Documentación Técnica

Este documento describe el diseño e implementación de un juego multijugador 2D por red, inspirado en Counter Strike, desarrollado como trabajo práctico para la facultad. El sistema sigue una arquitectura cliente-servidor, donde toda la lógica del juego se simula en el servidor de forma centralizada y determinista.

Los clientes pueden crear o unirse a partidas, seleccionar personajes, ejecutar acciones y recibir actualizaciones del estado del juego en tiempo real. La comunicación entre cliente y servidor se realiza mediante un protocolo binario eficiente, que utiliza estructuras bien definidas para representar mensajes y respuestas.

A continuación se detallan los componentes principales del sistema: lógica del juego, estructura del cliente, funcionamiento del servidor y especificaciones del protocolo de red.

Se incluyen diagramas de arquitectura y flujo en la carpeta “documents/diagrams” como material complementario.

Lógica del juego

En la clase Game se concentra la lógica principal del juego. Se encarga de ejecutar las acciones de los jugadores, gestionar las físicas, manejar las colisiones, el cambio de fase y finalización de rondas y del juego. Para ello, se apoya principalmente en las clases Player, GameMap y Team.

El manejo de disparos varía según el tipo de arma (struct Weapon en structures.h), para lo cual se definen valores específicos que determinan, entre otras cosas, si el arma dispara en ráfaga o cuántas balas se disparan por tiro. Esta lógica se implementa en las clases Game y Player.

Las colisiones, tanto entre jugadores como entre balas y paredes o jugadores, se gestionan dentro de Game. En el caso de los disparos, los impactos se registran en una cola que almacena información sobre el punto de origen, el punto de impacto y el objeto con el que se produjo la colisión. Las colisiones con el entorno se resuelven en GameMap.

El estado general de cada equipo se administra desde la clase Team, administrando cuestiones que involucran a todos los jugadores de dicho equipo, como el rol actual y el cambio de rol, quién porta la spike en caso de ser atacantes entre otros.

En cuanto a la tienda utilizada durante la fase de compra, se encuentra encapsulada en una clase estática llamada Store, desde la cual se obtienen los valores de los objetos que el jugador desea adquirir. A partir de esto, se valida si la compra es posible y se ejecuta la transacción.

Métodos importantes

El método público execute recibe una acción junto con el jugador que desea realizarla, y se encarga de ejecutarla. A través de este método, los jugadores pueden disparar, interactuar con la bomba o moverse.

El método público update recibe un delta time del servidor y actualiza tanto las entidades del juego como el tiempo transcurrido. Su función principal es actualizar las fases del juego y la posición de los jugadores, aplicando físicas básicas de movimiento.

El método público getState devuelve un StateGame, un struct que organiza el estado actual del juego, incluyendo la información de los jugadores, la bomba, las armas en el suelo y las balas disparadas. Las balas son instantáneas: se comportan como un láser, de modo que al momento del disparo ya se conoce el punto de impacto, por eso no se actualizan con el tiempo.

(ver diagrams/logic_game_diagram.png)

Cliente

La lógica de inicialización del menú y herramientas gráficas tanto para el mismo menú como para el juego más adelante se concentra en Launcher. Esto es hasta que el usuario se conecta a un servidor, en ese momento la lógica del cliente pasa a ser manejada por Client. Para no mezclar los apartados gráficos del menú y juego, el primero hecho con Qt y el otro con SDL. Por otro lado la lógica de Client está separada en otras dos clases que son MenuClient y GameClient. Es importante mencionar que el apartado gráfico del menú en sí es manejado por las distintas vistas (MainView, SearchPartyView, PartyView, etc), y el manejo de los eventos en las mismas lo maneja MenuController.

MenuClient se encarga de avisarle a MenuController de cualquier información recibida por parte del servidor y recibir eventos que hayan ocurrido en las vistas de MenuController que requieran mandar algún mensaje al servidor. Por otro lado, el apartado gráfico del juego es manejado por la clase GameView, y los eventos que ocurren dentro del mismo los maneja GameController. El GameClient entonces se encarga del loop principal del juego, de recibir mensajes del servidor y notificar a GameController de los mismos y también de recibir acciones del GameController que requieran enviar un mensaje al servidor.

Métodos importantes

El método GameView::update se encarga de actualizar los elementos visuales según la información que tiene del estado del juego.

El método GameController::updateGameState se encarga de avisarle a GameView los cambios de estados que fueron enviados por el servidor y los eventos que ocurrieron al cambiar de un estado a otro, por ejemplo, si hubo un cambio de fase, si hubo una muerte, entre otros.

El método GameController::processEvents se encarga de procesar inputs del usuario dentro del juego.

Los métodos listenTo de MenuController se encargan de procesar eventos ocurridos dentro de las vistas del menu.

El método buildLayout dentro de cada vista del menu es el que se encarga de construir ese vista.

Server

El servidor se estructura a partir de la clase *Server*, la cual crea un objeto *Admin* y lanza un hilo dedicado a aceptar conexiones entrantes. A medida que los clientes se conectan, el hilo aceptador los registra y delega al *Admin* su gestión. Este último mantiene un registro centralizado de todos los clientes activos, así como de las partidas en estado de lobby o juego. El servidor permanece en ejecución hasta que se recibe un 'q' por entrada estándar, momento en el que inicia el cierre controlado de todos los hilos y recursos.

Cada cliente conectado se gestiona mediante una instancia de *Client*, que lanza tres hilos: uno *Receiver*, uno *Sender*, y otro *Menu*. El *Receiver* recibe mensajes del cliente y los encola para procesamiento; el *Sender* toma las respuestas generadas y las envía; y el *Menu* maneja las interacciones del cliente mientras este se encuentra fuera de una partida.

