L3 Informatique

Rapport Technique

Système et Réseaux – Pega_S

Piton Léo, Semenov Egor

Novembre–Décembre 2025

Contents

1 Structures de base

1.1 Gestion du cycle de vie des objets

Fichiers /src/SupportStructure/fd_map.c et /include/SupportStructure/fd_map.h

Lors de l'utilisation de `epoll`, le serveur enregistre les descripteurs de fichiers avec des données utilisateur, qui sont ensuite retournées via la structure `epoll_event` lors de la survenue d'événements. Dans ce projet, ces données sont représentées par un pointeur vers une structure polymorphe. Cette approche permet de traiter les événements sans recherches supplémentaires, mais impose une contrainte importante: la mémoire associée à ces structures doit rester valide pendant toute la durée d'enregistrement du descripteur dans `epoll`, et être correctement libérée à la fin de son cycle de vie.

La tâche initiale consistait à disposer d'un mécanisme centralisé de suivi de toutes les structures allouées associées aux descripteurs de fichiers. La solution la plus évidente à première vue est l'utilisation d'une table de hachage associant un fd à un pointeur vers une structure. Toutefois, dans le cadre du projet, il était essentiel d'éviter les dépendances externes et de conserver une implémentation en C pur.

Il a été pris en compte que les descripteurs de fichiers sont attribués par le système d'exploitation de manière compacte: chaque nouveau fd correspond au plus petit entier non négatif libre. Cette propriété permet d'utiliser un simple tableau comme mécanisme de correspondance fd $\rightarrow$ objet. Ainsi, l'indice du tableau correspond directement au descripteur de fichier, et la valeur stockée est un pointeur vers la structure associée. Les cases libres du tableau contiennent `NULL`.

C'est ainsi qu'est apparue la structure `fd_map`, représentant un tableau dynamique extensible de pointeurs. L'ajout et la suppression d'éléments se résument à l'écriture ou à l'effacement de la case correspondante du tableau, ce qui rend l'implémentation extrêmement simple et prévisible. Cette implémentation est utilisée exclusivement dans un contexte mono-thread.

Au cours de l'évolution du projet, il est apparu que ce mécanisme est pratique non seulement pour les descripteurs de fichiers. Le serveur contient également d'autres entités à durée de vie dynamique: structures clients, routage des buffers de session. Celles-ci nécessitent également un suivi centralisé et une libération garantie des ressources. En conséquence, `fd_map` a été généralisée et complétée par un mécanisme de génération d'indices compacts pour des objets non liés aux fichiers. Cela a permis d'utiliser une infrastructure unifiée de gestion mémoire pour différentes sous-systèmes du serveur et de simplifier la procédure de nettoyage lors de la fermeture des connexions ou de l'arrêt du serveur.

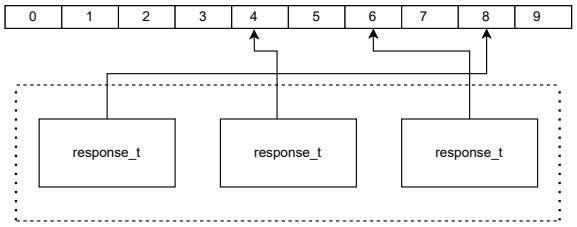


Figure 1: Structure fd_map

1.2 SPSC ring buffer

Fichiers /src/SupportStructure/spsc_buffer.c et /include/SupportStructure/spsc_buffer.h

L'architecture du serveur repose largement sur un modèle de traitement asynchrone des événements, basé sur une séparation des responsabilités entre les threads. Dans ce contexte, il est nécessaire de transmettre des messages entre threads sans blocage et avec un minimum de surcoûts. Ce problème est résolu à l'aide d'un canal de communication unidirectionnel avec un seul producteur et un seul consommateur (SPSC).

Le choix du modèle SPSC est dicté par l'architecture du système: chaque buffer possède strictement un thread producteur et un thread consommateur. Cela permet d'éliminer l'utilisation de mutex et d'implémenter une file de messages sous la forme d'un buffer circulaire sans verrou (lock-free). L'implémentation repose sur le ring buffer classique décrit dans la documentation du noyau Linux.

Le buffer est représenté par un tableau contigu de taille fixe et deux compteurs — `head` et `tail`. Le producteur incrémente `head` lors de l'écriture de nouveaux éléments, tandis que le consommateur incrémente `tail` lors de leur lecture. Chaque thread ne modifie que son propre compteur, ce qui exclut toute condition de course. La synchronisation repose sur l'utilisation de variables atomiques du standard C11 (_Atomic).

La correction de l'interaction entre les threads est assurée par l'utilisation des sémantiques mémoire `memory_order_release` lors de l'écriture des données et `memory_order_acquire` lors de leur lecture. Cela garantit que le consommateur observe toujours des éléments du buffer entièrement écrits.

La taille du buffer est définie comme une puissance de deux. Ce choix a une justification pratique: au lieu d'une opération coûteuse de modulo, une simple opération binaire "et" avec un masque est utilisée. L'indice d'un élément est calculé comme `pos & mask`, où `mask = size - 1`. Cette approche accélère significativement l'accès aux éléments et simplifie le calcul du nombre de cases occupées dans le buffer, par exemple via l'expression `(head - tail) & mask`.

Ainsi, le SPSC ring buffer implémenté constitue un mécanisme efficace et fiable de communication inter-threads, répondant pleinement aux exigences d'une architecture asynchrone orientée événements.

2 Protocole réseau

Fichiers src/Protocol/protocol.c et include/Protocol/protocol.h

Afin d'organiser l'échange de données entre le client et le serveur dans le cadre d'une architecture asynchrone, un protocole de transport spécifique a été développé. Ce protocole est destiné exclusivement à un usage interne au projet et résout le problème de la livraison fiable et non ambiguë des messages de niveau applicatif au-dessus de TCP.

Tous les messages du protocole possèdent un format unifié et se composent de deux parties logiques: un en-tête (Header) et un corps de paquet (Payload). L'en-tête a une taille fixe et est placé au début du message ; il est suivi d'un corps de longueur variable.

