Manual Técnico - Counter Strike 2D

Índice

1. Introducción

El presente documento describe la arquitectura, diseño e implementación técnica de una recreación 2D multijugador del juego Counter-Strike, desarrollada como trabajo práctico para la materia Taller de Programación I (75.42) de la FIUBA.

El proyecto fue construido bajo una arquitectura cliente-servidor, con soporte para múltiples partidas simultáneas y comunicación mediante un protocolo binario personalizado. El código fue desarrollado en C++20, utilizando librerías como SDL2 para el renderizado gráfico y Qt para la interfaz de usuario, respetando el estándar POSIX 2008 y ejecutándose en sistemas basados en Ubuntu 24.04.

El manual se enfoca en describir los componentes más relevantes del proyecto desde el punto de vista técnico, destacando:

- La estructura modular del servidor y del cliente.
- El diseño del protocolo binario.
- El manejo multithread y sincronización.
  Los diagramas de clases, secuencia y de hilos más importantes.
- Las principales decisiones de diseño orientadas a la mantenibilidad y extensibilidad del código.

2. Arquitectura general del sistema

El sistema fue diseñado bajo una arquitectura cliente-servidor modular, que permite múltiples partidas en simultáneo y una clara separación de responsabilidades entre los distintos componentes del proyecto. Está compuesto por dos aplicaciones independientes:

- **Cliente**: se encarga de la interfaz gráfica, la captura de entrada del jugador, el renderizado del mapa y la visualización de la partida en tiempo real. Además, procesa los snapshots enviados periódicamente por el servidor para mantener actualizada la representación local del juego.

- **Servidor**: controla toda la lógica del juego, administra el estado global de cada partida y procesa las acciones enviadas por los clientes. Soporta múltiples partidas concurrentes, cada una gestionada de forma independiente. Por cada cliente conectado, se lanza un hilo dedicado que gestiona la comunicación con ese jugador.

Ambos componentes se comunican utilizando sockets TCP bloqueantes y un protocolo binario personalizado. La estructura del protocolo permite interpretar los mensajes enviados por los clientes y ejecutar las acciones correspondientes en el servidor.

Todos los parámetros configurables del juego (como la cantidad de vida, frecuencia de disparo, precios de las armas, entre otros) se definen en archivos YAML, permitiendo ajustes rápidos sin necesidad de recompilar el código.

3. Componentes principales

3.1 Cliente

El cliente es responsable de renderizar el juego, capturar la interacción del usuario y mantener sincronizada su visión local con el estado del juego proporcionado por el servidor. La clase principal que organiza y coordina estos elementos es GameClient.

Clase GameClient

Es el punto de entrada principal del cliente. Se encarga de:

- Iniciar el renderizador (GameRenderer) con el mapa y el ID del jugador.
- Lanzar un hilo secundario (DataReceiver) que escucha continuamente los snapshots enviados por el servidor.
- Procesar los eventos de entrada del usuario (teclado y mouse) mediante la clase InputHandler.
- Aplicar los snapshots recibidos a la lógica visual del cliente.
- Controlar el bucle principal del juego en tiempo real, manteniendo una frecuencia de actualización de 30 FPS.
- Renderizar el estado del juego (jugadores, mapa, HUD, menú de compras) a partir de los datos procesados.

Componentes utilizados por GameClient

- Protocol& protocol: permite el envío y recepción de datos con el servidor.
- InputHandler: interpreta la entrada del usuario y determina las acciones.

- DataReceiver: hilo separado que se mantiene recibiendo snapshots del servidor.
- QueueFixed<Snapshot>: cola fija para almacenar snapshots entrantes de forma segura.
- GameRenderer: responsable del renderizado de la pantalla del juego, interfaz, jugadores y mapa.

Flujo general

1. El cliente recibe un PreSnapshot desde el servidor, que contiene el mapa y el ID del jugador.
2. Se inicializa el GameRenderer con esta información.
3. Se lanza un hilo DataReceiver que escucha permanentemente los Snapshot.
4. En cada iteración del loop principal:
    - Se captura el input del jugador.
    - Se actualiza la lógica del renderizado en base al último snapshot disponible.
    - Se actualiza la interfaz si el jugador se encuentra en el menú de compras.
    - Se renderiza la pantalla.
5. Al cerrar el juego, se detiene y une el hilo receptor de datos.

