimentant mieux les données, nous avons mécaniquement réduit la surface d'attaque potentielle du système.

3.5 L'Approche Alternative "Distribuée" (Approche Bis)

Parallèlement, nous avons théorisé et testé une approche radicale dite "distribuée". L'idée directrice était de transférer l'intégralité de la logique de commande directement au sein des sessions, ne laissant au serveur qu'un rôle de passe-plat minimal.

3.5.1 L'objectif théorique : La "Zéro Latence"

En traitant la commande au plus près de la session, sans intermédiaire, nous visions une latence quasi nulle. Cependant, la mise en pratique s'est heurtée à la réalité complexe du système Linux.

3.5.2 Obstacles Techniques et Abandon

Cette architecture "idéale" sur le papier s'est révélée impraticable en raison de barrières système majeures :

— **Transfert de Contextes (Sockets) :** Transmettre un socket ouvert (file descriptor) du thread serveur vers un thread de session est une opération complexe sous Linux. Le changement de contexte et de propriété du socket pose des problèmes de stabilité.

— **Fragmentation des instances `epoll` :** La création de multiples instances `epoll` dans différents threads (un par session) a complexifié la boucle d'événements globale, rendant le débogage et la maintenance ardus.

— **Gestion du Cycle de Vie :** La gestion des connexions et, surtout, des déconnexions intempestives de clients au sein même des sessions s'est avérée instable.

— **Risques de Sécurité et Maintenance :** Cette approche obligeait à dupliquer une partie du code de gestion réseau dans chaque session, augmentant la surface d'attaque

et violant le principe DRY (Don't Repeat Yourself).

Ces expérimentations ont été décisives : elles nous ont confirmé que la solution ne résidait ni dans la centralisation totale, ni dans la distribution totale de la logique réseau, mais dans l'utilisation de canaux de communication asynchrones (SPSC) que nous avons finalement adoptés.

3.6 Troisième Itération : Correction de l'Abstraction et Sécurisation

L'analyse approfondie de notre deuxième approche a révélé des faiblesses conceptuelles critiques, situées à la frontière entre le réseau et l'application.

3.6.1 Identification des Failles

Deux problèmes majeurs ont motivé cette troisième évolution :

— **Violation des Couches d'Abstraction :** Nous avons constaté que l'en-tête du paquet (*Header*), qui contient des informations de transport, était transmis tel quel jusqu'à la session de jeu (Couche Application). Or, selon le principe de séparation des couches, la session ne devrait traiter que la "charge utile" (*Payload*) et ignorer les détails du transport.

— **Faille de Sécurité (Usurpation d'Identité) :** Dans l'approche précédente, le client transmettait son propre ID dans l'en-tête du paquet. Cela créait une vulnérabilité critique : un client malveillant pouvait modifier ce champ pour usurper l'identité d'un autre joueur ("Spoofing"). Pour contrer cela, le serveur devait effectuer des vérifications redondantes et coûteuses à chaque réception de paquet.

3.6.2 La Solution : Refonte des Structures et du Protocole

Pour résoudre ces problèmes, nous avons procédé à une refonte structurelle en deux points :

1. **Division de la Structure Utilisateur :** Nous avons scindé la représentation d'un utilisateur en deux entités distinctes selon leur cycle de vie :
   — **Partie "Transport" (Durée de vie du Socket) :** Contient les informations liées à la connexion active. Elle est éphémère.
   — **Partie "Logique" (Durée de vie Permanente) :** Contient les données persistantes du joueur (score, identité).
2. **Épuration du Header :** Nous avons supprimé les identifiants explicites (ID Client/-Session) de l'en-tête envoyé par le client. Désormais, l'identité est déduite par le serveur via le socket de connexion, rendant l'usurpation impossible par design. De plus, l'en-tête est désormais consommé par la couche de transport et n'est plus jamais transmis à la couche application (Session), rétablissant ainsi une abstraction stricte.

3.7 Quatrième Itération : Réflexion sur le "Session Pooling" et Choix Final

Bien que la troisième itération ait résolu les problèmes de sécurité et d'abstraction, nous avons identifié une limite théorique liée à la scalabilité du modèle "Un Thread par Session".