2.1 Structure de l'en-tête

L'en-tête contient un ensemble minimal de champs nécessaires:

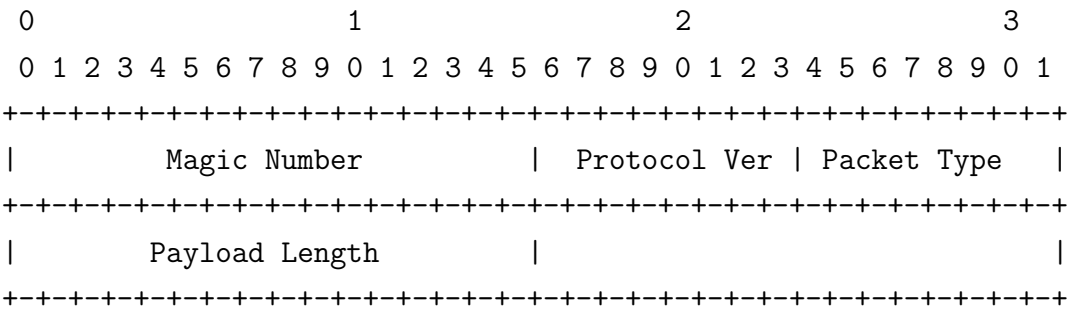
- un nombre magique permettant de vérifier rapidement la validité du flux de données entrant ;
- la version du protocole afin d'assurer la rétrocompatibilité ;
- le type de paquet, définissant la sémantique du corps du message ;
- la longueur du corps du paquet en octets, hors en-tête.

Cette composition de l'en-tête est le résultat de l'évolution du protocole. Aux premières étapes, l'inclusion des identifiants utilisateur et de session dans l'en-tête a été envisagée, mais cette décision a finalement été abandonnée.

Raisons de cet abandon:

- la présence d'identifiants utilisateur au niveau du transport introduit des risques de sécurité supplémentaires (usurpation, désynchronisation des états) ;
- cela viole le principe de séparation des niveaux d'abstraction, le niveau transport commençant à "connaître" la logique applicative ;
- les informations correspondantes sont déjà déterminées de manière univoque par l'état de la connexion côté serveur.

Il convient également de souligner le choix de stocker dans l'en-tête la longueur du corps du paquet, et non la taille totale du message. Étant donné que l'en-tête a une taille fixe, cela simplifie le traitement du flux entrant et permet au niveau applicatif de travailler exclusivement avec la charge utile, sans connaissance des détails de l'implémentation du transport.



Où:

- Magic Number — valeur fixe utilisée pour la validation du protocole ;
- Protocol Ver — version du protocole ;
- Packet Type — type de paquet ;
- Payload Length — longueur du corps du paquet en octets.

2.2 Représentation des paquets en mémoire

Au niveau de l'implémentation, tous les paquets du protocole sont représentés sous forme de buffers ordinaires de type `uint8_t*`. Cela les rend particulièrement simples à transporter via des sockets réseau et à intégrer avec des entrées-sorties non bloquantes.

Le parsing d'un message entrant se résume aux étapes suivantes:

1. lecture d'un nombre fixe d'octets correspondant à l'en-tête ;
2. transtypage du pointeur `uint8_t*` en pointeur vers la structure de l'en-tête ;
3. lecture directe des champs de la structure ;
4. lecture ultérieure du corps du paquet en fonction de la longueur indiquée dans l'en-tête.

Cette approche est rendue possible grâce à un contrôle strict de l'alignement des structures en mémoire.

```
#pragma pack(push, 1)
typedef struct {
    ...
} packet_t;
#pragma pack(pop)
```

L'utilisation de `#pragma pack(1)` garantit l'absence de bytes de remplissage (padding) ajoutés par le compilateur pour l'alignement. Ceci est critique, car la représentation binaire de la structure doit correspondre exactement au format des données transmises sur le réseau.

2.3 Extensibilité du protocole

Le protocole a été conçu dès l'origine comme extensible. L'ajout d'un nouveau type de paquet ne nécessite aucune modification des formats existants ni de la logique du niveau transport. Il suffit de:

- définir un nouveau Packet Type ;
- décrire la structure correspondante du corps du paquet ;
- ajouter un gestionnaire côté client et côté serveur.

Cela permet de faire évoluer la logique applicative indépendamment du sous-système de transport et de préserver la stabilité de l'infrastructure de communication de base.

3 Noyau réseau

Fichiers /src/Server/server.c et /include/Server/server.h Le noyau réseau du serveur est implémenté sous la forme d'une boucle event-driven mono-thread et est responsable de la réception et de l'envoi des données réseau, ainsi que du traitement primaire des messages du protocole. Le cœur du noyau repose sur le mécanisme `epoll` en mode `EPOLLET`, ce qui permet de gérer efficacement un grand nombre de connexions simultanées sans blocage ni scrutation active.

La boucle principale du serveur reste en attente jusqu'à l'apparition d'événements. Lors du déclenchement de `epoll_wait`, le serveur reçoit un tableau de structures `epoll_event`, chacune contenant un pointeur vers une structure utilisateur préalablement enregistrée. Ainsi, le serveur accède immédiatement au contexte courant de l'événement sans recherches ni correspondances supplémentaires.

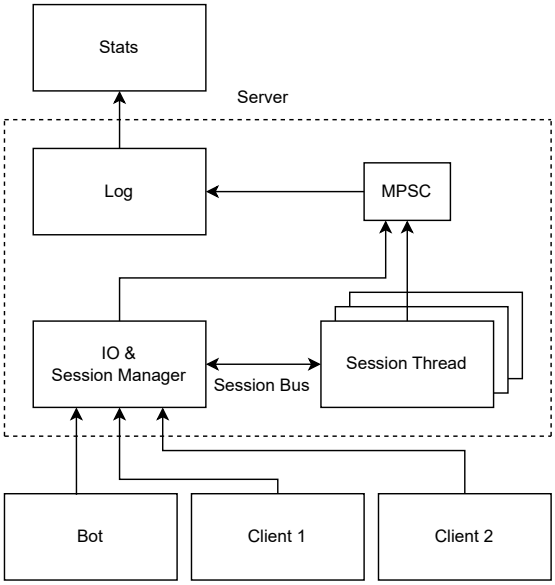


Figure 2: Architecture globale

3.1 Types d'événements et modèle de traitement

Dans le cadre du noyau réseau, trois types logiques d'événements sont distingués.

Le premier type correspond aux événements de connexion. Ils surviennent lorsque le socket d'écoute est prêt à accepter une nouvelle connexion. Dans ce cas, le serveur accepte la connexion entrante, place le socket en mode non bloquant et crée une structure de connexion `server_conn_t`, qui décrit intégralement l'état du client au niveau du transport.

Le deuxième type regroupe les événements provenant des clients. Il s'agit des événements de lecture et d'écriture apparaissant sur les sockets utilisateurs. Leur traitement dépend de l'état courant de la connexion et des indicateurs configurés dans `epoll`.

Le troisième type concerne les événements internes issus des sessions de jeu. Ces événements sont transmis via un mécanisme de notification distinct et sont examinés dans la section suivante consacrée aux interactions inter-threads.

