TALLER DE PROGRAMACIÓN
(TA045) CURSO 01 - Veiga

Documentación técnica

2do Cuatrimestre 2024

Integrantes

Santiago Domínguez	Federico Nemirovsky	Franco Rodríguez	Franco Valfré
110.010	108.560	108.799	105.607

INTRODUCCIÓN

Este archivo sirve como guía para comprender el funcionamiento de los programas presentes en este trabajo practico, entender la lógica y las decisiones tomadas. Principalmente, la arquitectura de este proyecto se puede dividir en dos grandes partes: el cliente y el servidor

CLIENTE

El cliente es el componente del sistema encargado de interactuar directamente con el usuario.

Inicialmente, el cliente opera en el lobby, donde establece una conexión con el servidor. Durante esta etapa, todas las operaciones se realizan de forma secuencial y en un único hilo de ejecución. En este entorno, el usuario puede decidir si desea crear una nueva partida o unirse a una ya existente. Para ello, el cliente crea un socket que permite la comunicación con el servidor, el cual responde enviando información relevante, como los mapas disponibles o las partidas disponibles para unirse.

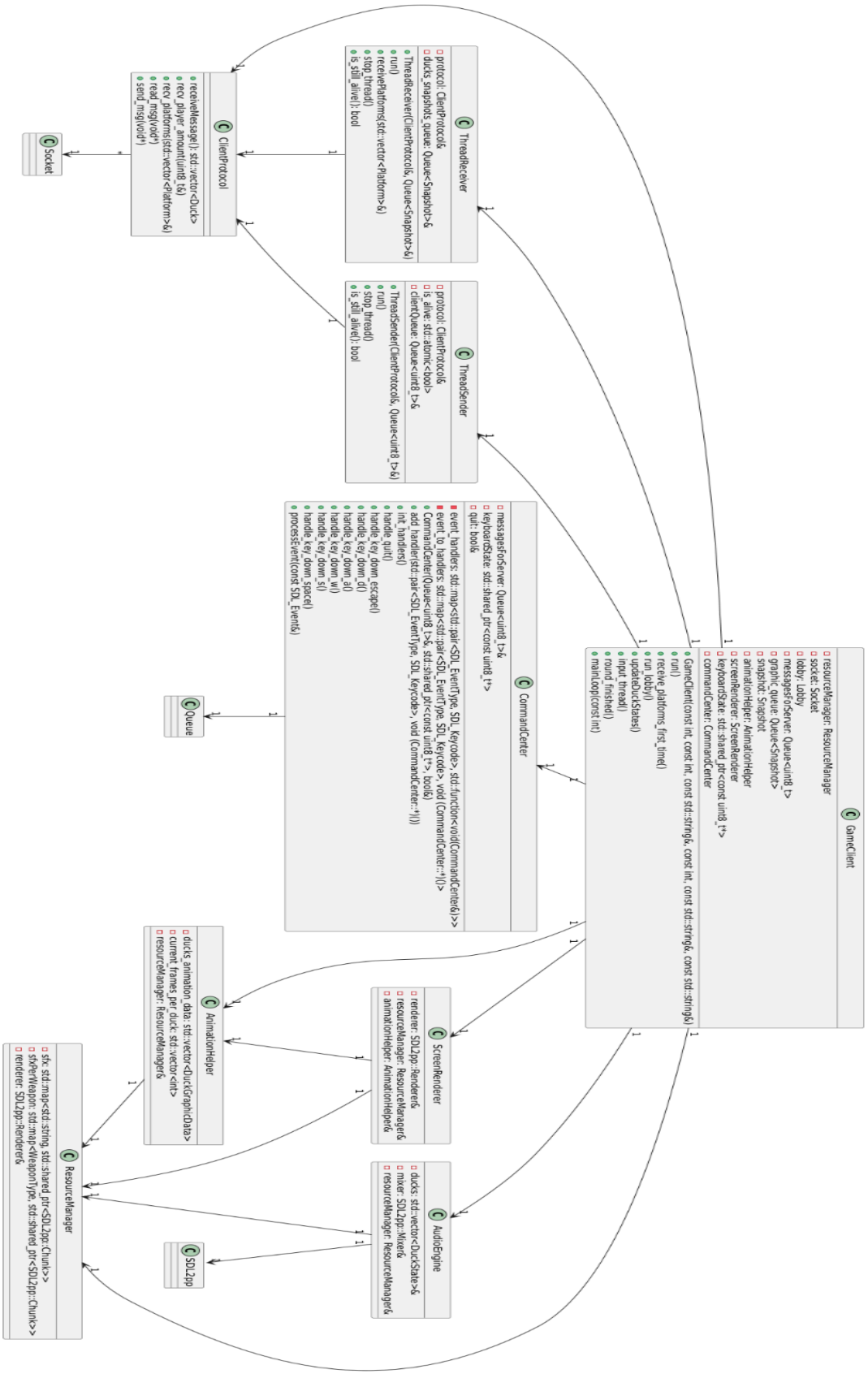
Tras completar la interacción en el lobby, el cliente permanece en espera hasta que la partida cuente con la cantidad de jugadores requerida, previamente especificada por el creador. Una vez que se alcanza este número, los jugadores pueden comenzar a moverse dentro del entorno del juego y competir por sobrevivir en cada ronda con el objetivo final de ganar la partida.