3.7.1 Limites de l'Approche 3 à Grande Échelle

Si le nombre de sessions devient très important, deux contraintes physiques apparaissent :

— **Saturation Mémoire :** Chaque thread de session alloue ses propres tampons de transmission. La multiplication linéaire de ces allocations risque d'épuiser la mémoire disponible.

— **Coût de Commutation (Context Switching) :** Même si les sessions sont indépendantes, le système d'exploitation doit jongler entre des milliers de threads, réintroduisant une latence système (overhead) préjudiciable.

3.7.2 La Solution Théorique : Regroupement de Sessions

Nous avons envisagé une quatrième architecture basée sur le constat suivant : la logique du jeu "6 qui prend" est algorithmiquement très légère (essentiellement des comparaisons d'entiers relatifs et des opérations sur des tableaux). L'idée était donc de **regrouper l'exécution de plusieurs sessions (N) au sein d'un unique thread**.

3.7.3 Obstacle et Décision Finale

Cependant, cette approche de "Session Pooling" induit une complexité logicielle majeure :

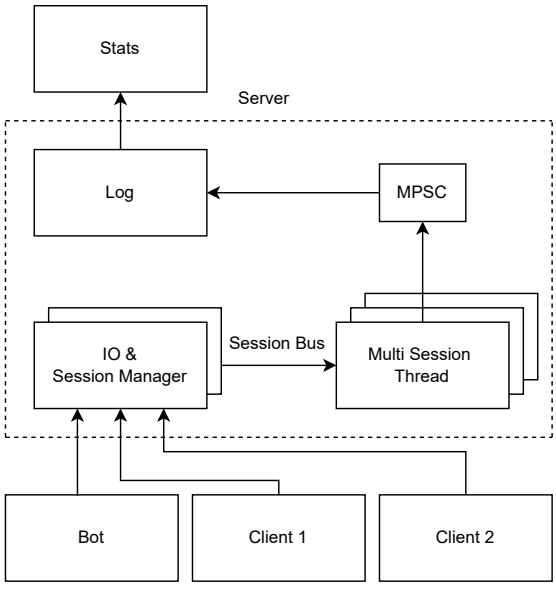

FIGURE 5 – Quatrième approche

— Il devient nécessaire d'implémenter un algorithme de routage interne complexe pour diriger le bon paquet vers la bonne instance de jeu au sein du même thread.
— La gestion des erreurs dans une partie pourrait impacter les autres parties partageant le même thread.

Face à cette complexité accrue du routage et considérant que la charge CPU par partie est minime, nous avons jugé que le gain de performance ne justifiait pas le coût de développement et de maintenance. **Nous avons donc décidé de valider et de conserver la troisième approche** pour notre architecture finale, celle-ci offrant le meilleur compromis entre sécurité, isolation des pannes et simplicité de maintenance.

3.8 Module : Server (Cœur de l'infrastructure)

Le module **Server** constitue le point d'entrée unique et le routeur central de notre application. Conçu pour la performance, il adopte une architecture entièrement **non-bloquante** et **orientée événements**. Il se distingue par son absence totale de logique de jeu ("Game Logic"), se concentrant exclusivement sur le maintien de l'infrastructure réseau et le routage des messages.

Note : Les détails d'implémentation du noyau réseau (mécanisme `epoll`, gestion des buffers circulaires SPSC, structures de connexion) sont documentés de manière exhaustive dans le ***Rapport Technique (Chapitre 3)****.*

Ses responsabilités architecturales se déclinent en trois axes majeurs :

— **Multiplexage des Entrées/Sorties :** Contrairement à une approche multi-threadée classique, le serveur utilise un unique thread principal ("Event-Loop") pour gérer simultanément l'ensemble des connexions TCP et des signaux internes. Cette centralisation permet de supporter une forte charge sans subir le coût des commutations de contexte système.
— **Gestion du Cycle de Vie et Sécurité :** Le module assure l'authentification et le suivi des clients. Il implémente une séparation stricte entre le contexte technique (le socket, qui est éphémère) et l'identité du joueur (persistante). Cette abstraction est essentielle pour permettre notre mécanisme de reconnexion sécurisée par tokens.
— **Orchestration et Routage :** Le serveur agit comme un aiguilleur ("Dispatcher"). Il analyse les paquets entrants pour séparer les requêtes administratives (traitées localement) des actions de jeu. Ces dernières sont alors injectées dans le bus de communication spécifique à la session de jeu concernée, qui s'exécute dans son propre thread isolé.

