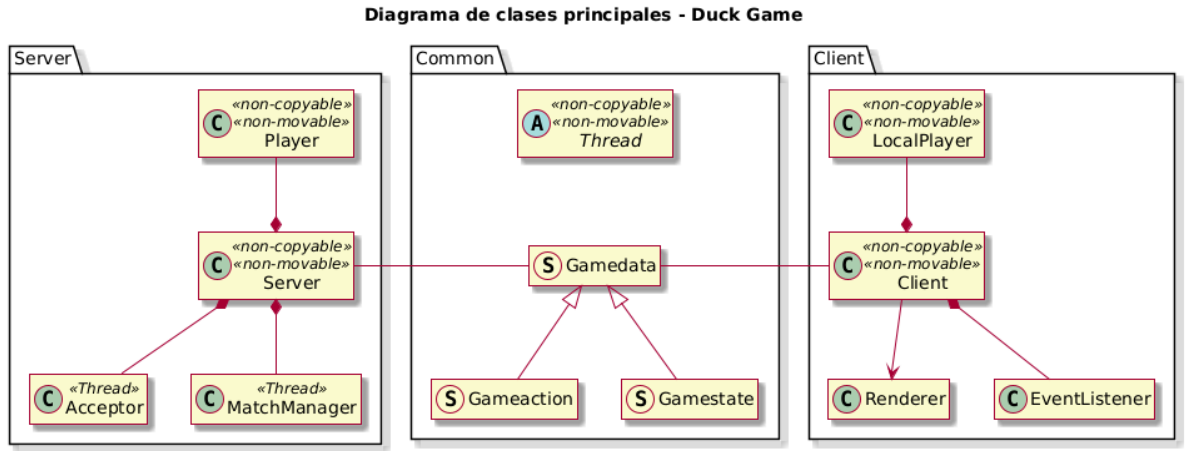
Documentación Técnica

Protocolo de comunicación



Junto con las clases principales, se vinculan las estructuras de datos que forman el protocolo de comunicación entre cliente y servidor: Gamedata, que solo posee el campo **player_id**, diferencia los distintos jugadores por un id dado al unirse un cliente. En esta demo sólo es posible el control del servidor de parte de un sólo jugador.

De cliente a servidor

Desde el cliente se puede mandar un sólo tipo de mensaje, y está encapsulado en el struct Gameaction. Cada atributo del struct es enviado como uint_8 y recibido de la misma manera.
Gameaction cuenta con 5 atributos, de los cuales 4 participan del protocolo: player_id (heredado de Gamedata), **match**, **type**, **key**, is_multiplayer (unused).

- En esta demo sólo se puede utilizar la partida 1 con match.
- Los distintos códigos de type permiten diferenciar las acciones del usuario durante la selección de partidas y durante el juego:

Código Type	Acción
2	Se comenzó a presionar una tecla
3	Se deja de presionar una tecla
4	Creación de partida
9	Se salió de la partida

- Los códigos de key diferencian qué tecla es la que está siendo presionada o soltada por el usuario, código obtenido mediante un map y el struct SDL_Keycode:

Código Key	Tecla
1	Tecla de dirección derecha
2	Tecla de dirección izquierda
3	Spacebar
4	G
5	F
9	Escape [Esc]

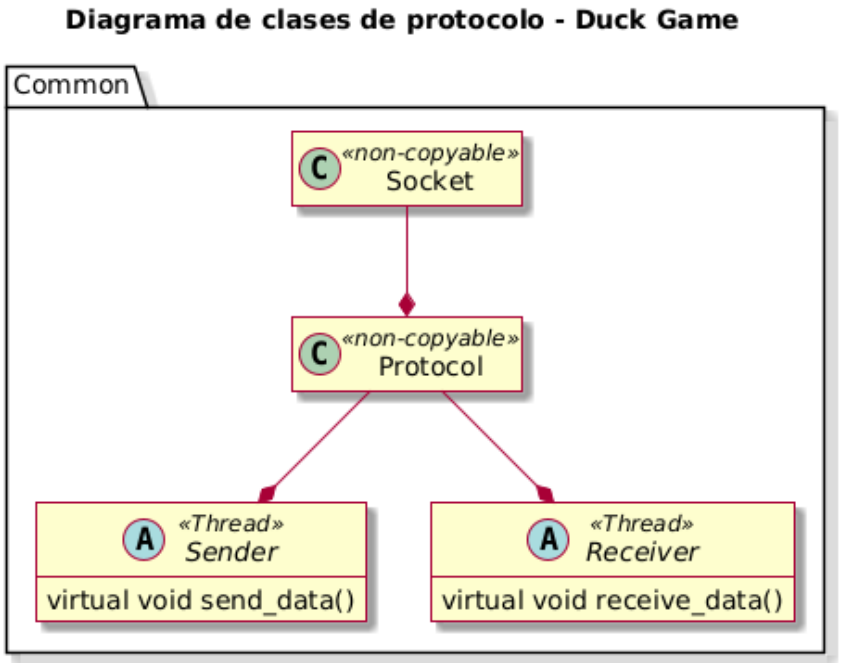
De servidor a cliente

Desde el servidor se puede enviar 8 tipos de mensajes o actualizaciones a los clientes enviando y recibiendo parcialmente los campos del struct Gamestate y diferenciados con el campo **type** (enviado y recibido en los ocho tipos de mensaje):

