A Report on Tactical Token Race: A Hybrid Concurrent Multiplayer Game Server Architecture

CSN6214 Operating Systems

Submitted to

Mr Ng Hu

Lecture Section: TC3L
Tutorial Section: TT12L

Prepared by Group 4

Name	Student ID
Sathishkugan A/L Thangarajoo	1211103319
Thasayaini A/P Murthy	1211104170
Wan Hanani Iman Binti Wan Mohd Azidi @ Sapawi	242UC244CK
Jasmyne Yap	242UC244PT

Date of Submission: ---------- 2025

Game Description and Rules

1.1. Description

Tactical Token Race is a turn-based multiplayer board game where players compete to reach the final tile of a shared linear board. The game is managed entirely by the server, which enforces all rules, validates moves, and updates the game state. Clients act only as input/output terminals, sending simple commands and receiving updates.

The game supports up to 4 players per session. Each game round follows a deterministic turn order decided through an initial dice roll. Players take turns rolling a virtual die and moving their token forward on the board. The first player to reach the final tile wins the game.

1.2. Setup and Objective

- Players per game: 3–5.
- Host: The first player to connect becomes the Host and chooses the lobby size (3, 4, or 5 players).
- Start positions: All players start at position 0.
- Objective: Be the first player to reach position 30 or higher on the track.

1.3. Turn Order

1.3.1. Once the lobby is full, the server rolls a die for each player to determine the initial order.

1.3.2. Higher rolls act earlier; ties are broken with additional rolls.

1.3.3. The resulting order is stored and used in round robin fashion for the entire game:

- Player 1 → Player 2 → … → Player N → back to Player 1

1.3.4. At any time, only the current player is allowed to act. The server:

- Announces whose turn it is.
- Sends a “YOUR TURN” message only to that player.
- Waits for their choice before proceeding

1.4. Actions Per Turn

1. **SAFE Move**
    - Server rolls a 6-sided die.
    - Movement based on the roll:
        - Roll 1–2 → move +1 step.
        - Roll 3–4 → move +2 steps.
        - Roll 5–6 → move +3 steps.
    - SAFE moves never move the player backward.
    - Any active Token bonus (see §1.5) is added to this movement.
2. **RISK Move**
    - Server rolls a 6-sided die.
    - Two cases:
        - Success (4–6): move forward by the exact roll (e.g., roll 5 → +5).
        - Failure (1–3): move 1 step backward (−1).SAFE moves never move the player backward.

- Positions can never go below 0; if a backward move would go negative, the position is set to 0.
- Any active Token bonus is added to this movement, which can partially or completely cancel the −1 on a failed roll. Any activ SAFE moves never move the player backward.

3. **EXIT Game**
    - The player voluntarily leaves the game.
    - They are immediately marked as inactive and cannot win that game.
    - The game continues with the remaining players.

1.5. Token Mechanic

- Token positions: 5, 10, 15, 20, and 25 on the track.
- When a player lands exactly on one of these positions after a move, they gain a Token.
- A Token provides a +2 step bonus on the next turn only:
    - Example SAFE: base move +2, Token +2 → total +4.
    - Example RISK success: roll 6 → +6, Token +2 → total +8.
    - Example RISK failure: base −1, Token +2 → net +1.
- Example RISK success: roll 6 → +6, Token +2 → total +8.
- Example RISK failure: base −1, Token +2 → net +1.
- After the bonus is applied on the next turn, the Token is consumed and removed.

1.6. Winning, Losing, and Disconnections

Standard Win

- After a move (including any Token bonus), if a player's position is 30 or higher, that player wins immediately.The server announces the winner and the game ends.

Forfeit Win

- If, during a game, all other players leave or disconnect, the last remaining active player wins by forfeit.

Loss

- Any player who is not the winner when the game ends is considered to have lost.
- A player who exits or disconnects before the game ends automatically loses that game.

Host Disconnection

- If the Host disconnects before choosing the lobby size, the lobby is cancelled and all waiting players are removed.
- If the Host disconnects after the game has started, the game continues with the remaining players; the Host's turn slot is simply skipped in the turn order.

Deployment Mode (IPC)

1.1. Mode Overview

- Selected Mode: Single-Machine Mode using IPC named pipes (FIFOs).
- Environment: Linux

1.2. Rationale for IPC Choice

- Easier to manage within a single host lab environment.
- Directly reinforces OS concepts:
  - POSIX named pipes
  - Shared memory via mmap
  - Process-shared mutexes and semaphores
- No network configuration overhead (ports, firewall).