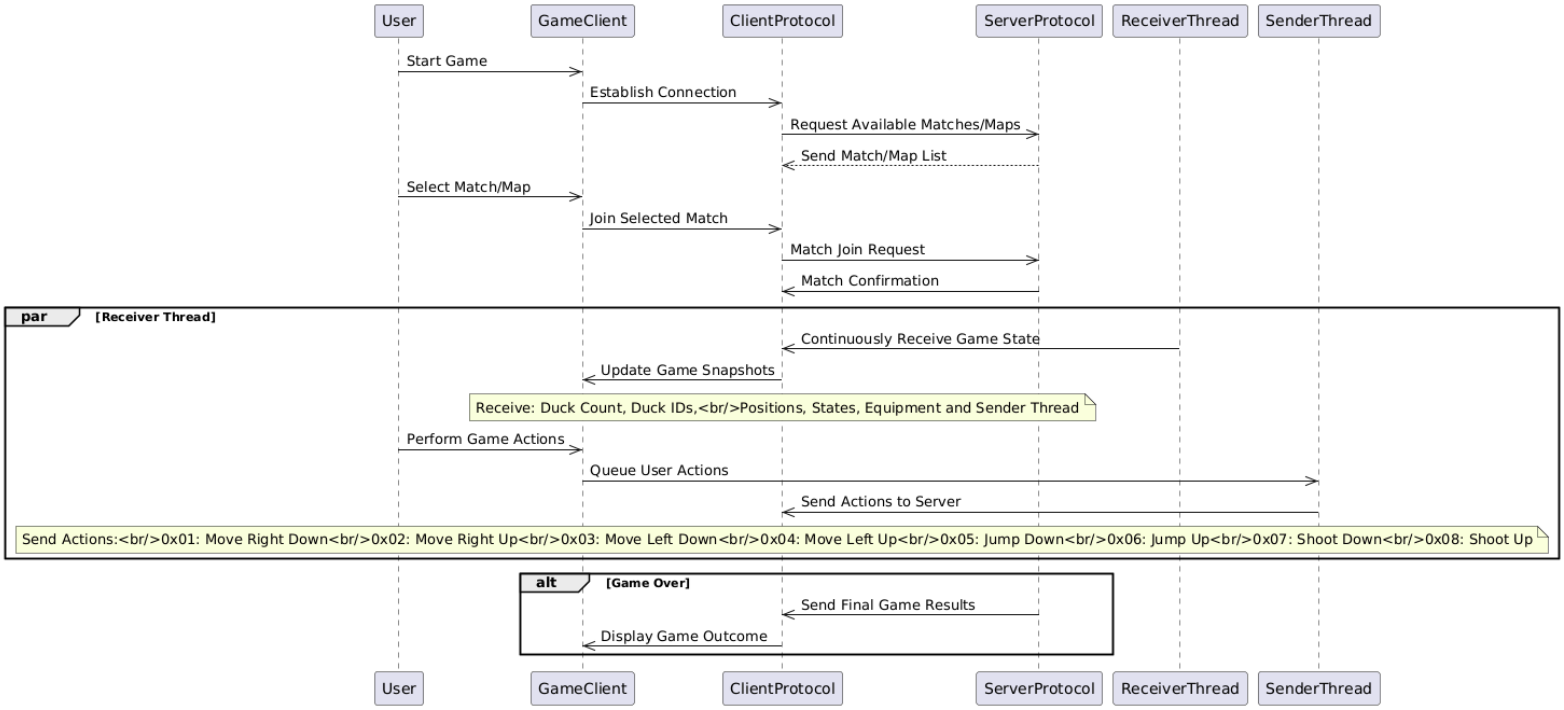
En términos técnicos, la clase principal, denominada **GameClient**, tiene como atributos principales el **socket**, el **lobby**, el **Renderer** y el **CommandCenter**. Al finalizar la ejecución del lobby, esta clase establece un protocolo con el mismo socket y lanza dos hilos: uno dedicado a recibir actualizaciones periódicas en forma de *snapshots* sobre el estado de la partida, y otro responsable de enviar los comandos realizados por los usuarios.

