CSS 432 Term Project: Zomboid2D

Maryam M, 2024

Gameplay

Zomboid2D is a simple top-down zombie-shooter game. When a player creates a game, the server handles map generation and spawning hostile mobs. Players can attack mobs with ranged weapons, while mobs deal melee damage to players.
Points are awarded for mob kills. When a player dies, their points reset, but the leaderboard retains their high score. If a player rejoins the game, their leaderboard score updates only if they exceed their previous high score. Once no players are left alive, the game ends.

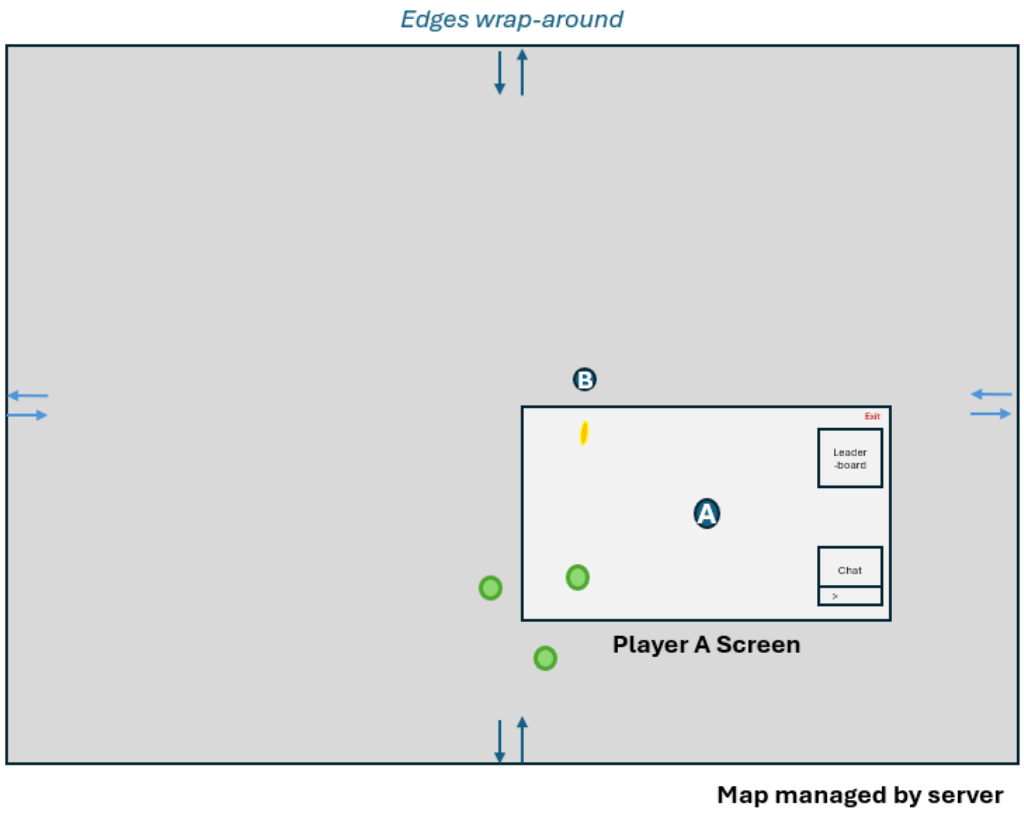


Figure: Zomboid2D Mockup

The NetworkAPI can support the following features:

- Player registration/game connection, or leaving/disconnecting.
- Mob/player movement/location and actions.
- Chat messaging.
- Leaderboard system.

Network Protocol

The protocol uses a **client-server architecture** because it simplifies game management. Having a server manage shared resources such as map generation, mob spawning, status management, player movement/action verification is simpler with a server rather than over P2P.

The server stores a **TCP and** a **UDP** connection for each client.

TCP/UDP Uses:

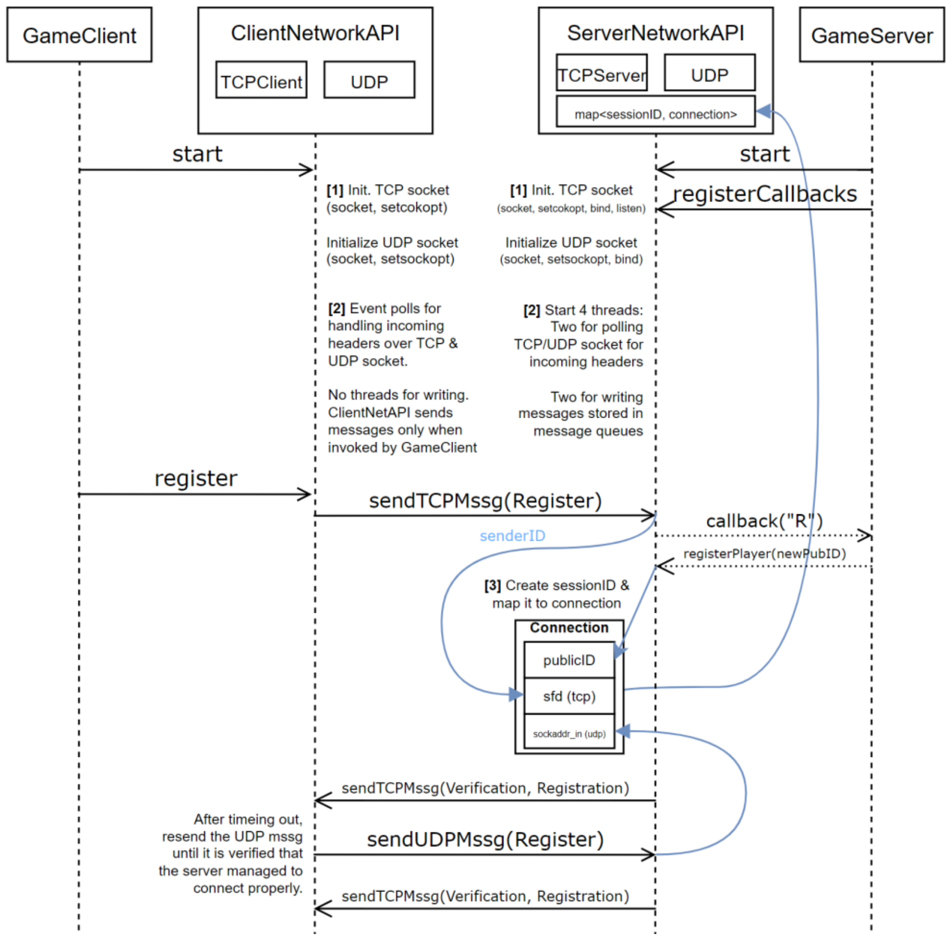
TCP ensures reliable message delivery, so it is used for important messages such as:

- Game setup, including player registration, creating/joining games
- Important events like player/mob deaths and score updates for the leaderboard
- Mob spawning
- Chat messages

UDP is used for time sensitive updates, such as:

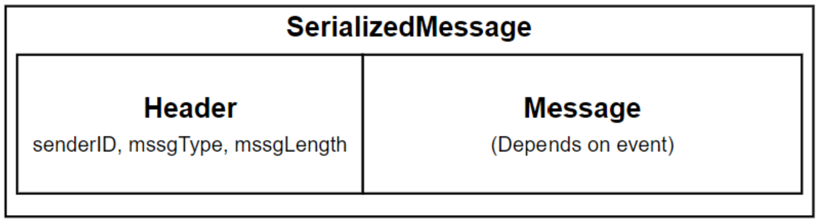
- Player/Mob movements and actions
- Damage updates

Connection Setup:


GameClient
ClientNetworkAPI
TCPClient
UDP
ServerNetworkAPI
TCPServer
UDP
map<sessionID, connection>
GameServer
start
start
registerCallbacks
[1] Init. TCP socket (socket, setcokopt)
Initialize UDP socket (socket, setsockopt)
[1] Init. TCP socket (socket, setcokopt, bind, listen)
Initialize UDP socket (socket, setsockopt, bind)
[2] Event polls for handling incoming headers over TCP & UDP socket.
No threads for writing. ClientNetAPI sends messages only when invoked by GameClient
[2] Start 4 threads: Two for polling TCP/UDP socket for incoming headers
Two for writing messages stored in message queues
register
sendTCPMssg(Register)
callback("R")
senderID
registerPlayer(newPubID)
[3] Create sessionID & map it to connection
Connection
publicID
sfd (tcp)
sockaddr_in (udp)
sendTCPMssg(Verification, Registration)
After timeing out, resend the UDP mssg until it is verified that the server managed to connect properly.
sendUDPMssg(Register)
sendTCPMssg(Verification, Registration)

Message Protocol:

When the NetworkAPI sends a message, it must be one of the provided formats (see *messages.h* for the message structures). *Every* message must be preceded by a header, since the socket-readings poll for the header size, and determine how the remaining will be handled.



Both the Header and the Message will be packed into a SerializedMessage struct, which provides de/serialization functions for sending/receiving the packet over the network.

For example: A player sends a Move message with mssgType="M", and the server broadcasts to all other players a Move message with mssgType="L" (for location).

Suggestions

- When it comes to projectiles, the server should only broadcast the projectile's original spawn point and rotation. Since the path is pre-set, the GameClient should just render its movement.
- Collision-detection (such as projectiles hitting mobs, and calculating damage & cause) should be done on the server-side to avoid inconsistency between clients.

API Usage:

Receiving: The NetworkAPI is inspired by event-based design and the observer pattern (especially the server-side, which can't afford to poll each client). In the initialization stage, the Game modules need to register callback methods for each event.

For example: Say the GameServer wants to "observe" a "registration event". Thus, it provides ServerNetAPI a callback for its own RegisterPlayer(...) function, associated with EventCode.Register.

A current flaw in the design is that the NetworkAPI is "aware" of the parameters of each callback, so the GameModules need to adhere to the API's format.

Sending: Additionally, GameModules can invoke send commands on their APIs:

ClientNetworkAPI:	**ServerNetworkAPI:**
• registerPlayer • getGameList • connectToGame(gamename) • sendMove(xCoord, yCoord) • sendChat(message)	• broadcastEvent(EventCode, args...) - *Same thing where args for each event are pre-determined by ServerNetworkAPI.* Methods like sendGameList are automatically invoked by the NetworkAPI on connection.

Code Compilation & Execution

Files you need

Both will need files under the “core” directory. Only the server-side will need the files under the “server” directory, and only the client-side will need the files under the “client” directory.

Compilation

Use the linux command to compile the NetworkAPI (if applicable, replace “client” with “server”):

```
g++  src/core/*.cpp src/client/* .cpp -o client
```

Execution

Unfortunately, the GameModules were not completed, and the NetworkAPIs are interfaces that cannot be executed stand-alone. However, NetworkAPI provides the means of integrating into other modules (callbacks for incoming messages, invoking public methods for sending messages), and documentation is pending to make it easier to do so.