3.2 Servidor

El servidor es responsable de gestionar toda la lógica del juego, coordinar la interacción entre múltiples jugadores y mantener actualizada la información que luego es enviada a los clientes. Está compuesto por cuatro clases clave:

Protocol

La clase Protocol centraliza la lógica de comunicación entre el servidor y todos los clientes conectados. Se encarga de:

- Aceptar nuevas conexiones entrantes mediante un Socket de escucha (acceptorSocket).

- Crear un ClientHandler por cada cliente conectado, ejecutándolo en su propio hilo.

- Mantener una requestQueue donde se encolan los mensajes entrantes.

- Asociar cada opcode recibido con un handler específico mediante un requestMapper, delegando la interpretación a ServerLobbyProtocol, GameLobbyProtocol o InGameProtocol según el estado de la partida.

- Enviar snapshots, pre-snapshots, y DTOs a los clientes utilizando el ClientHandler correspondiente.

ClientHandler

La clase ClientHandler representa un cliente conectado y corre en un hilo independiente. Sus responsabilidades incluyen:

- Leer continuamente del socket los opcodes enviados por el cliente y despachar el mensaje al handler adecuado según el estado del juego (lobby, game lobby o in-game).

- Transformar los datos recibidos en Request y encolarlos para su posterior procesamiento por el servidor.

- Enviar respuestas al cliente (snapshots, estados de conexión, lobby, etc.) utilizando la clase Sender.

- Detectar desconexiones y encolarlas como eventos según el contexto actual del jugador.

GameMonitor

GameMonitor representa la lógica interna de una partida. Cada partida tiene su propia instancia, ejecutándose en un hilo separado. Se encarga de:

- Actualizar el estado del juego en tiempo real (advance(time)), incluyendo movimiento, ataques, compras, plantado y desactivación de bomba, etc.

- Administrar la lógica de rondas y el cambio de equipos al finalizar la mitad del juego.

- Enviar periódicamente snapshots con el estado actualizado a los jugadores mediante el protocolo.

- Gestionar sincronización interna con mutex para evitar condiciones de carrera al modificar el estado del juego.

- Detectar el final de la partida y notificar al sistema que puede eliminarla (via eraserQueue).

Server

La clase Server es el punto de entrada del servidor. Sus responsabilidades son:

- Instanciar y coordinar los tres contextos del juego: ServerLobby, ServerGameLobby y ServerInGame.

- Leer continuamente órdenes (Order) desde el protocolo mediante getNextOrder().

- Delegar la ejecución de cada orden al componente correspondiente según su tipo (OrderType) a través del orderTranslator.

Esta estructura distribuida permite manejar múltiples partidas en simultáneo, manteniendo independencia entre los jugadores y escalabilidad. La separación entre protocolo, procesamiento y lógica de juego mejora la mantenibilidad y claridad del código.

4. Protocolo de comunicación

4.1 Objetivo del protocolo

El protocolo de comunicación es el mecanismo que permite el intercambio de datos entre el cliente y el servidor. Fue implementado utilizando sockets TCP bloqueantes y un formato binario personalizado.

4.2 Clases principales involucradas

El protocolo está dividido en varias clases, todas centradas en el flujo de datos desde el socket hasta la lógica del juego. Las principales del lado del servidor son:

- Protocol: clase base que mantiene una cola de requests entrantes y delega la deserialización al protocolo correspondiente.
- GameLobbyProtocol / InGameProtocol / ServerLobbyProtocol: componentes especializados que interpretan los datos según el estado del juego.

Del lado cliente:

- ProtocolClient: clase que permite enviar y recibir mensajes binarios con el servidor.

4.3 Flujo de comunicación

1. Se recibe una orden del cliente en Client Handler
2. Protocolo interpreta la orden para que el Server la procese
3. El Server general delega al server del estado actual del juego
4. El Server encargado de esa etapa delega a la función correspondiente
5. La función realiza la acción pedida

4.4 Estructura de los mensajes

Cada mensaje binario tiene el siguiente formato:

- Opcode (1 byte): indica el tipo de acción (por ejemplo, disparar, moverse, comprar, etc.).
- Payload variable: contiene los datos específicos según el tipo de mensaje.