Afin d'unifier le traitement, la structure `epoll_event` contient un pointeur vers une structure polymorphe `response_t`, qui inclut le type d'événement et un pointeur générique. Selon le type d'événement, ce pointeur référence soit une structure de connexion (`server_conn_t`), soit un objet de bus de messages de session (`session_bus_t`).

3.2 Structure de connexion

Lors de la création d'une nouvelle connexion client, le serveur instancie une structure `server_conn_t`, dont la durée de vie est strictement limitée à celle du socket correspondant.

```
typedef struct server_conn_s {
    int fd;
    conn_state_t state;

    rx_state_t rx;
    tx_state_t tx;

    int want_epollout;

    server_player_t* player; // NULL before registration/reconnect
} server_conn_t;
```

Cette structure représente l'utilisateur au niveau du transport et contient:

- le descripteur de fichier du socket ;
- l'état courant de la connexion ;
- l'état de réception des données ;
- la file d'envoi ;
- l'indicateur de nécessité d'attente de l'événement `EPOLLOUT`.

Ainsi, `server_conn_t` encapsule l'ensemble de la logique réseau et ne contient aucune logique applicative.

3.3 Réception des données et assemblage des paquets

Le code correspondant est implémenté dans `src/Server/server.c`, dans la fonction `server_main()`.

Tous les sockets clients sont placés en mode non bloquant. En conséquence, les données peuvent arriver de manière fragmentée, et les tentatives de lecture peuvent se terminer par les erreurs `EAGAIN` ou `EWOULDBLOCK`. Ce comportement est normal et permet au serveur de traiter un grand nombre de clients sans bloquer la boucle principale de gestion des événements.

3.3.1 Spécificités de la réception en mode EPOLLET

Le projet utilise le mécanisme `epoll` en mode `EPOLLET` (edge-triggered). Dans ce mode, la notification de disponibilité en lecture d'un socket n'est émise qu'en cas de changement d'état, et non à chaque présence de données dans le buffer d'entrée du noyau.

Cela impose l'obligation de lire intégralement les données disponibles sur le socket lors de la réception d'un événement de lecture. Le serveur doit continuer à appeler `recv` jusqu'à ce que l'opération se termine par l'erreur `EAGAIN` ou `EWOULDBLOCK`. Dans le cas contraire, une nouvelle notification peut ne pas être émise et une partie des données resterait non lue.

Pour cette raison, la lecture depuis le socket est toujours effectuée dans une boucle, et les données partiellement reçues sont stockées dans un buffer interne à la connexion. Cette

approche garantit un fonctionnement correct en mode edge-triggered et permet d'éviter à la fois la perte de données et l'appelle active des sockets.

3.3.2 Buffer d'entrée et assemblage des paquets

Le code correspondant est implémenté dans `src/Server/server.c`, dans la fonction `handle_read()`.

Le buffer d'entrée (`rx_buffer`) est utilisé comme stockage intermédiaire pour les données reçues depuis le socket. Sa présence est justifiée par le fait que les frontières des paquets du protocole applicatif ne coïncident pas avec celles des appels système `recv`: un seul appel peut retourner une partie de paquet ou plusieurs paquets consécutifs.

Après chaque lecture, les données sont ajoutées au buffer d'entrée, puis le serveur tente d'en extraire un ou plusieurs paquets complètement assemblés. L'assemblage s'effectue par étapes: d'abord la vérification de la présence d'un en-tête complet, puis la vérification de la quantité de données suffisante pour le corps du paquet, conformément à la longueur indiquée dans l'en-tête.

Si les données sont insuffisantes, l'analyse est interrompue jusqu'à la réception de la portion suivante. Si un paquet est assemblé intégralement, il est extrait du buffer, et les données restantes sont déplacées au début de celui-ci. Ainsi, le `rx_buffer` peut contenir simultanément la fin d'un paquet précédent et le début du suivant.

L'utilisation d'un buffer d'entrée séparé permet de distinguer clairement la logique d'entrée réseau du traitement applicatif et simplifie considérablement l'implémentation du protocole.

3.4 Traitement primaire des paquets

Après un assemblage réussi, le paquet est transmis au module de traitement primaire. Une partie de la logique est volontairement conservée au niveau du serveur et n'est pas entièrement déléguée à la couche de session, ce qui permet de réduire le couplage des composants et d'éviter la duplication de code.

L'implémentation de cette étape est concentrée dans le fichier `src/Server/packet_handler.c`, au sein de la fonction `handle_packet_main()`, qui constitue le point central de routage des paquets entrants.

Le cœur de cette étape est la fonction `packet_handler()`, responsable du traitement des paquets liés:

- à la procédure de poignée de main ;
- à l'enregistrement de l'utilisateur ;
- à la connexion et à la reconnexion aux sessions.

3.5 Handshake et enregistrement de l'utilisateur

Le protocole prévoit une séquence d'action (Handshake) sous la forme d'un échange de paquets `HELLO` et `WELCOME`. Cette étape permet de vérifier la validité de la connexion et la cohérence de la version du protocole entre le client et le serveur.

Après le Handshake réussie, l'enregistrement de l'utilisateur est effectué. Dans le projet, une séparation claire a été établie entre l'entité de transport `server_conn_t` et l'entité logique

server_player_t. Cette décision permet à l'utilisateur de se reconnecter en cas de coupure de connexion sans avoir à rejoindre de nouveau la session de jeu.

```
typedef struct server_player_s {
    player_state_t state;

    uint32_t user_id;
    uint32_t session_id;

    char* reconnection_token;
    struct server_conn_s* conn; // NULL if disconnected
} server_player_t;
```

La structure server_player_t contient un identifiant utilisateur unique, un identifiant de session de jeu et un jeton de reconnexion. Contrairement à la structure de connexion, la durée de vie du joueur n'est pas limitée à celle du socket.

La gestion de ces objets repose sur le mécanisme fd_map décrit précédemment, qui est utilisé ici comme une table des joueurs enregistrés. Il permet d'attribuer des identifiants compacts, de retrouver rapidement les joueurs par user_id et de traiter correctement les reconnexions.

3.6 File d'envoi et buffer de sortie (tx_buffer)

Fichiers /src/Protocol/protocol_io.c et /include/Protocol/protocol_io.h L'envoi de données en mode non bloquant nécessite également une mise en tampon. L'appel système send peut transmettre uniquement une partie des données ou retourner une erreur indiquant une impossibilité temporaire d'écriture. De plus, le serveur doit souvent garantir l'envoi de plusieurs paquets dans un ordre strict.

