[NETWORKED ASCII "BATTLE GAME"]

[Assignment 3]

MARCH 18, 2025

[COMPSCI 3N03]

[Computer Networks and Security]

Assignment 3
Networked ASCII “Battle Game”

COMPSCI 3N03 Computer Networks and Security
McMaster University

Contents

1 Introduction to Socket Programming in C

Socket programming in the C language involves using operating system APIs to perform network communication between processes running on the same machine or across different machines. A **socket** represents one endpoint of a two-way communication link, and it is associated with an IP address and a port number.

1.1 TCP vs. UDP

The main transport-layer protocols are:

- **TCP (Transmission Control Protocol)**: Provides reliable, ordered delivery of data. This is typically used for applications where correctness and completeness of data are crucial (e.g., web servers, turn-based games).
- **UDP (User Datagram Protocol)**: Provides faster, connectionless communication with no guarantee of delivery or ordering. Commonly used for real-time scenarios like video streaming or some online games where occasional data loss is tolerable.

In this assignment, we focus on **TCP sockets** for a turn-based or small real-time game where reliability matters.

1.2 Socket Lifecycle in C (Server Side)

A typical TCP server in C follows these steps:

1. `socket()`:
    - Creates an *unbound* socket descriptor.
    - Typically called like:
      `int serverSock = socket(AF_INET, SOCK_STREAM, 0);`
2. `bind()`:
    - Associates the socket with a specific IP address and port.
    - Example:
      `bind(serverSock, (struct sockaddr*)&serverAddr, sizeof(serverAddr));`
3. `listen()`:
    - Marks the socket as a listening socket for incoming connections.
    - Example:
      `listen(serverSock, SOMAXCONN);`
4. `accept()`:
    - Blocks until a client attempts to connect, returning a *new* socket descriptor dedicated to that client.

- Example:
  `int clientSock = accept(serverSock, (struct sockaddr*)&clientAddr, &addrLen);`

5. **Communication** (`send()`, `recv()`):
   - Use the returned `clientSock` to communicate with the connected client. You can `fork()`, create a `pthread`, or use `select()` to handle multiple clients concurrently.

6. `close()`:
   - When you are finished with the socket, call `close()` to release system resources.

1.3 Socket Lifecycle in C (Client Side)

A typical TCP client in C follows these steps:

1. `socket()`:
   - Similar to the server, the client creates a socket descriptor.
   - Example:
     `int clientSock = socket(AF_INET, SOCK_STREAM, 0);`

2. `connect()`:
   - Attempts to connect to the server's IP address and port.
   - Example:
     `connect(clientSock, (struct sockaddr*)&serverAddr, sizeof(serverAddr));`

3. **Communication** (`send()`, `recv()`):
   - Once connected, you can send or receive data. This continues until either side closes the connection.

4. `close()`:
   - When communication is complete, call `close(clientSock)` to free up resources.

1.4 Blocking Calls and Concurrency

By default, `accept()`, `connect()`, `send()`, and `recv()` are *blocking* calls:

- **Blocking** means the function will not return until the operation has succeeded (or an error/timeout occurred).
- For a multi-client server, you can:
  1. Use **threads** (`pthread_create`) to handle each client in parallel.
  2. Use **I/O multiplexing** (e.g., `select()`, `poll()`) to handle multiple connections with one or few threads.

1.5 Why TCP for this Assignment?

Since our game requires each client to receive accurate and ordered game state updates (and each command to be delivered reliably), TCP is well-suited. If a packet is lost or arrives out of order, TCP will handle retransmissions and reordering for us. This ensures that all players share a consistent game world.

2 Goal of the Assignment

You will build a small, text-based networked game called the *"Battle Game"*. The objective is to:

1. Learn the basics of TCP socket programming.
2. Design and implement a lightweight *application-level protocol* for communication between the client(s) and the server.
3. Practice concurrency on the server via threads (or select/poll, if preferred).
4. Implement game logic: a 2D grid, movement, obstacles, and an attack mechanic.

By completing this assignment, you will have a working understanding of how to:

- Create client-server architectures.
- Exchange messages over TCP sockets.
- Synchronize shared state among multiple connected clients.
- Use GitHub to track contributions and collaborate as a group.

2.1 Option: Implementing in Python

Although the above sections discuss sockets in C, you may also choose to implement this assignment in **Python** if you prefer. We provide two skeleton files, `server.py` and `client.py`, which mirror the same structure and `TODO` tasks as the C version.

In Python, you can use the `socket` and `threading` modules, which behave analogously to `socket.h` and `pthread` in C.

3 Group Declaration

3.1 GitHub Classroom

This is an **assignment link**, which you must click to accept this project on GitHub Classroom.

`https://classroom.github.com/a/GT67vYVA`

Clicking the link will guide you to accept the assignment, form or join a team, and set up your repository. If you encounter any difficulties during sign-up or group formation, please contact:

- Leonard Chen: `chenl299@mcmaster.ca`

When prompted:

1. Find your name and link your GitHub account to your name.
2. Form a team or join an existing team (the group size is up to four, unless otherwise specified).
3. Once your team is set, each member can clone the repository locally and begin working on the code.