4.5 Uso de DTOs (Data Transfer Objects)

Para estructurar y organizar los datos transmitidos, se utilizan objetos DTO. Algunos ejemplos son:

- Snapshot: Envía todo lo necesario para mostrar en el juego
- Order: Se utiliza para encapsular una orden del cliente.
- Request: En ella se encuentran todos los mensajes en binario

5. Formato de archivos

5.1 Configuración (YAML)

El archivo GameInfo.yaml contiene los valores básicos que rigen el funcionamiento general de una partida. Incluye:

- **Parámetros del jugador**: cantidad de vida inicial, dinero al comenzar y recompensa por eliminar enemigos.

- **Parámetros del juego**: monto de dinero por ronda, dinero por ganar una ronda, y duración de la fase de compra.

- **Información del mapa**: tamaño de cada celda (tile), utilizado para cálculos de movimiento y renderizado.

El formato es sencillo y estructurado, con claves como playerInfo, gameInfo y gameMapInfo, cada una agrupando los valores correspondientes.

5.2 Mapas (YAML)

Cada mapa del juego se define en un archivo independiente en formato YAML. Por ejemplo, el archivo aztec.yaml contiene toda la información necesaria para cargar un mapa en el juego.

A continuación, se describe la estructura general de un archivo de mapa:

- **Nombre del mapa**: se especifica mediante la clave name.

- **Grilla de celdas (tiles)**: es una matriz de números enteros donde cada número representa un tipo de casillero (por ejemplo, suelo, pared, punto de aparición, zona de bomba, etc.). Esta grilla define visual y funcionalmente cómo es el mapa.

- **Sección walls**: contiene una lista de identificadores de tile que deben comportarse como paredes. Estos casilleros no pueden ser atravesados por los jugadores.

- **Sección path**: contiene los identificadores de tile que representan caminos, es decir, casilleros transitables por los jugadores.

- **Secciones de puntos de aparición**:

    - terroristSpawnPoints: lista de coordenadas { x: <int>, y: <int> } donde pueden aparecer los jugadores del equipo terrorista.

    - counterSpawnPoints: coordenadas de aparición del equipo antiterrorista.

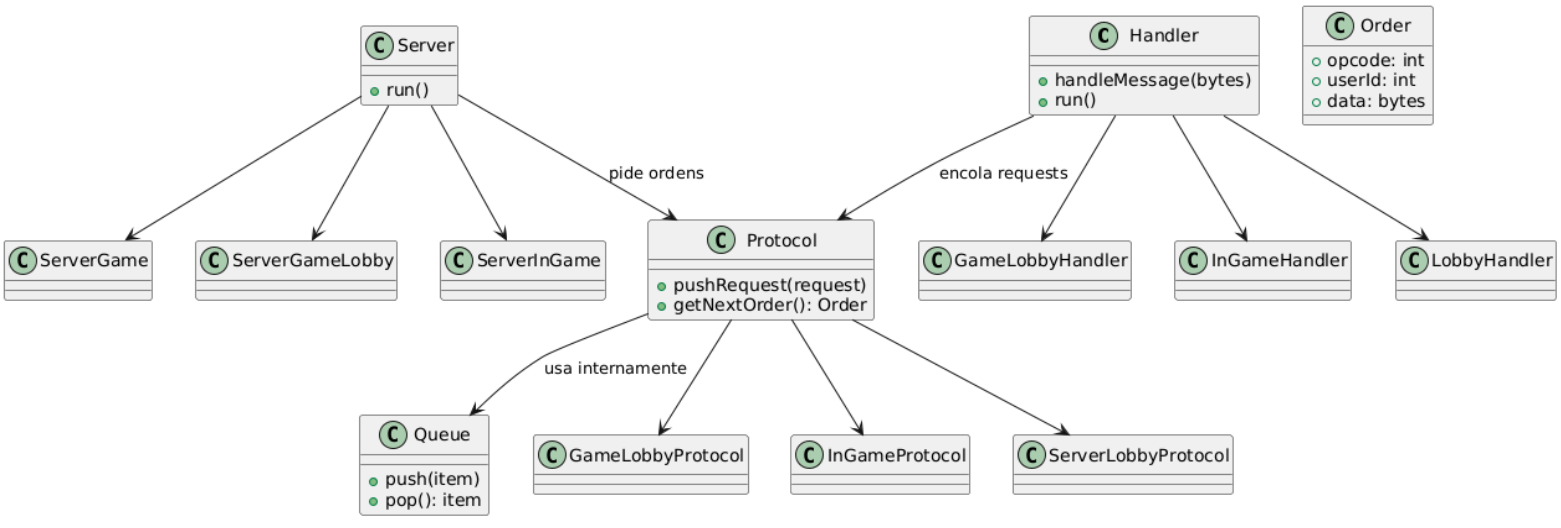
    - bombPlantPoints: posiciones válidas para plantar la bomba.