3.9 Module : Log (Système Asynchrone)

Le module **Log** est une composante critique pour la performance du système. Son rôle architectural est de déporter les coûteuses opérations d'entrées/sorties (I/O) sur disque vers un thread dédié, garantissant ainsi que ni la boucle événementielle du serveur, ni la logique temps réel des sessions ne subissent de latence (phénomène de "IO Wait").

Note : L'implémentation du tampon circulaire "Multi-Producer Single-Consumer" (MPSC) et l'utilisation des opérations atomiques pour la synchronisation sont détaillées dans le ***Rapport***

Technique (Chapitre 6).

Ses caractéristiques principales sont :

— **Découplage Temporel (Architecture Producteur-Consommateur) :** Les composants du jeu (Serveur, Sessions, Bots) n'écrivent jamais directement sur le disque. Ils déposent leurs messages dans une file d'attente partagée non-bloquante. Un thread consommateur unique se charge ensuite de l'écriture physique en arrière-plan.

— **Standardisation et Analyse :** Le module impose un format strict à tous les messages (Timestamp, Niveau de gravité, Contexte). Cette structuration en amont a pour but de faciliter le débogage et de permettre une future analyse statistique automatisée des parties (via des scripts `awk` par exemple).

— **Intégrité et Sécurité :** Le cycle de vie du logger est géré pour assurer qu'aucun message critique n'est perdu, même en cas d'arrêt du serveur, grâce à une gestion explicite des tampons système (*flushing*) après chaque écriture.

3.10 Module : Game (Moteur Logique et Algorithmique)

Le module **Game** encapsule l'intégralité des règles du "6 qui prend". Il agit comme un moteur purement logique, totalement décorrélé des contraintes réseaux ou de la gestion des threads. Il reçoit des commandes (ex : "Le joueur A joue la carte 12") et retourne un nouvel état de jeu, sans se soucier de *comment* ces données sont transportées.

Note : Les structures de données (`game_t`) et les algorithmes de résolution de tour sont présentés dans le ***Rapport Technique (Chapitre 5 et Annexe)***.

Ses fonctions clés sont :

— **Modélisation Aostique :** Le module maintient l'état complet de la table (Plateau, Deck, Mains) via des structures de données abstraites. Le plateau est notamment représenté par un tableau linéaire dynamique, permettant de supporter facilement des variantes de règles (taille de plateau variable) sans refonte structurelle.

— **Algorithme de Résolution de Tour :** Le cœur du module réside dans une fonction de résolution complexe déclenchée une fois que tous les joueurs ont soumis leur carte. Cet algorithme implémente strictement les règles du jeu : tri des cartes, calcul de la "plus petite différence positive" pour le placement sur les lignes, et gestion des pénalités (ramassage de lignes).

— **Gestion des États de Jeu (State Machine) :** Pour gérer l'asynchronisme inhérent au jeu multijoueur (attente des coups), le moteur repose sur une Machine à États Finis (FSM). Elle permet de figer le jeu dans des états d'attente critique (ex : `WAITING_EXTRA` lorsqu'un joueur doit choisir une ligne à ramasser) et de garantir l'intégrité de la séquence de jeu.

3.11 Module : Session (Contrôleur et Conteneur d'Exécution)

Le module **Session** agit comme un conteneur autonome encapsulant une instance de jeu unique. Il constitue la couche intermédiaire essentielle ("Middleware") chargée d'orchestrer le

déroulement de la partie en transformant les requêtes asynchrones des clients en une séquence logique d'actions de jeu.

*Note : Le fonctionnement de la boucle d'événements interne (basée sur `poll`) ainsi que la gestion des messages sont décrits dans le **Rapport Technique (Chapitre 5)**.*

Ses fonctions architecturales principales sont :

— **Contrôle et Validation (Message Handling) :** Le module filtre et discrimine les flux entrants. Il sépare la gestion administrative (connexions, cycle de vie) des commandes de jeu, assurant une validation contextuelle (ex : respect du tour de jeu) avant de solliciter le moteur logique.