Hybrid Architecture

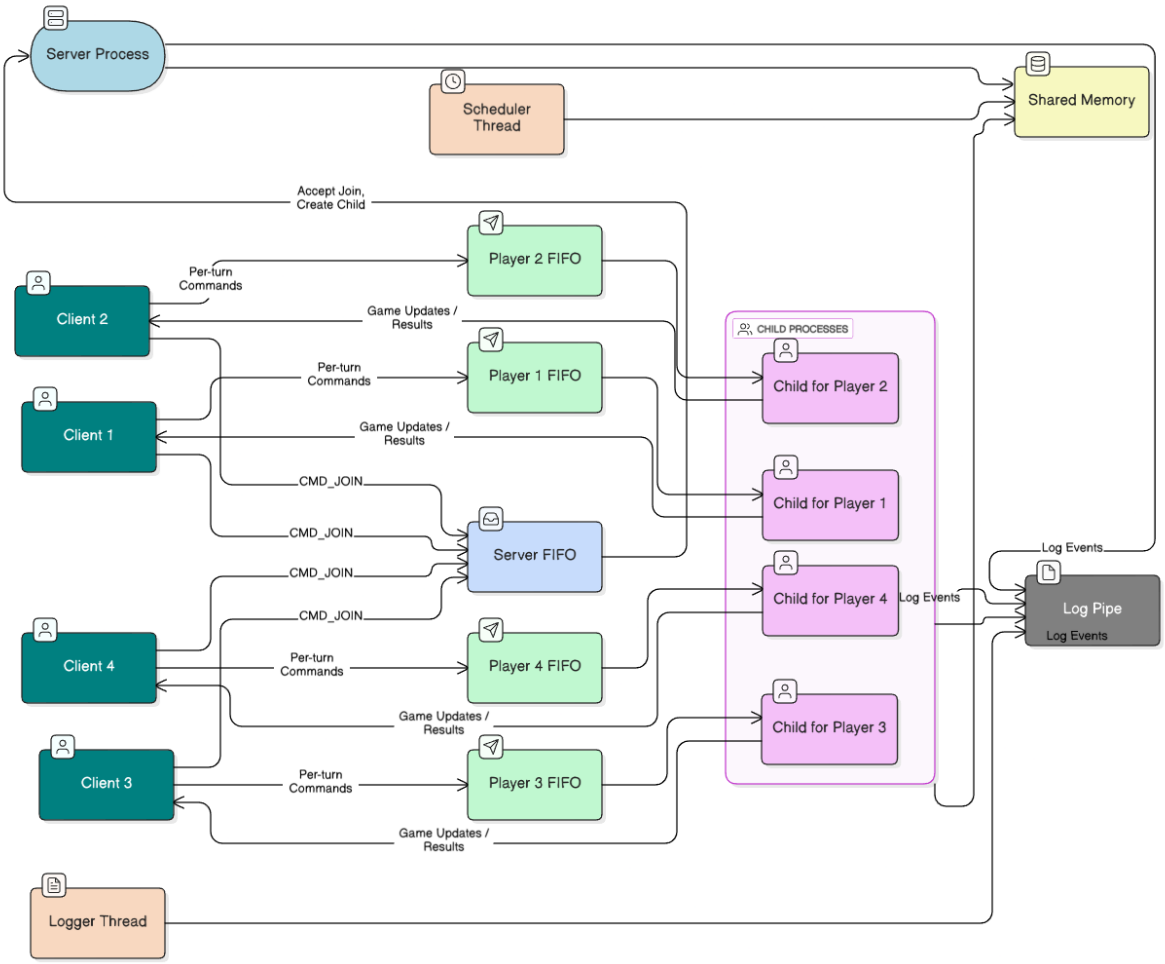


Figure 1 Hybrid Architecture Diagram

Figure 1 Hybrid Architecture Diagram shows the hybrid architecture of Tactical Token Race. The server runs as a parent process that creates a Scheduler thread and a Logger thread and forks one child process per connected player. All server processes and threads share a common **game_state_t** structure in POSIX shared memory. Clients communicate with the server via named pipes: a shared SERVER_FIFO for join requests and per-player FIFOs for per-

turn commands and game updates. All processes write log messages to a pipe consumed by the Logger thread, which serializes them into data/game.log.

IPC Mechanism and Shared Memory Layout

In this project, Inter-Process Communication (IPC) is required so the main controller process and multiple terminal processes can exchange information. Since each terminal runs as a separate process with its own unique PID, IPC ensures they communicate in a consistent and controlled manner.

IPC Mechanism Used

The project uses shared memory because it allows fast, direct data exchange between processes without needing to copy data through pipes or files. Shared memory lets all processes access the same memory region simultaneously. To prevent conflicts when multiple processes try to read or write at the same time, semaphores are used to synchronize access. This avoids race conditions and ensures data integrity.

Communication Flow

- The main process creates the shared memory and initializes the semaphore.
- Each child process attaches to the shared memory when it starts.
- Terminals write commands or updates into shared memory.
- The main process monitors the shared region and reacts to the commands.