- **Sección drops**: define armas preexistentes en el mapa. Cada entrada incluye una coordenada y el código del arma (weaponCode), por ejemplo: {x: <int>, y: <int>, weaponCode: <int> }.

6. Diagramas

6.1 Diagrama de clases

- Estructura principal del servidor



Este diagrama representa la arquitectura base del lado servidor, destacando los módulos responsables de recibir mensajes, interpretarlos y ejecutar acciones dentro del juego.

Componentes principales:

- **Server**: clase principal encargada de iniciar la aplicación y gestionar el ciclo de ejecución principal. Delegará la lógica a ServerGame, ServerGameLobby o ServerInGame según el estado del juego.

- **Handler**: encargado de recibir los mensajes binarios de los clientes. Cada mensaje se delega al handler correspondiente (GameLobbyHandler, InGameHandler, o LobbyHandler) en función de su tipo (opcode).

- **Protocol**: componente central para el procesamiento de mensajes. Se encarga de almacenar las requests recibidas y convertirlas en objetos del tipo Order para que el servidor los interprete.

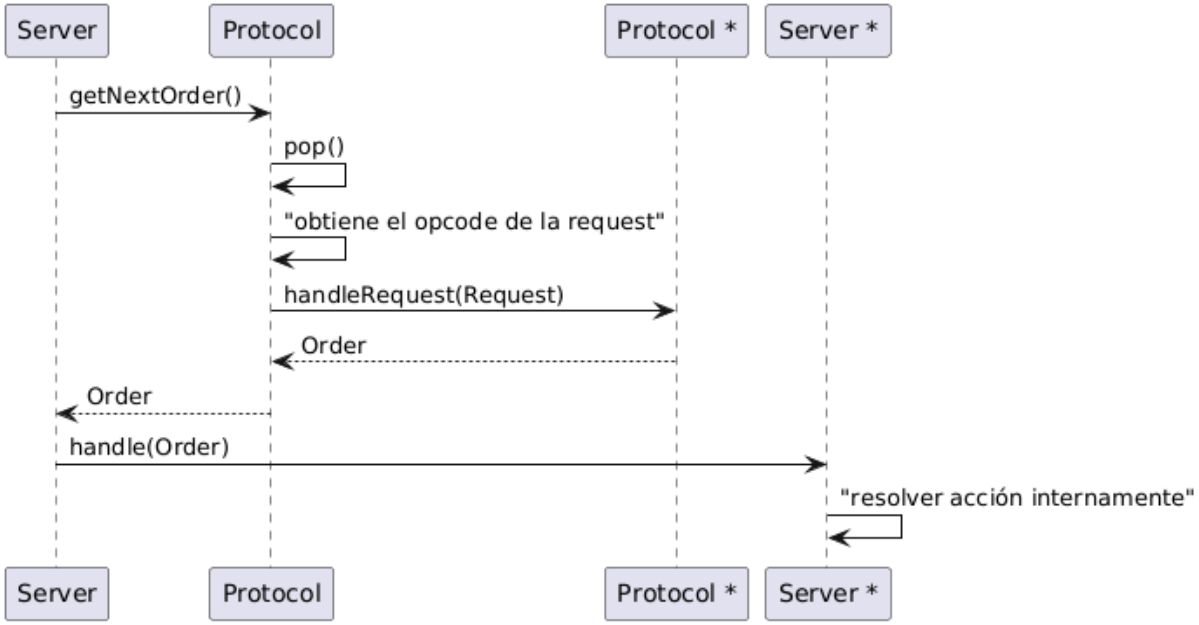
- **Order**: estructura que contiene toda la información necesaria (opcode, userId, datos) para que el servidor procese una acción del cliente.