— **Gestion Flexible des Participants :** La session maintient dynamiquement l'état des utilisateurs, distinguant les joueurs actifs des spectateurs. Elle intègre également un mécanisme de résilience permettant de conserver l'état d'un joueur déconnecté temporairement, facilitant ainsi sa reconnexion transparente sans interrompre la partie.

3.12 Module : Display (Interface Utilisateur Terminal)

Le module **Display** transforme l'environnement console standard en une **TUI (Text User Interface)** riche et interactive. Son rôle est de convertir les données brutes reçues du réseau en une représentation visuelle structurée, palliant l'austérité habituelle des terminaux pour offrir une meilleure ergonomie.

*Note : Les algorithmes de rendu spécifiques (notamment le "Slice Rendering" pour l'affichage horizontal) et les détails de l'implémentation graphique sont documentés dans le **Rapport Technique (Chapitre 8)**.*

Ses principales fonctionnalités sont :

— **Moteur de Rendu Graphique (Unicode) :** Le module exploite les caractères Unicode étendus (*Box-drawing characters*) pour dessiner physiquement les cartes et le plateau. Il intègre un algorithme d'affichage par "strates" ("Slice Rendering") qui permet de contourner la contrainte d'écriture séquentielle du terminal pour aligner correctement de multiples cartes sur une même ligne.

— **Colorimétrie Sémantique (ANSI) :** L'utilisation intensive des séquences d'échappement ANSI permet de hiérarchiser l'information. Le code couleur est dynamique : l'apparence d'une carte change automatiquement selon son poids en "têtes de bœuf" (du neutre vers le rouge vif), permettant au joueur d'évaluer le danger d'une main en un coup d'œil.

— **Guidage Contextuel et HUD :** L'interface agit comme un assistant actif (Heads-Up Display). Elle adapte l'affichage et les menus d'aide en fonction de l'étape de la session (Lobby, Jeu, Résultats), guidant l'utilisateur pour qu'il ne voie que les commandes pertinentes à l'instant T.

3.13 Module : Protocol (Standardisation des Échanges)

Le module **Protocol** définit le "contrat" de communication strict entre le client et le serveur. Pour répondre aux exigences de performance du temps réel et minimiser la bande passante, nous

avons privilégié un format binaire compact ("struct-packing") plutôt qu'un protocole textuel verbeux.

Note : La structure des paquets et les mécanismes de sérialisation sont spécifiés dans le ***Rapport Technique (Chapitre 2)****.*

Ses piliers architecturaux sont :

— **Structure Rigoureuse (Header + Payload) :** Chaque échange suit une anatomie fixe composée d'un en-tête de validation (Magic Number, Version, Taille) et d'une charge utile variable. Cette standardisation permet de rejeter immédiatement les connexions parasites et de router efficacement les messages sans parsing complexe.

— **Abstraction du Flux TCP (Stream Handling) :** Le sous-module `proto_io` résout la complexité inhérente au protocole TCP (qui est un flux continu d'octets). Il implémente une couche d'abstraction capable de réassembler les paquets fragmentés (problème de fragmentation) et de gérer une file d'attente d'émission non-bloquante, garantissant l'intégrité des données sans jamais geler le thread principal.

4 Gestion de la concurrence

La gestion simultanée de multiples clients implique un accès concurrent aux ressources partagées, ce qui expose le système à des risques de conditions de course et d'interblocages. Afin d'éviter ces problèmes, une approche non bloquante a été privilégiée pour l'ensemble de l'architecture.

La majorité des échanges entre composants repose sur une architecture orientée événements et sur des structures de communication adaptées aux modèles SPSC et MPSC. Ce choix permet de limiter l'utilisation de mécanismes de synchronisation classiques et de réduire les coûts liés au verrouillage, tout en garantissant la cohérence des données.