El ciclo de juego termina cuando uno de los jugadores, representado por un pato, gana cinco rondas. En este punto, todos los recursos se liberan correctamente para evitar fugas de memoria. Además, se implementa un manejo exhaustivo de excepciones, lo que garantiza una interacción eficiente y fluida con el programa, incluso en caso de errores inesperados.

Además, el cliente incluye la posibilidad de ejecutar un programa adicional, independiente de los procesos mencionados anteriormente, cuyo propósito es la creación de mapas personalizados. Este programa permite al usuario diseñar libremente el escenario del juego, incluyendo la colocación de plataformas, la configuración de las armas disponibles, y la asignación de los puntos de aparición (*respawn*) para los personajes. Esto brinda flexibilidad y creatividad, permitiendo a los jugadores personalizar su experiencia de juego.

A continuación se encuentran algunos diagramas del client:


GameClient
resourceManager: ResourceManager
socket: Socket
lobby: Lobby
messagesForServer: Queue<uint8_t>
graphic_queue: Queue<Snapshot>
snapshot: Snapshot
animationHelper: AnimationHelper
screenRenderer: ScreenRenderer
keyboardState: std::shared_ptr<const uint8_t*>
commandCenter: CommandCenter
GameClient(const int, const int, const std::string&, const int, const std::string&, const std::string&)
run()
receive_platforms_first_time()
run_lobby()
updateDuckStates()
input_thread()
round_finished()
mainLoop(const int)
ThreadReceiver
protocol: ClientProtocol&
ducks_snapshots_queue: Queue<Snapshot>&
ThreadReceiver(ClientProtocol&, Queue<Snapshot>&)
run()
receivePlatforms(std::vector<Platform>&)
stop_thread()
is_still_alive(): bool
ThreadSender
protocol: ClientProtocol&
is_alive: std::atomic<bool>
clientQueue: Queue<uint8_t>&
ThreadSender(ClientProtocol&, Queue<uint8_t>&)
run()
stop_thread()
is_still_alive(): bool
ClientProtocol
receiveMessage(): std::vector<Duck>
recv_player_amount(uint8_t&)
recv_platforms(std::vector<Platform>&)
read_msg(void*)
send_msg(void*)
Socket
CommandCenter
messagesForServer: Queue<uint8_t>&
keyboardState: std::shared_ptr<const uint8_t*>
quit: bool&
event_handlers: std::map<std::pair<SDL_EventType, SDL_Keycode>, std::function<void(CommandCenter&)>>
event_to_handlers: std::map<std::pair<SDL_EventType, SDL_Keycode>, void (CommandCenter::*)()>
CommandCenter(Queue<uint8_t>&, std::shared_ptr<const uint8_t*>, bool&)
add_handler(std::pair<SDL_EventType, SDL_Keycode>, void (CommandCenter::*)())
init_handlers()
handle_quit()
handle_key_down_escape()
handle_key_down_d()
handle_key_down_a()
handle_key_down_w()
handle_key_down_s()
handle_key_down_space()
processEvent(const SDL_Event&)
Queue
AnimationHelper
ducks_animation_data: std::vector<DuckGraphicData>
current_frames_per_duck: std::vector<int>
resourceManager: ResourceManager&
ScreenRenderer
renderer: SDL2pp::Renderer&
resourceManager: ResourceManager&
animationHelper: AnimationHelper&
AudioEngine
ducks: std::vector<DuckState>&
mixer: SDL2pp::Mixer&
resourceManager: ResourceManager&
ResourceManager
sfx: std::map<std::string, std::shared_ptr<SDL2pp::Chunk>>
sfxPerWeapon: std::map<WeaponType, std::shared_ptr<SDL2pp::Chunk>>
renderer: SDL2pp::Renderer&
SDL2pp