Pour cela, une file d'envoi (tx_buffer) est utilisée, implémentée sous la forme d'une liste chaînée de buffers. Chaque élément de la file contient un pointeur vers les données, un décalage de la position courante d'envoi et la longueur du buffer. Lors de l'ajout d'un nouveau paquet, le serveur tente d'abord de l'envoyer immédiatement. Si cela est impossible ou partiel, le reste des données est placé dans la file.

Lorsqu'il existe des données dans la file, l'attente de l'événement EPOLLOUT est activée pour le socket. À la réception de cet événement, le serveur poursuit l'envoi des données depuis la file jusqu'à son épuisement complet ou jusqu'à l'apparition de l'erreur EAGAIN.

Un tel mécanisme assure un fonctionnement correct avec des sockets non bloquants et permet:

- d'éviter les blocages et l'attente active ;
- de gérer correctement les envois partiels ;
- de préserver l'ordre de transmission des paquets indépendamment de l'état du buffer réseau.

4 Interaction inter-threads

Fichiers /src/SupportStructure/session_bus.h et /include/SupportStructure/session_bus.c

Dans l'architecture du serveur, la logique de jeu est exécutée dans des threads distincts, isolés du noyau réseau. Pour l'échange de données entre le serveur et les sessions, un mécanisme de communication inter-threads a été conçu, fondé sur la transmission de messages via un bus d'événements spécialisé. Cette approche permet de séparer strictement les responsabilités entre les niveaux réseau et applicatif et d'éviter les appels directs entre threads.

4.1 Structure des messages

Au cours de la conception, il est apparu que les événements échangés entre les threads peuvent avoir des natures différentes. Une partie des messages est initialisé par les actions des utilisateurs, tandis qu'une autre partie concerne l'état interne du système et n'est pas directement liée à un paquet réseau spécifique. Dans ce contexte, une structure de message polymorphe unifiée a été introduite, permettant d'identifier de la même manière le type d'événement et son contenu.

```
typedef struct {
    session_message_type_t type;
    union {
        system_message_t system;
        user_message_t user;
    } data;
} session_message_t;
```

Le champ `type` définit l'origine et la sémantique du message, tandis que l'union `union` permet de stocker les données dans le format approprié sans conversions supplémentaires.

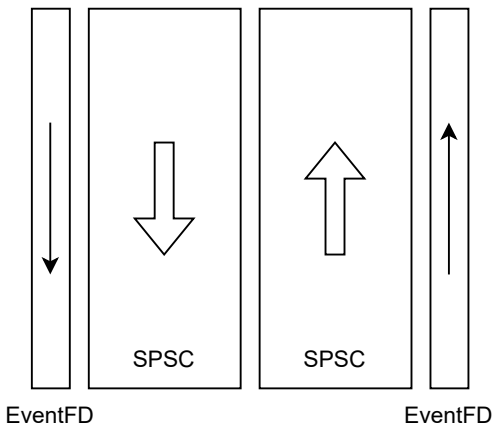


Figure 3: Session bus

4.2 Messages utilisateur

Les messages de type utilisateur sont générés par le serveur après la réception et le traitement primaire d'un paquet réseau, puis transmis à la session de jeu.

```
typedef struct {
    uint32_t client_id;
    uint8_t packet_type;
    uint16_t payload_length;
    uint8_t buf[SP_MAX_FRAME];
} user_message_t;
```

Afin de préserver une séparation correcte des niveaux d'abstraction, l'en-tête du paquet réseau n'est pas transmis à la couche applicative. À la place, le serveur en extrait les informations nécessaires et les transmet explicitement avec payload. En particulier, le message contient:

- l'identifiant interne de l'utilisateur, permettant d'identifier de même manière le joueur au sein de la session ;
- le type de paquet, déterminant la logique de traitement ultérieure ;
- la longueur de payload ;
- le buffer de données sans l'en-tête du protocole.

Cette approche permet à la logique de jeu d'isoler complètement des détails du protocole de transport et de ne traiter que des événements applicatifs.

4.3 Messages système

Outre les événements utilisateur, le bus de messages est utilisé pour la transmission de notifications système reflétant les changements d'état du serveur et des sessions.

```
typedef struct {
    system_message_type_t type;
    uint32_t session_id;
    uint32_t user_id;
    uint8_t capacity;
    uint8_t player_count;
    uint8_t is_visible;
    uint8_t bot_difficulty;
} system_message_t;
```

Le type de message système est défini par l'énumération suivante:

```
typedef enum {
    SESSION_UNREGISTER, SESSION_UNREGISTERED,
    USER_CONNECTED, USER_DISCONNECTED,
    USER_UNREGISTER, USER_LEFT,
    UPDATE_STATE, ADD_BOT
} system_message_type_t;
```

Les messages système sont utilisés pour résoudre les tâches suivantes:

- notifier le serveur du début et de la fin de la procédure de suppression d'une session ;
- informer une session de la connexion ou de la déconnexion d'un utilisateur ;
- assurer la désinscription correcte d'un utilisateur en cas d'erreurs de connexion ;
- mettre à jour les métadonnées de la session stockées côté serveur (nombre de joueurs, visibilité, paramètres de difficulté, etc.).

L'utilisation d'une catégorie distincte de messages système permet d'éviter la surcharge du canal utilisateur et de rendre l'interaction entre le serveur et les sessions plus formalisée.

5 Session

Fichiers /src/Session/session.c et /include/Session/session.h La session représente un contexte isolé d'exécution de la logique de jeu et fonctionne indépendamment du noyau réseau du serveur. Comme le serveur, la session est construite selon un modèle event-driven et traite les événements dans sa propre boucle de traitement.

La session est lancée dans un thread distinct et attend des notifications via `eventfd`. L'utilisation de `eventfd` permet de réveiller le thread sans blocage ni attente active, et s'intègre naturellement dans l'architecture asynchrone du serveur. Contrairement aux variables conditionnelles, ce mécanisme ne nécessite pas la possession d'un mutex et simplifie l'interaction entre threads.

5.1 Boucle de traitement des événements de la session

Le fonctionnement de la session repose sur une boucle de traitement des messages, activée lors de la réception d'une notification sur `eventfd`. Après son réveil, la session extrait les messages du bus et les traite de manière séquentielle.

Selon le type de message, celui-ci est transmis à l'une des deux fonctions suivantes:

- `handle_system_message()` — traitement des messages système ;
- `handle_game_message()` — traitement des événements de jeu.

Les messages système incluent les notifications de connexion et de déconnexion des utilisateurs, les requêtes d'ajout de bots, les modifications des paramètres de la session et les signaux de terminaison. Les messages de jeu contiennent les événements influençant directement le déroulement de la partie: démarrage du jeu, actions des joueurs, requêtes d'informations supplémentaires.