Une exception notable concerne le module de log. Les opérations d'écriture sur disque nécessitant une stricte intégrité des données, une synchronisation explicite y est utilisée. L'accès au buffer de log est protégé par des verrous POSIX standards (`pthread_mutex`), garantissant qu'un seul thread puisse écrire à un instant donné et évitant toute interférence entre les messages.

5 Évolutions ultérieures et perspectives

Bien que l'architecture actuelle réponde aux exigences du projet, plusieurs axes d'évolution ont été identifiés pour un développement ultérieur :

— **Optimisation de la scalabilité :** Regroupement de plusieurs sessions légères au sein d'un même thread d'exécution (*session pooling*) afin de réduire le coût des commutations de contexte.

— **Distribution de la charge réseau :** Parallélisation de la couche d'entrées/sorties via plusieurs threads IO et utilisation de mécanismes du noyau Linux (par exemple `SO_REUSEPORT`) pour équilibrer les connexions entrantes.

— **Analyse de données et intelligence artificielle :** Mise en place d'un module statistique persistant pour l'analyse des parties et développement de bots adaptatifs basés sur des techniques d'apprentissage automatique.
— **Renforcement des tests réseau :** Simulation de conditions réseau dégradées (latence, gigue, pertes de paquets) afin de valider la robustesse du protocole et des mécanismes de reconnexion.

6 L'utilisation d'outils d'IA générative

6.1 Code

L'IA Générative comme Chatgpt ou bien Gemini ont été utiliser dans certaines parties de code, spécifiquement dans les parties suivantes :

— Module Bot : fonction shuffle (pour mélanger), fonction best_card_choice_deep (choisi la meilleur carte)
— Module Protocol : Conception de division de structure Serveur/Connexion et Code de Création des paquets
— Module Serveur : Aider à écrire fonction handle_read et handle_write et Génération du token de connexion
— Module Game : fonction mélange_head(pour mélanger de manière proportionnelle les têtes)
— Module Session : fonction push_modifcation_log(ajouter les modifications au log de manière écrire et non en terme conception)
— Module SupportStructure : la grande partit de spsc_buffer(Ring Buffer)
— Module User : Réécriture de User_loop (boucle général de l'utilisateur), Partie de traitement de commande console et Interface Debug
— Et une partie de la création du script Awk pour transformer notre test.log en rapport_de_session.pdf

La rédaction du rapport et du rapport technique on été utiliser par l'IA pour reformuler nos propos de manière structurer et soigné.

7 Conclusion

Ce projet nous a aidé a utilisé les différentes approches que nous avons vu en cours. Ce qui nous a permis de faire nos recherche de notre coté pour approfondir nos connaissance sur les structures Linux, la conception d'architecture , la gestion de concurrence , l'optimisation de serveur , la mise en oeuvre de Thread et d'autres. Certaine compétences lors du projet ce sont amélioré tel que le travail d'équipe , la répartition du travail et la recherche d'information. Nous avons passé beaucoup de temps sur ce projet ce qui nous a permis de mieux analyser les façons de modéliser notre structure de projet mais aussi détecter les problèmes et les solutions plus rapidement.

Liste de références

Références

[1] Mark Richards et Neal Ford. *Fundamentals of Software Architecture : An Engineering Approach*. O'Reilly Media, 2020.

[2] Michael Kerrisk. *The Linux Programming Interface : A Linux and UNIX System Programming Handbook*. No Starch Press, 2010.

[3] Brian "Beej Jorgensen" Hall. *Beej's Guide to Network Programming : Using Internet Sockets*. Guide en ligne.

[4] Linux Programmer's Manual. *open(2) - open and possibly create a file*. Man7.org. Disponible sur : `https://man7.org/linux/man-pages/man2/open.2.html`

[5] Linux Programmer's Manual. *epoll(7) - I/O event notification facility*. Man7.org. Disponible sur : `https://man7.org/linux/man-pages/man7/epoll.7.html`

[6] CppReference. *Atomic types in C (stdatomic.h)*. Disponible sur : `https://en.cppreference.com/w/c/language/atomic.html`

[7] Documentation du Noyau Linux (Kernel.org). *Circular Buffers*. Disponible sur : `https://www.kernel.org/doc/Documentation/circular-buffers.txt`