Cuando un cliente crea una partida, su hilo *Menu* se detiene y se lanza un nuevo hilo *Match*, que comienza en modo lobby. El cliente se asocia a esta instancia, la cual mantiene referencias a todos los clientes involucrados. Lo mismo ocurre cuando otro cliente se une a la partida: se detiene su *Menu* y se lo incorpora al *Match*. Si un cliente abandona el lobby, se lo desvincula del *Match* y se relanza su *Menu*. Una vez que se inicia la partida, el *Match* cambia de estado e ingresa al ciclo de juego activo.

Mientras la partida se encuentra en lobby, se ejecuta un *LobbyLoop* que monitorea los cambios en la composición del lobby (como la entrada o salida de jugadores). Cada vez que ocurre un cambio, se envía un estado actualizado del lobby a todos los clientes conectados para que reflejen la nueva información.

Previo al inicio del juego, el servidor envía a los clientes un mensaje *INITIAL_DATA*, que contiene toda la información necesaria para sincronizar la simulación: los nombres de los jugadores, el mapa a utilizar y diversas constantes del juego. Esta preparación asegura que todos los clientes arranquen la partida desde un mismo estado inicial coherente.

Durante la ejecución del juego, el *Match* entra en el *GameLoop*, un ciclo basado en ticks, con frecuencia (*tickrate*) y cantidad máxima de eventos por tick configurables. En cada iteración del bucle se calcula el delta de tiempo desde el último tick, y se utiliza una instancia de la clase *Game* para simular la lógica del juego, incluyendo movimiento, disparos, físicas, colisiones, condiciones de victoria y otras mecánicas. Esta simulación es puramente determinista y centralizada en el servidor, garantizando coherencia entre los jugadores.

La comunicación entre los hilos *Sender*, *Receiver*, *Menu* y *Match* se realiza mediante colas *thread-safe*. Los *Receiver* escriben los mensajes entrantes en colas que son leídas por *Menu* o *Match*. En el caso de los *Match*, todos los *Receiver* de los clientes que participan comparten una cola común, lo que permite a la instancia de *Match* procesar centralizadamente todas las acciones de los jugadores. Por su parte, los *Sender* reciben desde una cola de salida individual por cliente y envían los datos al socket correspondiente.

El objeto *Admin* se encarga de la coordinación general del sistema: crea y destruye clientes, menús y partidas, garantiza la integridad de los recursos y permite un cierre ordenado del servidor.

(ver diagrams/server.png)

Protocolo

El protocolo de red que utilizamos está diseñado para mantener una comunicación binaria eficiente y estructurada entre el cliente y el servidor. Se basa en dos estructuras principales: *Message* (enviado por el cliente) y *Response* (enviado por el servidor). Proporciona una interfaz simple pero flexible, que abstrae los detalles de serialización y deserialización según el tipo de mensaje transmitido.
Desde el lado del servidor, los mensajes entrantes se reciben mediante *recv_message*, y las respuestas se envían con *send_response*. Por su parte, el cliente utiliza funciones *send_** (como *send_create*, *send_action*, etc.) para enviar mensajes, y *recv_response* para recibir las respuestas del servidor.

Tipos de mensajes implementados

Cliente → Servidor
NAME, CREATE, JOIN, LIST, ACTION, LEAVE, START, DISCONNECT
Servidor → Cliente
STATE_LOBBY, LOBBY_READY, NOT_LOBBY_READY, INITIAL_DATA, STATE, FINISH
Además de respuestas simples que incluyen un resultado (OK o FAIL) y un mensaje descriptivo.

Serialización y formato binario

Cada tipo de *Message* o *Response* tiene un formato de serialización propio y está respaldado por funciones específicas que transforman la información estructurada en datos binarios para su transmisión.

Las *Response* del servidor serializan tipo, resultado, menaje y pueden incluir estructuras complejas como:

- *LobbyList*: lista de nombres de partidas en espera
- *StateLobby*: jugadores en el lobby actual
- *InitialData*: información del mapa, jugadores, armas, tienda y tiempos
- *StateGame*: entidades del juego, disparos activos, rondas y fase actual

Ejemplo de serialización de un nombre:

1. 1 byte para el tipo de mensaje (0x00)
2. 2 bytes (big-endian) con la longitud del nombre
3. n bytes correspondientes al contenido del string (en UTF-8)

Ejemplo de flujo:

1. El cliente se conecta y envía su nombre (NAME):

- 1 byte → tipo de mensaje: NAME (valor fijo)
- 2 bytes → largo del string del nombre (en big-endian)
- n bytes → string del nombre del cliente

2. El cliente crea un lobby (CREATE):

- 1 byte → tipo de mensaje: CREATE
- 2 bytes → largo del nombre del lobby
- n bytes → nombre del lobby

3. El servidor responde con *STATE_LOBBY*:

- 1 byte → tipo de respuesta: STATE_LOBBY
- 1 byte → resultado (0 exito, 1 fallo)
- 2 bytes → cantidad de jugadores (vector size)
- Por cada jugador:
    - 2 bytes → largo del nombre
    - n bytes → string del nombre
- 2 bytes → largo del mensaje informativo
- k bytes → string del mensaje

4. El cliente envía una acción (ACTION):

- 1 byte → tipo de mensaje: ACTION
- 1 byte → subtipo de acción (por ejemplo: MOVE)
- Luego según el subtipo, por ejemplo:
    - MOVE:
        - 4 bytes → entero vx (2 bytes) y vy (2 bytes)

(ver diagrams/crear_salir_lobby.png, diagrams/entrar_salir.png y diagrams/game.png y diagrams/flujo_juego.png)