Les messages de réponse sont générés par la session et placés dans le bus des messages sortants. Le serveur les transmet ensuite soit aux utilisateurs concernés, soit les traite comme des événements système.

5.2 Gestion des ressources et cycle de vie

Lors de la création d'une session, l'ensemble des ressources associées est initialisé, y compris le bus de messages et les structures de la logique de jeu. Se pose alors la question de la responsabilité de la libération de ces ressources.

Si le nettoyage était effectué par le serveur, une situation pourrait survenir où la session aurait déjà reçu un message mais serait supprimée avant la fin de son traitement. De plus, le serveur devrait maintenir un registre des sessions actives et implémenter une logique supplémentaire de nettoyage, ce qui complexifierait l'architecture et augmenterait les surcoûts.

Dans le projet, il a été décidé que la session gère elle-même son cycle de vie. À cette fin, un mécanisme de suppression en trois étapes a été implémenté:

1. la session notifie le serveur de son intention de terminer ;
2. le serveur retire la session des registres de routage et confirme la désinscription ;
3. la session libère ses propres ressources et termine son exécution.

La session est lancée en mode `pthread_detach`, ce qui lui permet de se terminer de manière autonome, sans contrôle externe.

5.3 Joueurs et rôles

Au sein d'une session, il existe deux types de clients: les joueurs et les observateurs. Le nombre de joueurs est fixé lors de la création de la session. Les utilisateurs rejoignant après le début de la partie ou en l'absence de places libres obtiennent automatiquement le rôle d'observateur.

Les joueurs et les observateurs sont stockés dans un même tableau. Cela permet de modifier dynamiquement le rôle d'un utilisateur sans changer la structure de données, ce qui est important en cas de reconnexion ou de déconnexion en cours de partie. Les identifiants internes des joueurs au sein de la session ne sont pas compacts, ce qui rend l'utilisation de `fd_map` inappropriée dans ce contexte.

Les joueurs sont représentés par la structure suivante: (include/Session/player.h)

```
typedef struct {
    uint32_t player_id;
    int* player_cards_id;
    uint8_t nb_head;
    player_role_t role;
    uint8_t is_connected;
} player_t;
```

Cette structure contient toutes les informations nécessaires à la logique de jeu. Aucune séparation supplémentaire des entités entre les niveaux transport et applicatif n'est appliquée au sein de la session, car cela entraînerait des problèmes de cohérence.

La session conserve les informations relatives à son créateur, car lui seul est autorisé à démarrer la partie et à supprimer la session. En cas de départ du créateur, son rôle est automatiquement transféré à un autre joueur.

5.4 Logique de jeu

Fichiers /src/Session/Game/game.c et /include/Session/Game/game.h Le code de la logique de jeu est concentré dans le module `Session/Game` et est divisé en deux parties: la logique de démarrage de la partie et la boucle principale de jeu. Toutes les données relatives à l'état du jeu sont stockées dans la structure `game_t`.

```
typedef struct {
    % parametres de la partie (voir include/Session/Game/game.h)
    uint32_t extra_wait_from;
    state_t game_state;
    card_t* deck;
    int* board;
    int* card_ready_to_place;
    int card_ready_to_place_count;
} game_t;
```

5.4.1 Logique de démarrage de la partie

Le démarrage de la partie est réservé exclusivement au créateur de la session. L'initialisation du jeu est effectuée une seule fois et consiste en une séquence d'étapes déterministes garantissant un état initial correct de la partie.

Les cartes du jeu sont représentées par la structure `card_t`:

```
typedef struct {
    int num;
    int numberHead;
    uint32_t client_id;
} card_t;
```

Au cours de la partie, les cartes ne sont pas copiées entre les entités. Pour les besoins de la logique de jeu, chaque carte est identifiée par son propre indice dans le tableau `deck`. Cette approche permet d'éviter la copie des structures, de simplifier la transmission des données entre sous-systèmes et de préserver la cohérence des informations relatives au propriétaire de la carte et à l'historique de son placement sur le plateau.

Lors de la première étape, le tableau des cartes `deck` est initialisé. Chaque carte se voit attribuer un numéro unique et un nombre aléatoire de "têtes" conformément aux règles du jeu. Un tableau auxiliaire d'indices de cartes est ensuite créé, contenant la séquence des entiers de `0` à `nbrCards - 1`.

Ce tableau d'indices est soumis à un premier mélange, puis transmis à la fonction `melange_head()`, chargée de la répartition des valeurs de pénalité (têtes) entre les cartes.

Après l'attribution des pénalités, le tableau d'indices est mélangé une seconde fois. Ce mélange est alors utilisé directement pour la distribution des cartes aux joueurs. Les cartes sont distribuées en copiant les indices depuis le tableau vers la structure du joueur correspondant. La distribution s'effectue séquentiellement à partir de la position courante dans le tableau

d'indices, garantissant l'absence de doublons et une répartition uniforme des cartes entre les participants.

Chaque joueur reçoit un nombre fixe de cartes, déterminé par les paramètres de la session. Après la distribution, les cartes restantes sont considérées comme inactives et ne participent pas à la partie en cours.

À l'issue de toutes les étapes d'initialisation, la structure `game` est placée dans l'état `WAITING`, ce qui indique que la partie est prête à recevoir les premiers coups des joueurs et à entrer dans la boucle principale de jeu.

5.4.2 Boucle principale de jeu

Les messages des utilisateurs arrivent de manière asynchrone, mais grâce à l'architecture du serveur, ils sont traités séquentiellement. Cela permet de considérer le traitement d'un coup comme une opération atomique.

Pour chaque coup, une vérification de validité est effectuée, après quoi l'indice de la carte choisie est enregistré dans le tableau `card_ready_to_place`. L'indice interne du joueur est utilisé, ce qui est possible grâce au stockage des joueurs et des observateurs dans un même tableau. Pour les utilisateurs inactifs, les valeurs correspondantes sont égales à `-1`.

Une fois que tous les joueurs ont effectué leur coup, le tableau des cartes jouées est trié puis placé sur le plateau conformément aux règles du jeu. Si une carte ne peut pas être placée, une requête d'information supplémentaire est envoyée au joueur concerné, et la partie passe dans l'état `WAITING_EXTRA`. L'identifiant du joueur attendu est enregistré dans `extra_wait_from` afin d'empêcher toute substitution.

Après le traitement de la requête supplémentaire, les cartes sont placées et les points de pénalité sont attribués. À la fin de chaque coup, la condition de fin de partie est vérifiée — l'absence de cartes dans la main des joueurs.