User
GameClient
ClientProtocol
ServerProtocol
ReceiverThread
SenderThread
Start Game
Establish Connection
Request Available Matches/Maps
Send Match/Map List
Select Match/Map
Join Selected Match
Match Join Request
Match Confirmation
par
[Receiver Thread]
Continuously Receive Game State
Update Game Snapshots
Receive: Duck Count, Duck IDs,
Positions, States, Equipment and Sender Thread
Perform Game Actions
Queue User Actions
Send Actions to Server
Send Actions:
0x01: Move Right Down
0x02: Move Right Up
0x03: Move Left Down
0x04: Move Left Up
0x05: Jump Down
0x06: Jump Up
0x07: Shoot Down
0x08: Shoot Up
alt
[Game Over]
Send Final Game Results
Display Game Outcome
User
GameClient
ClientProtocol
ServerProtocol
ReceiverThread
SenderThread

SERVER

El servidor constituye un componente fundamental de la arquitectura del sistema del Duck Game. Su diseño responde a la necesidad de gestionar múltiples aspectos de la comunicación y la lógica de juego, proporcionando una infraestructura robusta que permite la interacción simultánea de varios jugadores.

La arquitectura del servidor se fundamenta en una estructura modular que descompone la funcionalidad en distintos componentes. El núcleo de este sistema lo componen tres elementos principales: el Monitor de Partidas (MatchesMonitor), el Juego (Game) y el Monitor de Estado de Partida (MatchStateMonitor). Cada uno de estos componentes cumple roles específicos que garantizan el funcionamiento integral del sistema.

El proceso de comunicación se estructura en dos etapas principales: la conexión inicial y el desarrollo de la partida. Durante la fase de conexión, un hilo aceptador (AcceptorThread) recibe las conexiones entrantes y asigna cada cliente a una sesión (ClientSession) específica. En este punto, los jugadores pueden crear partidas, unirse a partidas existentes o consultar la información de mapas disponibles.

Una vez iniciada la partida, dos hilos especializados gestionan la comunicación: un hilo receptor (ReceiverThread) que procesa las acciones de los jugadores y un hilo emisor (SenderThread) que transmite actualizaciones del estado del juego (Snapshot). Esta arquitectura garantiza una comunicación fluida y en tiempo real entre el servidor y los clientes.

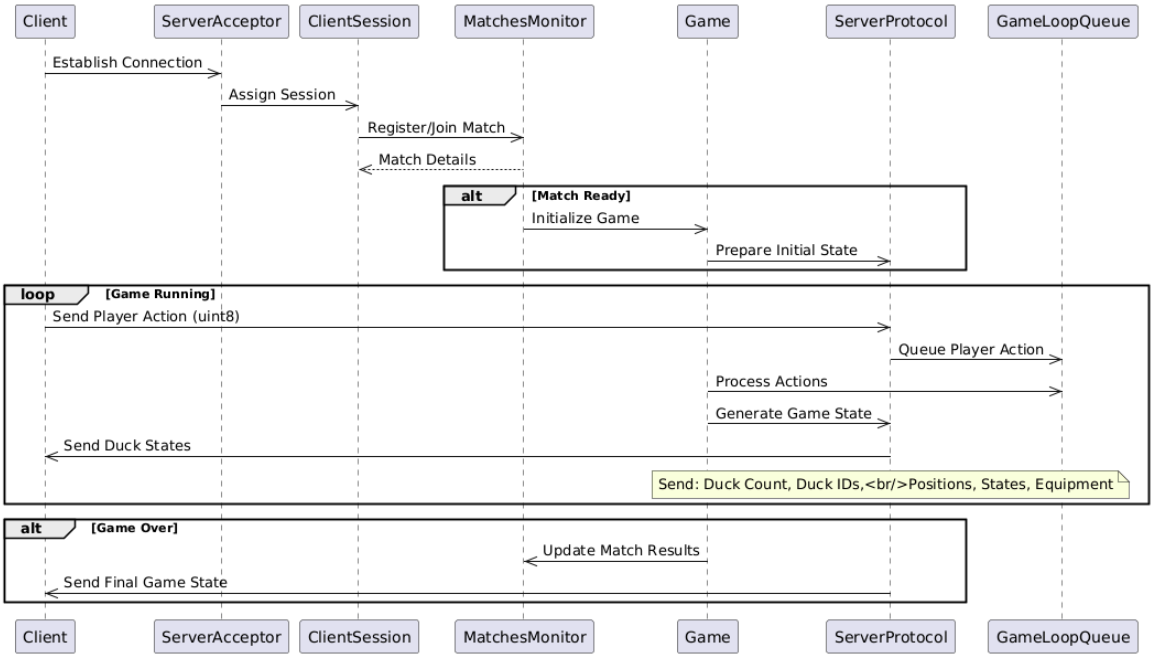
Se utilizan punteros inteligentes para prevenir fugas de memoria, y se han implementado mecanismos de control de recursos (Queue) que permiten una liberación adecuada de los mismos. La concurrencia se gestiona mediante el uso de monitores y colas bloqueantes que garantizan la sincronización entre diferentes componentes del sistema.