- **Queue**: estructura auxiliar utilizada internamente por Protocol para almacenar y recuperar pedidos de forma ordenada y segura.

- **Protocolos específicos**: GameLobbyProtocol, InGameProtocol y ServerLobbyProtocol implementan la lógica de decodificación para diferentes tipos de órdenes según el contexto.

6.2 Diagramas de secuencia

- Manejo de orders



El servidor ejecuta getNextOrder() sobre el Protocol, consultando si hay órdenes pendientes a procesar.

El Protocol hace un pop() interno de la cola de requests.

Extrae la request y obtiene su opcode para identificar el tipo de mensaje recibido.

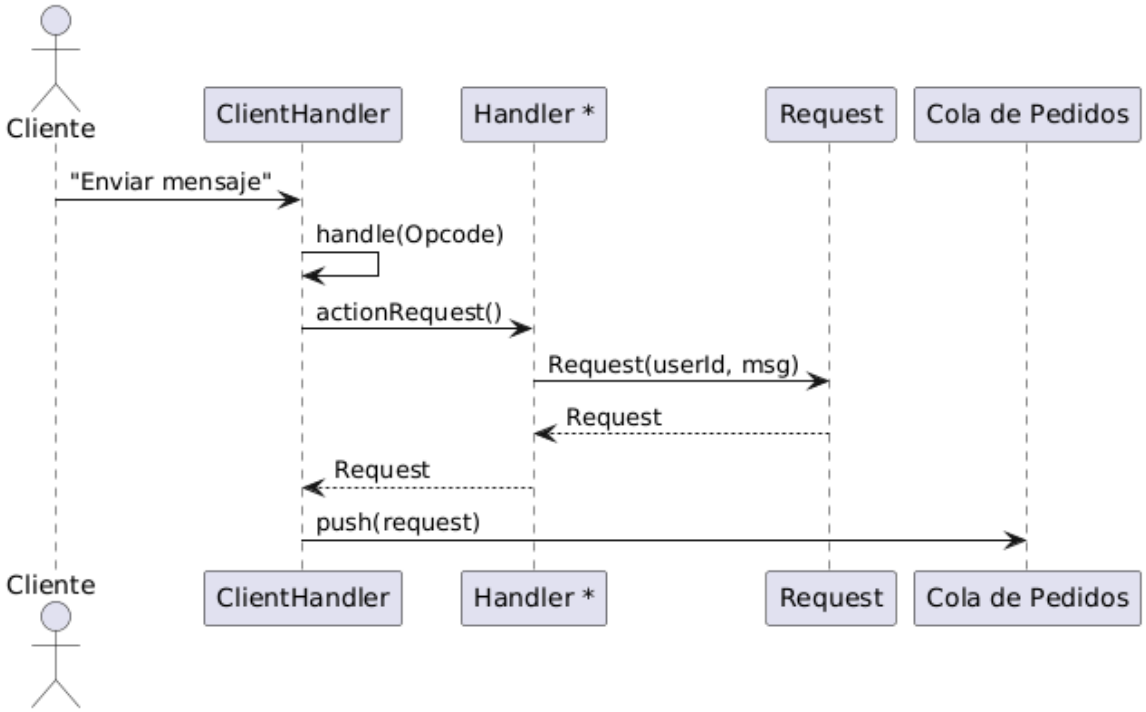
Luego, delega el procesamiento de la request a uno de los protocolos especializados (GameLobbyProtocol, InGameProtocol, etc.) mediante handleRequest(request).

El protocolo correspondiente devuelve un objeto Order con todos los datos ya interpretados (opcode, ID de usuario, datos).

El servidor llama a handle(order), que reenvía la orden al módulo correspondiente (por ejemple, ServerInGame) según el tipo.

Finalmente, el servidor resuelve internamente la acción solicitada por el cliente

- Envío de mensaje del cliente



El cliente envía un mensaje hacia el servidor. Este mensaje llega al ClientHandler correspondiente, que es el encargado de leer los datos del socket.