Shared Memory Layout (Simplified)

Field	Description
active_terminal_count	Number of running terminals
last_command_pid	PID of the terminal that sent the latest command
command_buffer	Stores the most recent command
message_buffer	General message are

Synchronization Strategy

1.1. Process-Shared Mutex

- game_mutex is initialized with PTHREAD_PROCESS_SHARED.
- It protects all modifications to shared game state:
    - Joining/leaving players
    - Updating positions, tokens, winner
    - Changing players_connected, lobby_ready, target_players
    - Reading/updating leaderboard

1.2. Semaphores

1.2.1. Per-player semaphores turn_sems[]:

- Child blocks on its entry.
- Scheduler posts when it is that player's turn, or when the game ends and it needs to send final status.

1.2.2. Scheduler semaphore scheduler_sem:

- Child posts when it has finished processing its turn or when a disconnect occurs.
- Scheduler waits on this to know when it can safely choose the next player and modify turn state.

1.3. Critical Sections and Invariants

- All code sections that modify shared data are inside pthread_mutex_lock(&game->game_mutex) / unlock.
- Invariants maintained, e.g.:
  - 0 <= players_connected <= MAX_PLAYERS
  - winner_id == -1 except when a winner is declared
  - active[i] == 1 only if the player slot is valid and in use

Logger Design

Round Robin (RR) Scheduler

Persistence Strategy

Multi-Game Handling

The scheduler_thread() function runs an infinite outer loop. The reset process starts by clearing all players states such as positions, token flags, and all players are marked inactive. The lobby is then configured to it's initial state :

```
game->players_connected = 0;
game->target_players = MAX_PLAYERS;
game->lobby_ready = 0;
game->winner_id = -1;
game->game_running = 1;
```

This line of code :

```
pthread_mutex_unlock(&game->game_mutex);
break;
```

Unlocks the mutex so other threads can access shared memory. 'break' exits the current game session loop and returns to the scheduler's outer loop that waits for players in the new lobby. There are 2 ways in which a game could end and the session restarts :

i. When all players leave the game

```
// If everyone left, RESET
if (game->players_connected == 0) {
printf("[SCHEDULER] All players left. Resetting...\n");
// Reset Session
for(int i=0; i<MAX_PLAYERS; i++) {
game->positions[i] = 0;
game->token_flag[i] = 0;
game->active[i] = 0;
sem_init(&game->turn_sems[i], 1, 0);
}
game->players_connected = 0;
game->target_players = MAX_PLAYERS;
game->lobby_ready = 0;
game->winner_id = -1;
game->game_running = 1;

pthread_mutex_unlock(&game->game_mutex);
break; // Go back to lobby wait
}
```

This occurs when all players disconnect from the game, and the scheduler detects players_connected ==0. All game states will reset to initial values. 'Break' exits the inner game loop and returns to the outer loop.

ii. When a game names a winner

```
pthread_mutex_lock(&game->game_mutex);
for(int i=0; i<MAX_PLAYERS; i++) {
game->positions[i] = 0;
game->token_flag[i] = 0;
game->active[i] = 0;
sem_init(&game->turn_sems[i], 1, 0);
}
game->players_connected = 0;
game->target_players = MAX_PLAYERS;
```

```
game->lobby_ready = 0;
game->winner_id = -1;
game->game_running = 1;
printf("[SCHEDULER] Session Ended. Lobby Reset.\n");
pthread_mutex_unlock(&game->game_mutex);
break;}
```

A winner is named when a player reaches or exceeds the target score which is 30. The winner_id is set to that player's name