L'état courant de la partie est diffusé aux joueurs et aux observateurs à l'aide du paquet `PKT_SESSION_INFO`.

6 Log

Fichiers /src/Log/log.h et /include/Log/log.c Le serveur développé est orienté vers de hautes performances et un traitement asynchrone des événements. Dans ce contexte, la sous-système de log revêt une importance particulière, car elle permet d'analyser le comportement du système sous charge et d'identifier les goulets d'étranglement. Dans le cadre du projet, le log a été utilisée principalement pour collecter des informations sur les erreurs liées à la gestion de la mémoire, celles-ci constituant potentiellement un facteur limitant pour la scalabilité du serveur.

6.1 Structure de log

La tâche de log diffère fondamentalement de celle de la transmission de messages entre le serveur et les sessions. Alors que le bus de messages repose sur un modèle à un producteur et un consommateur (SPSC), le système de log comporte de multiples sources de messages: le noyau réseau, les sessions de jeu, les sous-systèmes de traitement des paquets et de gestion des ressources. Le consommateur, en revanche, reste unique.

Ainsi, un modèle MPSC (Multiple Producer, Single Consumer) a été retenu pour la log. Contrairement au buffer SPSC utilisé précédemment, cette structure autorise des écritures concurrentes depuis plusieurs threads. L'implémentation repose sur l'utilisation d'un `pthread_mutex_t`, acquis lors de l'ajout d'un message dans la file de log et libéré à la fin de l'opération.

Cette approche garantit la correction du fonctionnement lors de la génération simultanée de logs depuis différentes parties du système et ne nécessite pas d'algorithmes lock-free complexes, la log ne se situant pas sur le chemin critique du traitement des événements clients.

6.2 Architecture de la sous-système de log

La sous-système de log est exécutée dans un thread distinct. Ce choix s'explique par le fait que les opérations d'écriture sur disque sont lentes et potentiellement bloquantes. Leur exécution dans le thread réseau entraînerait une dégradation des performances et une augmentation des latences lors du traitement des requêtes clientes.

Au démarrage du serveur, le thread de log est créé et lancé en premier. Ensuite, le serveur principal est initialisé ; il conserve à la fois le descripteur du thread de log et un pointeur vers le bus de logs partagé (`log_bus`). Lors de la création des sessions de jeu, une référence au `log_bus` est transmise à chaque session, ce qui leur permet d'émettre des messages dans un journal unique sans interaction directe avec le serveur.

Le thread de log extrait périodiquement les messages de la file et les écrit sur un support externe. La lecture des logs s'effectue à une périodicité fixe (une fois par seconde). Théoriquement, cela peut conduire à un débordement de la file en cas d'intensité élevée de log ; toutefois, l'utilisation de `eventfd` n'est pas possible ici en raison de la nature multi-producteurs des sources de messages.

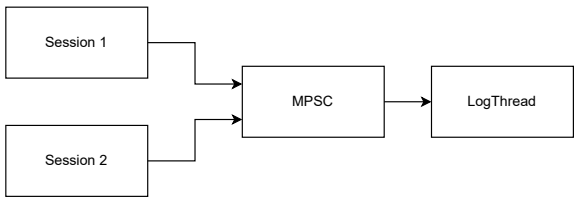


Figure 4: Schéma Log

6.3 Format des messages de log

Les messages de log possèdent un format unifié:

```
typedef enum {
    LOG_INFO, LOG_WARN,
    LOG_ERROR, LOG_DEBUG, LOG_SESSION } log_level_t;

typedef struct {
    uint64_t timestamp;
    uint32_t session_id;
    log_level_t level;
    char message[256]; } log_entry_t;
```

Chaque message contient un horodatage, un niveau de log et un texte descriptif. Un identifiant de session est également fourni afin d'indiquer la session à laquelle le message se rapporte. La valeur `session_id = 0` est utilisée pour les logs relatifs au serveur dans son ensemble. Cette distinction permet de filtrer les messages et d'analyser le comportement de sessions de jeu individuelles.

Les niveaux de log suivants sont utilisés dans le projet:

- `DEBUG` — messages destinés au débogage et à la vérification du bon fonctionnement du serveur ;
- `INFO` — informations sur les événements normaux, tels que la connexion d'un client ou la création d'une session ;
- `WARN` — messages signalant un comportement anormal mais tolérable, par exemple la réception d'un paquet incorrect ;
- `ERROR` — erreurs entraînant une perturbation du fonctionnement du serveur, notamment les erreurs d'allocation mémoire ;
- `SESSION` — niveau spécifique destiné à l'enregistrement des événements de jeu.

6.4 Fiabilité et modes d'émission des logs

Dans la majorité des cas, l'émission des messages de log est effectuée en mode non bloquant. Cela signifie qu'en cas de saturation de la file, un message de log peut être perdu. Ce comportement est acceptable pour les messages diagnostiques et informatifs, car leur perte n'affecte pas la correction du fonctionnement du système.

Les messages de niveau `SESSION` constituent une exception. Ils sont envoyés en mode bloquant, ce qui garantit leur livraison. La perte d'un seul de ces messages peut rendre impossible la reconstruction du déroulement d'une session de jeu ; la fiabilité est donc prioritaire sur les performances pour ce niveau.

Des messages de log ont été ajoutés dans toutes les sections critiques du serveur et de la logique de jeu, permettant une analyse détaillée du comportement du système et un diagnostic précis des erreurs.

6.5 Exemple de logs

Ci-dessous figure un extrait des logs collectés lors de l'exécution du serveur. Le format des messages est unifié et inclut l'horodatage, le niveau de log, l'identifiant de session et le texte du message.

```
[1765796426] [DEBUG]   [ID:3] Client 13 joined as PLAYER
[1765796426] [DEBUG]   [ID:0] server: accepted connection from
   127.0.0.1
[1765796426] [SESSION] [ID:1] START:0_73_1;1_42_2;2_101_2;3_30_3;
```

Dans cet exemple:

- le premier message enregistre la connexion d'un utilisateur à une session de jeu et l'attribution du rôle de joueur ;
- le second message concerne le serveur dans son ensemble et indique l'établissement d'une nouvelle connexion réseau ;
- le troisième message de niveau SESSION contient des informations sur le début de la partie et l'état initial des cartes, permettant de reconstruire le déroulement de la session de jeu.

Ce format de log facilite l'analyse du fonctionnement du serveur, simplifie le débogage et permet une reconstruction a posteriori des événements de jeu.