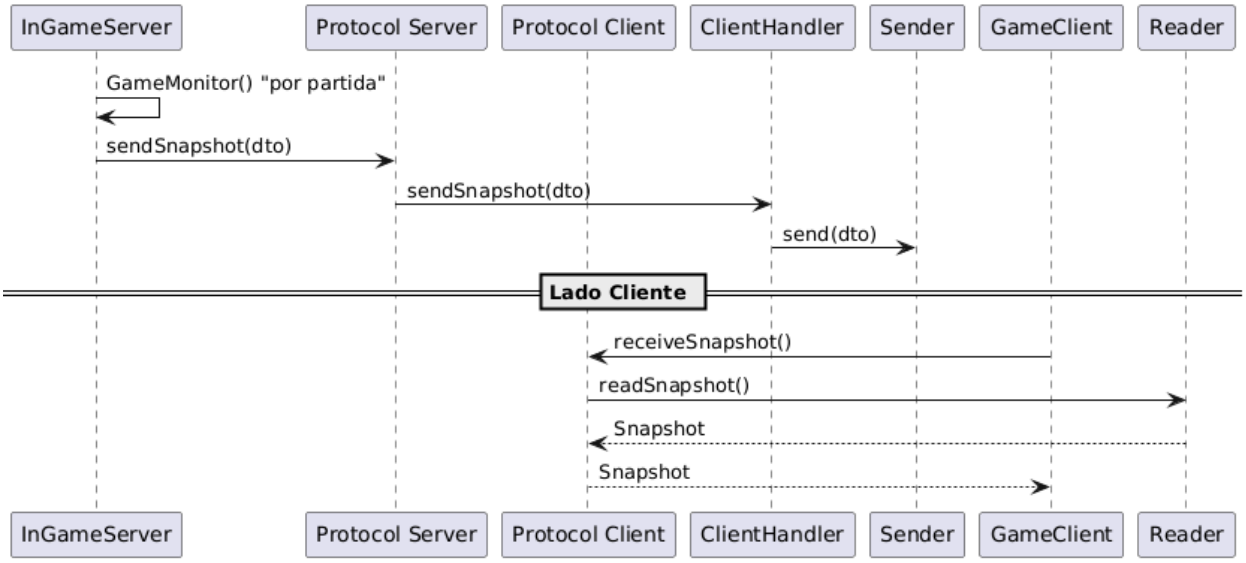
El ClientHandler interpreta el opcode recibido y llama al Handler correspondiente mediante handle(opcode).

El Handler especializado genera un objeto Request, que contiene el userId y los datos en bruto del mensaje (msg).

La Request es devuelta al ClientHandler, que la reenvía al Protocol.

Finalmente, el Protocol inserta la Request en la cola de pedidos, quedando lista para ser procesada cuando el servidor consulte la próxima orden.

- Envío de snapshot



El InGameServer, mediante su instancia de GameMonitor (una por partida), genera periódicamente un snapshot del estado del juego (dto).

Este snapshot se transmite usando el ProtocolServer, que lo envía al ProtocolClient del lado cliente.