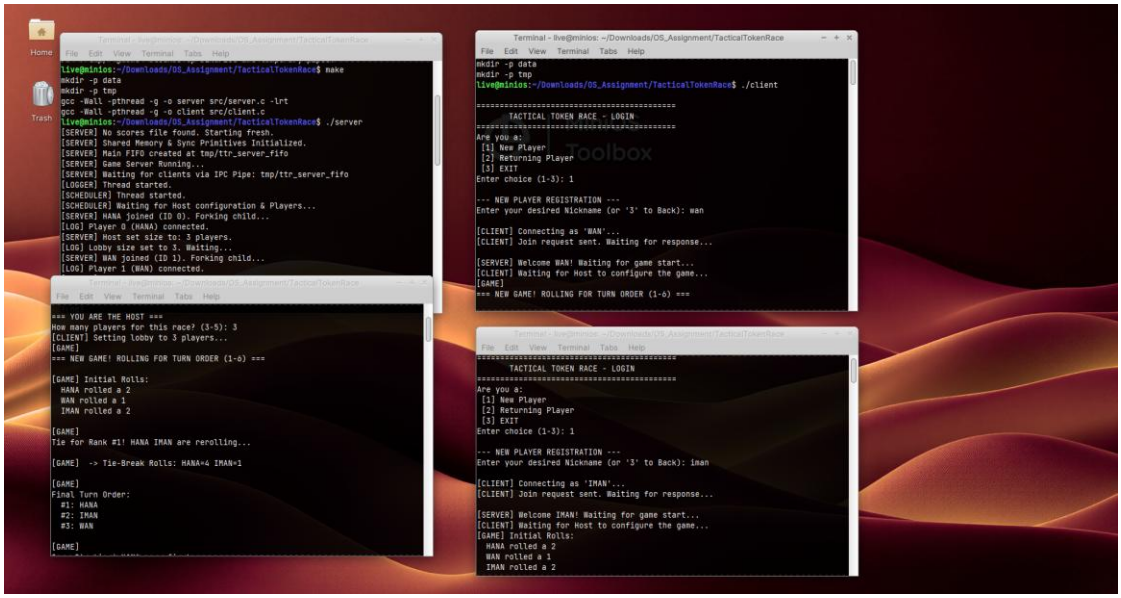
iii. Host leaves during setup

```
if (bytes <= 0) {
pthread_mutex_lock(&game->game_mutex);
printf("[SERVER] Host disconnected during setup (EOF).\n");
// Mark Host as gone
game->active[0] = 0;
game->players_connected--;
game->player_pids[0] = 0;
memset(game->player_names[0], 0, 32);
// Nuke the lobby for everyone else
printf("[SERVER] Aborting lobby. Kicking %d waiting players...\n", game->players_connected);
for(int i=1; i<MAX_PLAYERS; i++) {
if (game->active[i]) {
char target_fifo[64];
snprintf(target_fifo, sizeof(target_fifo), CLIENT_FIFO_TEMPLATE, game->player_pids[i]);
int t_fd = open(target_fifo, O_WRONLY | O_NONBLOCK);
if (t_fd != -1) {
ipc_msg_t kill_msg;
kill_msg.command = MSG_LOSE;
snprintf(kill_msg.payload, sizeof(kill_msg.payload), "Host disconnected. Lobby Closed.");
write(t_fd, &kill_msg, sizeof(kill_msg));
close(t_fd);}
game->active[i] = 0;
game->players_connected--;}}
```

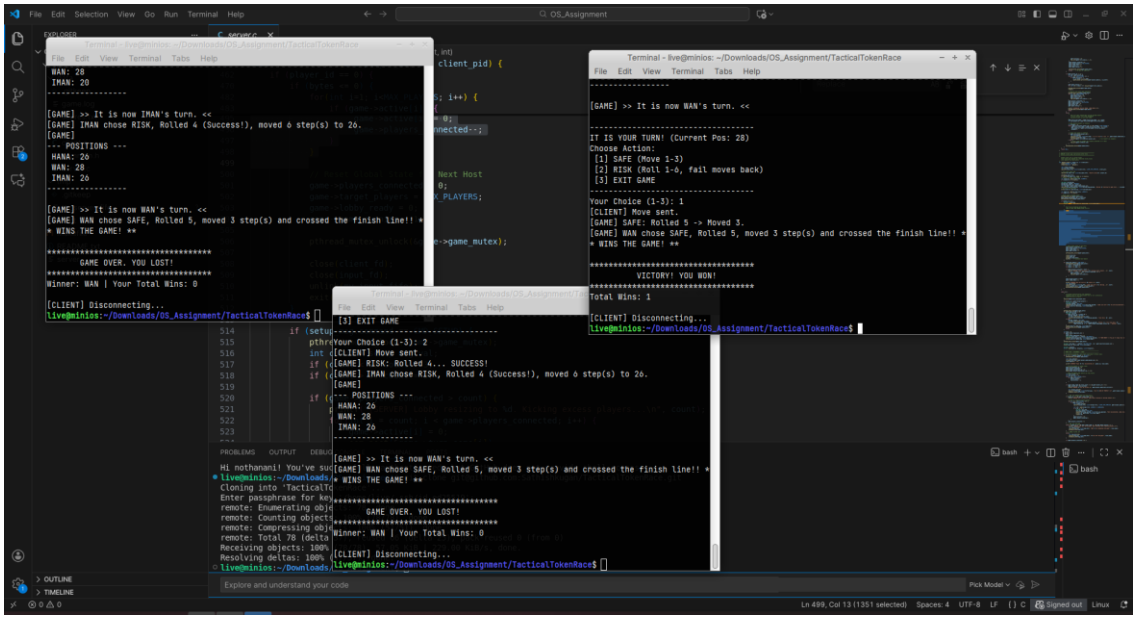
Testing Evidence

Gameplay screenshots +sample game.log

1. Starting a game session by initializing the server (top left), and registering the host and players



2. Gameplay and server



Screenshots

-each c;ient view, part of loger and persistent storage content