7 Client

Fichiers /src/User/user.c et /include/User/user.h Le code relatif à la partie cliente du système est situé dans le paquet `User` et dépend exclusivement du module `Protocol`. Le client constitue une entité autonome, entièrement séparée du serveur et interagissant avec celui-ci uniquement via le protocole réseau. À l'instar du serveur, le client est construit selon un modèle de traitement événementiel, bien que le mécanisme utilisé diffère.

7.1 Architecture de la boucle cliente

Le fonctionnement du client repose sur une boucle de traitement des événements implémentée à l'aide de l'appel système `poll`. Le choix de `poll` s'explique par la nécessité de surveiller simultanément les événements provenant du socket réseau et de l'entrée standard (`stdin`). Cela permet de traiter à la fois les paquets réseau entrants et les commandes saisies par l'utilisateur dans le terminal.

7.2 Réception et envoi des données

À l'instar du serveur, le client utilise un buffer d'entrée et une file de messages sortants. Cette approche permet de travailler avec le socket en mode non bloquant et de gérer correctement les lectures et écritures partielles.

La boucle de traitement des paquets réseau reproduit en grande partie la logique côté serveur:

- les données entrantes sont lues dans le buffer ;
- les paquets sont assemblés à partir de l'en-tête du protocole ;
- les paquets complets sont transmis au gestionnaire du niveau applicatif.

Cette similarité architecturale simplifie la maintenance du code et réduit la probabilité d'erreurs logiques.

7.3 Traitement des commandes utilisateur

Les commandes saisies par l'utilisateur via la console sont traitées indépendamment des paquets réseau. La logique de traitement des commandes est concentrée dans le module `/src/User/user_packet_ha` au sein de la fonction `user_handle_stdin()`.

La séparation entre le traitement des événements réseau et celui de l'utilisateur permet:

- d'éviter les blocages ;
- de maintenir un mode de fonctionnement interactif ;
- d'implémenter une interface textuelle simple sans impact sur la partie réseau du client.

7.4 Interface utilisateur

Le rendu de l'état du client et l'affichage des informations à l'utilisateur sont assurés par le module `Display`. Ce module est exclusivement responsable de la présentation visuelle et n'intervient ni dans le traitement des événements réseau ni dans la logique de jeu.

Une telle séparation permet de modifier ou d'étendre l'interface utilisateur indépendamment de la logique du client et du protocole d'interaction avec le serveur.

8 Interface Utilisateur

Fichiers /src/Display/display_session.c et /include/Display/display_session.h

L'interface initiale, purement fonctionnelle et basée sur des caractères ASCII simples (parenthèses, underscores), s'est avérée insuffisante pour offrir une lisibilité correcte du jeu. Nous avons donc développé un moteur de rendu graphique pour terminal (*Terminal User Interface* - TUI) plus avancé.

8.1 Conception Visuelle des Cartes (Unicode)

Pour améliorer l'immersion, nous avons remplacé la représentation textuelle abstraite par une représentation quasi-graphique utilisant les caractères **Unicode Box-Drawing**. Chaque carte est dessinée comme un objet rectangulaire structuré:

- **Contours:** Utilisation de traits épais pour délimiter clairement les cartes sur le fond noir du terminal.
- **Indicateurs de Valeur:** Le numéro de la carte (1-104) est affiché dans les coins supérieur-gauche et inférieur-droit pour garantir la lisibilité quel que soit le sens de lecture.

- **Indicateur de Pénalité:** Le centre de la carte affiche le nombre de "têtes de bœufs" (points de pénalité), représentés par des symboles distincts.

8.2 Algorithme de Rendu Horizontal ("Slice Rendering")

Une contrainte technique majeure des terminaux standards est l'affichage séquentiel ligne par ligne (flux de sortie standard `stdout`). Lors de nos premiers tests, l'affichage d'une main de 10 cartes s'effectuait verticalement (une carte sous l'autre), ce qui rendait le jeu injouable.

Pour obtenir un alignement horizontal des cartes (comme une vraie main de joueur), nous avons dû réécrire la logique d'affichage pour procéder par "tranches" (*slices*):

1. **Slice 1 (Haut):** Le programme itère sur toutes les cartes de la main et n'affiche que la bordure supérieure.
2. **Slice 2 (Milieu - Haut):** Il itère à nouveau pour afficher la ligne contenant le numéro supérieur.
3. **Slice 3 (Centre):** Il affiche la ligne centrale avec les têtes de bœuf pour toutes les cartes.
4. **Slice 2 (Milieu - bas):** Il itère à nouveau pour afficher la ligne contenant le numéro supérieur.
5. **Slice 4 (Bas):** Il termine par les bordures inférieures.

Cette approche, implémentée dans les fonctions `up_display_card`, `mid_display_card`, etc., permet de dessiner une rangée complète de N cartes comme un seul bloc de texte cohérent.

8.3 Sémantique des Couleurs et Ergonomie

L'interface utilise les séquences d'échappement **ANSI** pour enrichir l'information visuelle:

- **Dégradé de Danger:** La couleur des cartes est dynamique et dépend de leur poids en pénalité. Une carte inoffensive (1 tête) apparaît dans une couleur neutre, tandis que les couleurs deviennent progressivement plus vives (jaune $\rightarrow$ orange $\rightarrow$ rouge/violet) à mesure que le nombre de têtes augmente.
- **Agencement (HUD):** L'écran est divisé en zones fonctionnelles séparées par des bandeaux explicites: le Plateau de jeu (*Board*), la Main du joueur, et la zone de Status.
- **Guidage:** Un rappel constant des commandes disponibles est affiché, adapté au contexte (Lobby, Jeu, Choix de ligne), facilitant la prise en main sans nécessiter la mémorisation des commandes.

9 Bot

Fichiers /src/Bot/bot.c et /include/User/bot.h Les robots de jeu sont implémentés comme des clients autonomes et réutilisent, d'un point de vue architectural, la majorité des structures et des fonctions de la partie utilisateur. En pratique, un bot représente une fine surcouche au-dessus du code client, qu'il étend par des champs supplémentaires et une logique de prise de décision. Cette approche permet d'éviter la duplication du code réseau et du protocole, et garantit que les bots interagissent avec le serveur strictement par les mêmes mécanismes que les clients ordinaires.

La structure d'un bot est définie comme suit:

```
typedef struct {
    uint8_t level;

    user_t user;
    uint32_t session_id;
    int index_row_to_take;
    int is_creator;

    bot_state_machine_t state;
    time_t next_action_time;
    int id;

    int* placed_cards;
} bot_t;
```

La structure embarquée `user_t` est responsable de l'interaction réseau et du traitement des paquets du protocole, tandis que les champs supplémentaires sont utilisés pour stocker l'état du bot, le niveau de difficulté, la planification des actions et la prise de décisions de jeu.