- type (1) - Setup de personajes inicial, utilizando: player_id (heredado de Gamedata), type, **pos_X**, **pos_Y**, **is_running**, **is_jumping**, **is_flapping**, **move_direction**, **is_alive**, **jump_speed**.
- type (2) - Actualización de posiciones de todos los personajes, utilizando type y el campo **positions_by_id** (un map con claves struct **Coordinates** y id).
- type (3) - Actualización de estado de un personaje, utilizando: player_id, type, is_running, is_jumping, is_flapping, move_direction, is_alive.
- Mensajes de actualizaciones con respecto a armas:

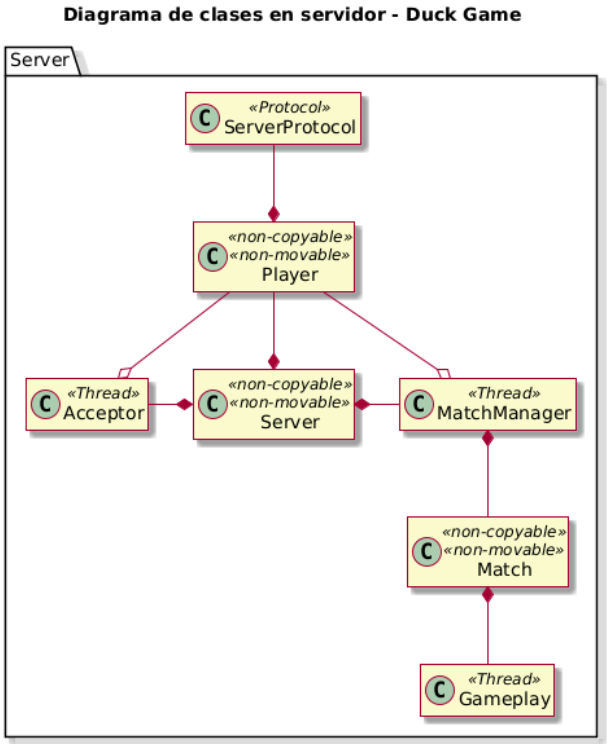
type (n)	Mensaje
5	Inicialización de un arma
6	Actualización de posiciones de armas
7	Inicialización de una bala
8	Actualización de posiciones de balas
9	Señal de bala destruida

Clases de protocolo



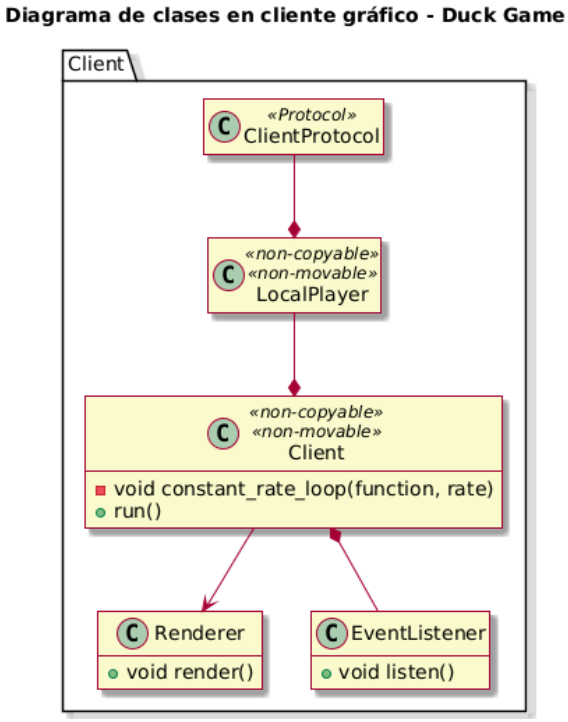
- La clase Protocol es la portadora del Socket peer que espeja al socket del cliente al que escucha. También tiene un mutex para proteger el socket (al menos en el servidor) de race conditions tratando de cerrarlo (con close_comms(), método implementado en Protocol).
- Los hilos Sender y Receiver tienen la referencia de un Protocol como atributo (y por lo tanto acceso a mandar y recibir mensajes de un sólo socket). Por otro lado, los loops implementados en estas dos clases dependen de la lectura de un atomic_bool is_connected, cuya referencia obtienen del protocolo: si el socket produce algun error, hace que los hilos se detengan.
  En las implementaciones de las clases hijas (ServerReceiver, ServerSender, ClientSender, ClientReceiver) se maneja separadamente el graceful shutdown.
  Las clases hijas poseen referencias a Queues de Gameaction o Gamestate (según corresponda) como recursos de envío y recibimiento de mensajes.
- Las clases Player en servidor y LocalPlayer en cliente, poseen un protocolo y sus propios Sender y Receiver, constituyendo la trifecta de comunicación de cada lado del sistema.
  Estas dos son las encargadas de lanzar y detener los hilos que poseen como atributos. También se puede terminar la comunicación con sus respectivos métodos.