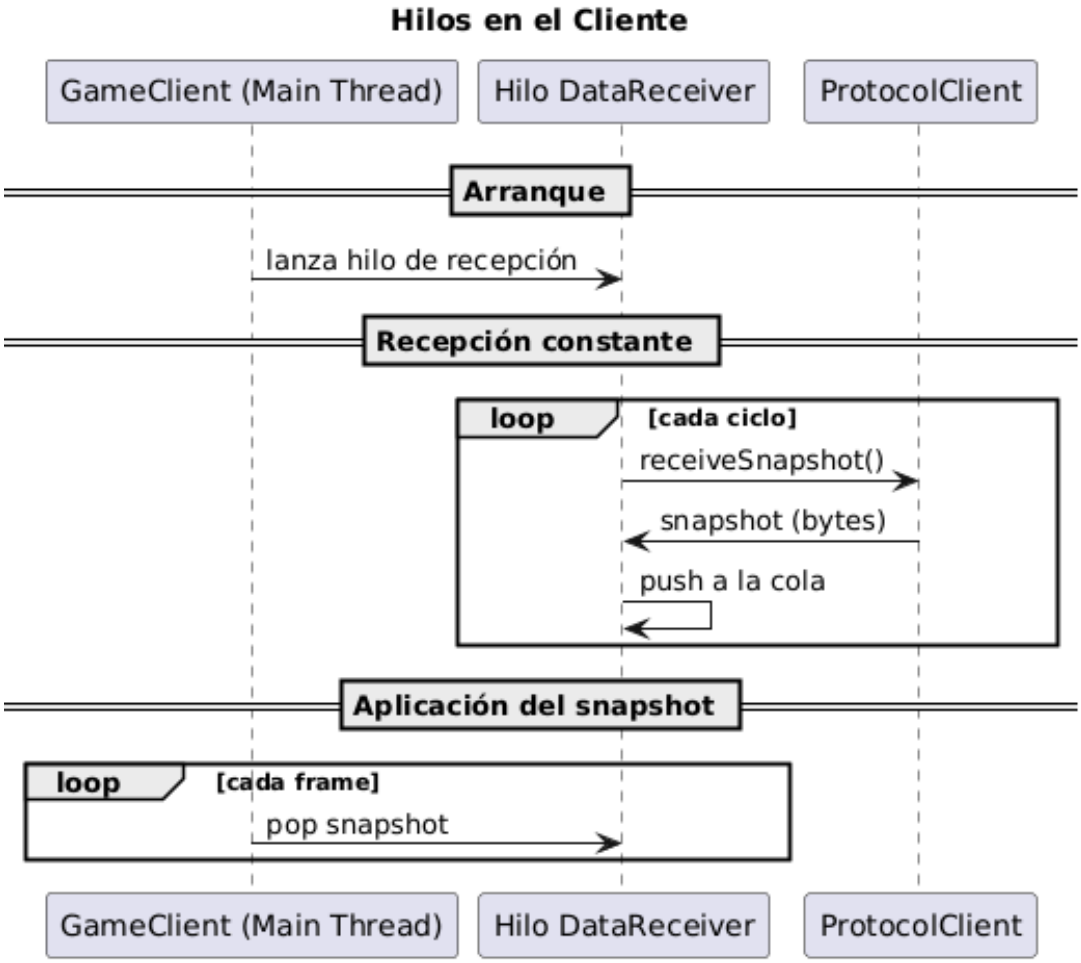
El ProtocolClient utiliza el ClientHandler y el componente Sender para realizar el envío del snapshot de forma binaria por socket.

En el cliente, el ProtocolClient llama a receiveSnapshot() para recibir los datos.

Luego, se invoca a readSnapshot(), que procesa el contenido recibido.

Finalmente, se transmite el snapshot al Reader, que lo interpreta y lo entrega al GameClient para que actualice su representación local del juego.

- Hilos en el Cliente



GameClient (Main Thread): se encarga de la lógica del juego, renderizado, entrada del jugador y aplicación de snapshots.

Hilo DataReceiver: corre en segundo plano y se encarga de recibir continuamente los snapshots enviados por el servidor.

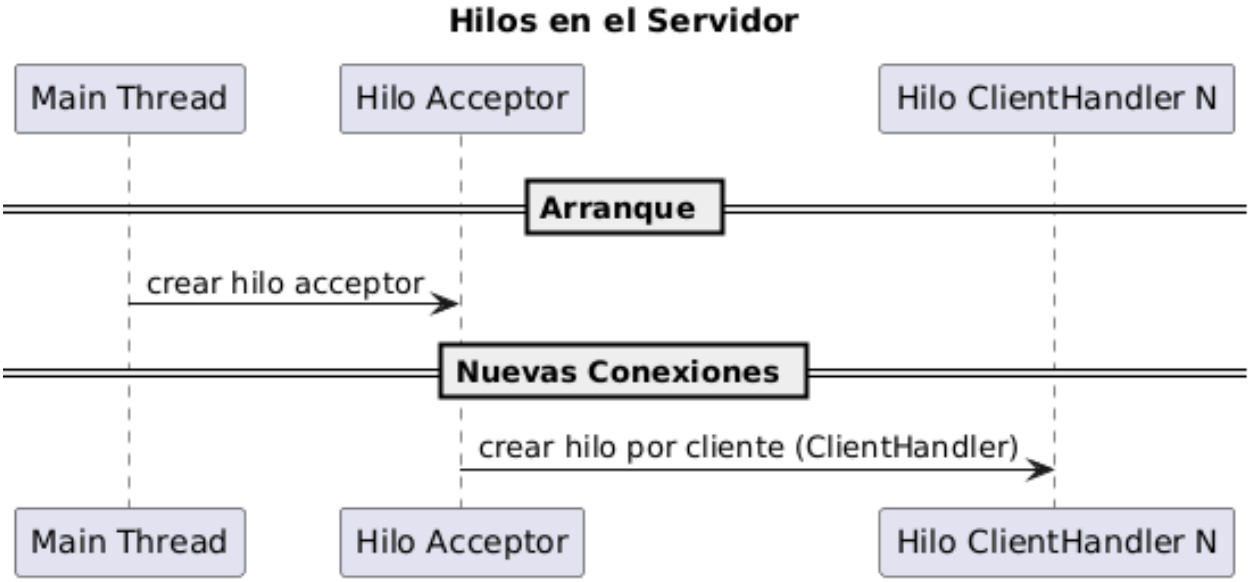
Al iniciar el juego, GameClient lanza el hilo DataReceiver.
El hilo DataReceiver entra en un loop continuo donde:

- Llama a receiveSnapshot() del ProtocolClient.
- Recibe los bytes del snapshot y los encola para su posterior uso.

Mientras tanto, el GameClient en su bucle principal:

- Hace pop de la cola de snapshots.
- Aplica la información recibida al estado del juego.

- Hilos en el Servidor



Main Thread: arranca el servidor y lanza el hilo Acceptor.

Hilo Acceptor: escucha nuevas conexiones entrantes.

Hilo ClientHandler N: por cada cliente conectado, se lanza un hilo individual para manejar su comunicación.

El hilo principal lanza un hilo Acceptor que se mantiene a la espera de nuevas conexiones.
Por cada nueva conexión, el Acceptor lanza un nuevo hilo ClientHandler, encargado exclusivamente de ese cliente.

Esto permite al servidor manejar múltiples jugadores en paralelo, manteniendo independencia entre conexiones.

7. Componentes claves

- Sender: El Sender es un componente que permite al usuario de la clase abstraerse de la utilización del socket para mandar los datos correspondientes, esto asegura un correcto envío de los datos en cada situación y una validación transparente.
- Reader: El Reader permite, de manera análoga al sender, recibir la información de un socket manteniendo un alto grado de transparencia en esta transacción.

8. Consideraciones técnicas

El proyecto se desarrolló en C++20 bajo el estándar POSIX 2008, cumpliendo con los lineamientos establecidos en el enunciado. La arquitectura del sistema está basada en un esquema cliente-servidor, donde:

- El servidor gestiona la lógica del juego, múltiples partidas simultáneas y mantiene la comunicación con los distintos clientes.

- El cliente proporciona la interfaz gráfica de usuario, interacción con los controles y renderizado de la partida.

- El editor permite diseñar y validar mapas, definir zonas de inicio y puntos de activación de bomba, todo mediante una interfaz gráfica.

Las comunicaciones entre cliente y servidor se implementan mediante un protocolo binario propio, sin el uso de bibliotecas externas. Todos los sockets utilizados son bloqueantes.

Las configuraciones generales del juego, así como las constantes numéricas (vida, daño, frecuencia de disparo, precios, etc.), se definen en archivos YAML, lo cual permite modificar valores sin necesidad de recompilar el código.

La interfaz gráfica fue desarrollada utilizando las bibliotecas SDL2 y Qt6, permitiendo soporte para modo ventana y pantalla completa, con resolución configurable.

9. Dependencias y compilación

El proyecto está diseñado para ser compilado y ejecutado en sistemas Ubuntu 24.04 o Xubuntu 24.04.

Las dependencias necesarias incluyen:

- g++ (con soporte para C++20)
- CMake
- SDL2
- SDL2_image
- SDL2_mixer
- SDL2_ttf
- Qt6
- yaml-cpp
- GoogleTest (para los tests del protocolo)

10. Anexos

- https://github.com/eldipa/hands-on-threads
- https://github.com/eldipa/hands-on-sockets-in-cpp
- https://www.sounds-resource.com/pc_computer/counterstrikeglobaloffensive/sound/23423/
- https://www.voicy.network/official-soundboards/games/csgo
- https://www.youtube.com/playlist?list=PLvv0ScY6vfd-p1gSnbQhY7vMe2rng0IL0
-