9.1 Rôle du bot dans le système

Un bot peut soit créer sa propre session de jeu, soit rejoindre une session existante. Son comportement est entièrement déterminé par la logique cliente: le bot se connecte au serveur, passe par la procédure d'enregistrement et participe à la partie au même titre que les utilisateurs ordinaires.

Pour exécuter les actions de jeu, le bot implémente des gestionnaires pour les paquets clés du protocole, en particulier `PKT_SESSION_INFO` et `PKT_SESSION_REQUEST_EXTRA`. Sur la base des informations contenues dans ces paquets, le bot prend des décisions concernant le choix des cartes et les actions supplémentaires au cours de la partie.

9.2 Niveaux de difficulté des bots

Trois niveaux de difficulté des bots sont implémentés dans le projet, se distinguant par l'algorithme de prise de décision.

Premier niveau de difficulté. Ce niveau repose sur une évaluation locale du risque. Pour chaque carte disponible, le nombre attendu de points de pénalité susceptibles d'être obtenus lors de son jeu est calculé. La carte présentant la valeur de pénalité minimale est ensuite choisie. Cette logique est implémentée dans la fonction `calculate_danger()` et constitue un algorithme rapide, mais limité en termes de qualité des décisions.

Deuxième niveau de difficulté. Ce niveau utilise un algorithme de Monte-Carlo avec simulation de l'état courant du jeu. Le bot tient compte des cartes déjà jouées et, pour chacune de ses cartes, exécute une série de simulations (de l'ordre de 1000), dans lesquelles les cartes restantes sont distribuées aléatoirement aux adversaires. Les coups des adversaires dans les simulations sont sélectionnés à l'aide de l'algorithme du premier niveau de difficulté. À l'issue des simulations, le nombre moyen de points de pénalité est évalué et le coup le plus avantageux est choisi.

Troisième niveau de difficulté. Ce niveau met en œuvre une simulation Monte-Carlo approfondie. Contrairement au niveau précédent, le bot évalue non seulement les conséquences du coup courant, mais également son influence sur plusieurs coups ultérieurs. Cela permet de prendre en compte les effets différés des décisions et d'effectuer des choix plus stratégiques, au prix d'une complexité computationnelle accrue.

9.3 Intégration des bots avec le serveur

Une session peut initier l'ajout d'un bot en envoyant une requête correspondante au serveur. Dans ce cas, le serveur lance le bot en tant que processus autonome et lui transmet l'identifiant de la session à laquelle il doit se connecter. Le bot agit ensuite de manière entièrement autonome, en interagissant avec le serveur exclusivement via le protocole réseau.

Une telle solution permet:

- d'utiliser des bots pour le jeu en solitaire ;
- de faire évoluer le nombre de bots indépendamment du serveur ;
- de garantir l'identité de comportement des bots et des clients réels au niveau du protocole.

10 Tests de charge

Après l'achèvement du développement du serveur, il est devenu nécessaire d'évaluer son comportement sous forte charge et de déterminer le nombre de connexions simultanées et de sessions de jeu qu'il est capable de prendre en charge. À cette fin, des tests de charge ont été réalisés.

L'idée initiale consistait à lancer plusieurs centaines de bots à l'aide d'un script BASH afin qu'ils créent des sessions et participent aux parties. Toutefois, cette approche s'est révélée inefficace. La machine sur laquelle les bots étaient exécutés présentait une utilisation maximale de tous les cœurs du processeur, tandis que le serveur, lancé sur une machine distincte, ne subissait pratiquement aucune charge. Cela s'explique par le fait que chaque bot était un processus séparé et que la charge principale provenait des commutations de contexte du système d'exploitation, plutôt que des interactions réseau avec le serveur.

Dans ce contexte, il a été décidé de modifier l'approche de test et de générer une charge

intensive sur le serveur à partir d'une seule machine. L'objectif principal était de créer un grand nombre de connexions réseau simultanées sans surcoûts significatifs liés à la gestion des threads et des processus.

Pour la mise en œuvre de ce test, le mécanisme `epoll` a été utilisé, car il permet de gérer efficacement des milliers de sockets ouverts au sein d'un seul thread, sans attente active ni commutations fréquentes de contexte. Cela a permis de rapprocher le modèle de charge d'un scénario réel de fonctionnement du serveur avec un grand nombre de clients.

En complément, un mécanisme d'actions différées a été implémenté pour la sous-système de log, afin de simuler des délais entre l'envoi des paquets. Cette approche permet de rapprocher la nature du trafic du comportement réel des utilisateurs et d'éviter des charges artificiellement synchrones.

Dans le cadre des tests, un scénario distinct (`test_bot_V2.c`) a été développé, dans lequel 2000 bots étaient lancés et répartis sur 500 sessions de jeu. Malgré ce nombre significatif de clients et de sessions actives, le serveur n'a pas montré de signes notables de surcharge.

Les résultats obtenus s'expliquent par l'utilisation efficace de `epoll`, des entrées-sorties non bloquantes, ainsi que par la relative simplicité de la logique de jeu et par l'approche architecturale choisie, fondée sur la séparation des niveaux réseau et applicatif. Cela confirme que le serveur est capable de monter en charge et de fonctionner de manière stable avec un haut degré de parallélisme.

11 Conclusion

Dans le cadre de ce travail, un serveur asynchrone pour un jeu de cartes a été développé, fondé sur une architecture événementielle et des entrées-sorties non bloquantes. Le serveur a été conçu avec une séparation claire des responsabilités entre le niveau réseau, la sous-système de communication inter-threads et la logique applicative du jeu.

Une attention particulière a été portée à la gestion du cycle de vie des objets et des threads. Les sessions sont implémentées comme des entités autonomes gérant elles-mêmes leurs ressources, ce qui a permis d'éviter des schémas de synchronisation complexes et un nettoyage centralisé. La communication inter-threads repose sur des bus de messages spécialisés, adaptés à des scénarios d'interaction spécifiques.

La sous-système de log est exécutée dans un thread séparé et assure la collecte des informations diagnostiques et des événements de jeu sans impacter les performances du noyau réseau. L'implémentation de bots de jeu a démontré la possibilité de réutiliser le code client et a permis de réaliser des tests de charge du système.

Les résultats des tests de charge montrent que le serveur fonctionne de manière stable avec un grand nombre de connexions et de sessions simultanées, et que les choix architecturaux retenus permettent une montée en charge efficace du système. Dans l'ensemble, le serveur développé confirme la pertinence de l'architecture événementielle et des mécanismes d'entrées-sorties asynchrones pour la conception d'applications réseau à forte charge.

Liste de références

References

[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`