El servidor implementa un sistema de juego basado en rondas, donde cada partida se divide en conjuntos de cinco rondas. La finalización de una ronda ocurre cuando únicamente un jugador permanece con vida. El sistema rastrea las victorias de cada jugador y determina el ganador final cuando un jugador alcanza el número de victorias establecido.

La actualización del estado del juego (Game::updateGameState()) representa un proceso crítico. En cada iteración, el servidor procesa múltiples aspectos: actualización de posiciones, gestión de colisiones, interacciones de armas y generación de Snapshots que se transmiten a los clientes.

A continuación se encuentran algunos diagramas del server:

AcceptorThread
acceptor_skt: Socket
clients: std::list<ClientSession*>
matches: MatchesMonitor
AcceptorThread(const std::string&)
run()
accept_connection()
ClientSession
identity: Identity
skt: Socket
protocol: ServerProtocol
matches_monitor: MatchesMonitor&
ClientSession(Socket, MatchesMonitor&)
run_lobby_loop()
run_receiver_loop()
get_match_queue(): Queue<GameloopMessage>*
MatchesMonitor
matches: std::unordered_map<std::string, std::shared_ptr<Match>>
create_match(std::string, uint8_t, DuckIdentity&, Queue<std::shared_ptr<Snapshot>>*, MapInfo): bool
join_match(std::string, DuckIdentity&, Queue<std::shared_ptr<Snapshot>>*): bool
ServerProtocol
recv_msg(uint8_t&)
send_snapshot(std::shared_ptr<Snapshot>)
ReceiverThread
gameloop_queue: Queue<GameloopMessage>&
protocol: ServerProtocol&
run()
SenderThread
protocol: ServerProtocol&
client_queue: Queue<std::shared_ptr<Snapshot>>&
run()
Match
gameloop_queue: Queue<GameloopMessage>
game: Game
state_monitor: MatchStateMonitor
add_player(Queue<std::shared_ptr<Snapshot>>*, DuckIdentity&)
get_gameloop_queue(): Queue<GameloopMessage>*
Game
ducks: std::unordered_map<uint8_t, std::unique_ptr<Duck>>
message_queue: Queue<GameloopMessage>&
monitor: MatchStateMonitor&
action_handler: PlayerActionHandler
addPlayer(DuckIdentity&, const MapInfo&)
updateGameState()
MatchStateMonitor
data_mtx: std::mutex
player_limit: const uint8_t
player_count: std::atomic<uint8_t>
status: std::atomic<MatchStatus>
add_player(Queue<std::shared_ptr<Snapshot>>*, uint8_t&)
push_to_all(std::shared_ptr<Snapshot>)
PlayerActionHandler


Client
ServerAcceptor
ClientSession
MatchesMonitor
Game
ServerProtocol
GameLoopQueue
Establish Connection
Assign Session
Register/Join Match
Match Details
alt
[Match Ready]
Initialize Game
Prepare Initial State
loop
[Game Running]
Send Player Action (uint8)
Queue Player Action
Process Actions
Generate Game State
Send Duck States
Send: Duck Count, Duck IDs,
Positions, States, Equipment
alt
[Game Over]
Update Match Results
Send Final Game State
Client
ServerAcceptor
ClientSession
MatchesMonitor
Game
ServerProtocol
GameLoopQueue