The repository you receive will contain:

- `server.c` and `client.c` templates (with `TODO` sections) for a C implementation.
- `server.py` and `client.py` templates (with `TODO` sections) for an equivalent Python implementation.
- A `README.md` describing how to set up and compile (or run) your code.

3.2 A2L

On the **A2L** platform, you must also:

- Declare your group members, ensuring *the same group composition* as on GitHub Classroom.
- Upload the required PDF deliverable (the protocol explanation and diagrams) by the specified due date.

Important: The group members on GitHub Classroom *must match* those declared on A2L, so that grading is consistent.

4 Deliverables

1. **GitHub Repository** (one repository per group):
    - Completed `server.c` and `client.c` **or** `server.py` and `client.py` (depending on your chosen language).
    - A `README.md` (with game logic description, compilation or run instructions, protocol overview, etc.).
    - A `README.md` (with game logic description, compilation instructions, protocol overview, etc.).

   - Contributions should be traceable via **git commits** from each member.

2. **PDF Document (uploaded to A2L)**:
   - Explains your final protocol (commands, message flow).
   - Includes diagrams or sequence charts illustrating how the server and client communicate.
   - Make sure to note your group members' names and student IDs in the PDF.

5 Requirements of the Assignment

1. **Server Requirements**:
   - Create a TCP socket, bind it to a specified port, and listen for connections.
   - Accept up to 4 clients concurrently.
   - Maintain a 2D `GRID` with obstacles (`#`) and player positions (`A`, `B`, `C`, `D`).
   - Implement concurrency (e.g., `pthreads`) to handle each client's commands.
   - Update the game state (e.g., movement, attacks) and *broadcast* the new state to all connected clients.

2. **Client Requirements**:
   - Create a TCP socket and `connect()` to the server.
   - Continuously read user commands (e.g., `MOVE`, `ATTACK`, `QUIT`) and send them to the server.
   - Receive updated game states and display them in an *ASCII grid* format.
   - Gracefully handle disconnection or `QUIT` commands.

3. **Protocol Design**:
   - At minimum, handle textual commands (e.g., `"MOVE UP"`, `"ATTACK"`, `"QUIT"`).
   - Server responses might be in the form of `"STATE\n"` plus the 2D ASCII grid, or other text-based structures.
   - Document all messages in your code or PDF (i.e., which commands are recognized, how the server responds).

4. **Basic Game Logic**:
   - Players can move (up, down, left, right) within the grid, but cannot pass through obstacles (`#`) or move out of grid bounds.
   - `ATTACK` can reduce another player's health if they are in an adjacent cell.
   - `QUIT` disconnects the client; server updates the state accordingly.

5. **Additional Features**:
   - Up to 1% of your overall mark will be allocated to these advanced or creative mechanics within the 10% total.
   - Possible ideas include (but are not limited to):
     (a) **Turn-based Mechanics**: Only one player can move/attack at a time.
     (b) **Advanced Attacks or Spells**: Ranged attacks, diagonal attacks, or special abilities (e.g., "FIREBALL" affecting a 2x2 area).
     (c) **Multiple Rooms/Lobbies**: Let players join separate rooms, each with its own grid.
     (d) **Chat Messages**: A command like `SAY <message>` that broadcasts within the same game instance.
     (e) **Item Pickup and Inventory**: Players can pick up items for bonuses (e.g., extra HP, speed, or damage).
   - Document these features in your `README.md` and PDF (protocol changes, new commands, etc.).

6. **Documentation**:
   - `README.md` in the GitHub repository must include:
     - **Compilation Steps** (for C) or **Run Instructions** (for Python).
     - **Quickstart Guide**: instructions on running the server (e.g., `./server <PORT>` or `python server.py <PORT>`) and connecting with one or more clients (e.g., `./client <SERVER_IP> <PORT>` or `python client.py <SERVER_IP> <PORT>`).
     - **Gameplay / Command Overview**: explain the basic commands (`MOVE`, `ATTACK`, `QUIT`, etc.) and any *additional features* (extra commands, turn-based rules, etc.).
   - **PDF Document** submitted on A2L must include:
     - A **Detailed Explanation of Your Protocol**
     - A **Sequence Diagram or Flow Chart** showing how the client and server communicate.

7. **Compiler Restriction (C) or Environment (Python)**:
   - For C implementations, you must use `gcc` or `clang`. Test on a standard Unix-like environment.
   - For Python implementations, use Python 3 (`>=3.10`) and ensure your code runs without additional libraries.

6 Example Walkthrough of a Sample Project

Below is an illustrative example of how a final, working game session might proceed. Note that *your implementation details can differ* in appearance or logic, as long as the requirements are met. Additionally, the **client displays** shown here reflect an example approach to printing the ASCII grid and status information.

6.1 Server Startup

```
$ ./server 12345
Server listening on port 12345...
```

The server is now ready to accept up to 4 clients.

6.2 First Client Joins

```
$ ./client 127.0.0.1 12345
Connected to server 127.0.0.1:12345

--- GAME STATE ---
A....
..#..
..#..
.....
.....

Players:
 Player A: HP=100 Pos=(0,0)

Enter command (MOVE/ATTACK/QUIT):
```

Explanation:

- The client connected successfully to the server at `127.0.0.1:12345`.
- The server assigned this client the label `A` and placed them at `(0,0)`.
- The ASCII grid is 5 rows by 5 columns, with some obstacles (`#`) at positions `(1,2)` and `(2,2)` (for example).

6.3 Second Client Joins

In another terminal:

```
$ ./client 127.0.0.1 12345
Connected to server 127.0.0.1:12345

--- GAME STATE ---
A....
..#..
B.#..
.....
.....

Players:
 Player A: HP=100 Pos=(0,0)
 Player B: HP=100 Pos=(2,0)

Enter command (MOVE/ATTACK/QUIT):
```

Explanation:

- The second player is labeled `B` and placed at `(2,0)` by the server.
- Both clients now see a grid containing `A` and `B`, along with obstacles in the middle rows.

6.4 Moving Around

Player A enters:

```
Enter command (MOVE/ATTACK/QUIT): MOVE DOWN
```

The server processes `MOVE DOWN` for Player A and updates the position from `(0,0)` to `(1,0)` (assuming no obstacle is there). Then it broadcasts the new state to *all* players.
Client A sees:

```
--- GAME STATE ---
.....
A.#..
B.#..
.....
.....

Players:
 Player A: HP=100 Pos=(1,0)
 Player B: HP=100 Pos=(2,0)
```

Client B sees a similar output:

```
--- GAME STATE ---
.....
A.#..
B.#..
.....
.....

Players:
 Player A: HP=100 Pos=(1,0)
 Player B: HP=100 Pos=(2,0)
```

Explanation:

- The grid now shows `A` in row 1, `B` in row 2.
- Obstacles remain at `(1,2)` and `(2,2)` (marked as `#`).

6.5 Attacking

If Player A moves adjacent to Player B, A might type:

```
ATTACK
```

If the server's logic detects that B is in an adjacent cell, it deducts HP (say 20). Then all clients might see something like:

```
--- GAME STATE ---
.....
A.#..
B.#..
.....
.....

Players:
 Player A: HP=100 Pos=(1,0)
 Player B: HP=80  Pos=(2,0)
```

Explanation:

- B's HP is reduced from 100 to 80 because of the adjacent attack.
- If B's HP drops to 0 or below, B is considered defeated or inactive (depending on your implementation).

6.6 Quitting

Whenever a player types:

```
QUIT
```

the client disconnects, and the server marks them as inactive. The remaining players still see a grid without that player symbol. If all clients quit, the server will remain open, waiting for new connections (unless you design it to shut down after all players leave).

Summary of Client Display:

- After every valid command from *any* player, the server sends an updated game state to *all connected* clients.
- Each client can immediately see changes in positions, obstacles, and HP.
- The exact format of the ASCII grid and player info is up to your *protocol design*, but it must be consistent and clearly reflect the current state.

7 Rubric and Assessment

Your final grade (10%) is allocated as follows:

Category	**Percentage**
Socket Implementation & Compilation (Correctly compiling and running, creating sockets, binding, listening, connecting, concurrency, error handling)	3%
Protocol Design (Clear commands/responses, proper handling of messages, protocol documentation)	2%
Basic Game Logic (Movement, obstacle handling, attacks, HP mechanic, correctness of updates)	2%
Additional Features (Advanced game mechanics beyond the basics, e.g. turn-based approach, multiple rooms, special abilities)	1%
Documentation and README (Instructions for compilation, usage, clarity of readme, PDF diagrams)	1%
Git Usage & Collaboration (Commit history, evidence of each member's contributions)	1%

Total = 10%

- Note that *up to* 1% of your grade is awarded for any *additional Features* beyond the basic game logic.

- It is possible to achieve a full 10% by effectively completing all categories, including some or all of the advanced mechanics.
- If you do not implement extra features, that portion (1%) will go unclaimed, but you can still demonstrate a working solution for the rest of the categories.

8 Additional Notes

- For a well-commented, fully working solution, you should anticipate around **500 lines of code in** `server.c` and about **200 lines in** `client.c` (if in C). Python solutions may differ, but keep similar logic and detail.
- A recorded tutorial on socket programming in C is available at:

  `https://www.youtube.com/watch?v=ehUNwEHkABQ`

  It is recommended watching it to review the fundamentals of network programming, including socket creation, binding, and listening.
- Additional supplementary materials regarding socket programming can be found on A2L.
- Always handle potential socket errors (e.g., check return values of `bind()`, `listen()`, `accept()`, etc.).
- Test your program using multiple terminals on the same machine (`127.0.0.1`) and, if possible, on different machines in a LAN environment.
- Ensure all group members frequently `push` and `commit` to track contributions on GitHub.

The End.