Clases de servidor



- La clase Server posee una lista de Player y los hilos Acceptor y MatchManager como atributos. Así como las queues principales en las que se basa la comunicación con los clientes: **Queue<Gameaction> users_commands** y **Queue<Gamestate> server_messages**
- El hilo Acceptor recibe a los jugadores (socket.accept()) y crea punteros a Player con los sockets creados, se establece la comunicación con un cliente al construirse Player.
- El MatchManager recibe los mensajes de todos los usuarios y crea punteros a Match con una lista de jugadores que va creciendo a medida que se unen a la misma. Esta clase también da su id a los jugadores y envía información de las partidas a todos los jugadores.
- Por otro lado Match tiene un hilo Gameplay como atributo y es la encargada de lanzarlo y detenerlo, mediante un graceful shutdown.
- Cada Gameplay posee la referencia de una Queue<Gameaction> de la cuál procesa mensajes que tienen que ver con el juego; y una lista de jugadores fija. Para enviar mensajes, agrega un Gamestate a cada jugador (a la Queue de Player) desde esa lista.

Clases de cliente



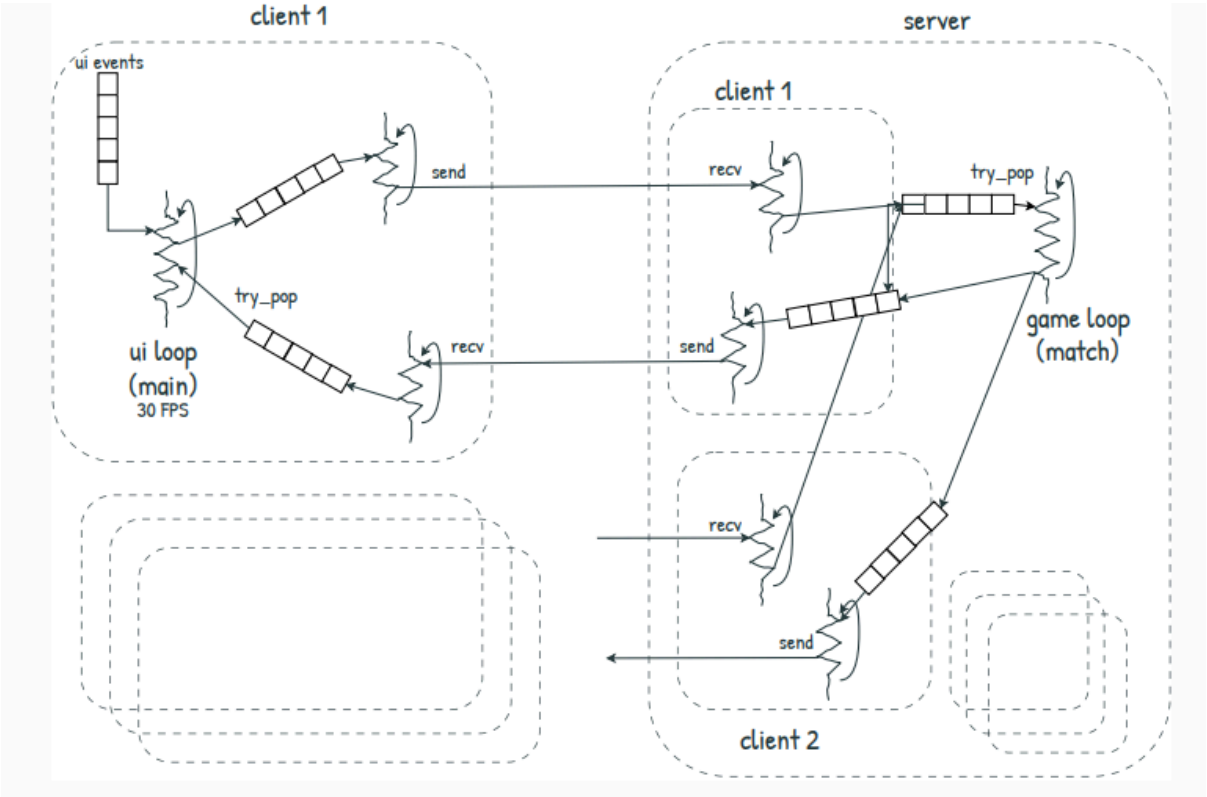
- La clase Client es el núcleo del cliente gráfico, antes de empezar el loop principal, lanza el Lobby (desarrollado con QT) que tiene como atributo.

  El loop principal consta de escuchar eventos del usuario y renderizar un frame. Su condición para repetirse la obtiene de dos booleanos, que chequean que la comunicación siga activa, y el usuario no haya salido del juego. Todo esto con un frame-dropping con rest activo entre ciclos.

- El EventListener escucha los eventos por teclado del usuario con SDL_PollEvent para no bloquear el hilo (se ejecuta en el main thread de cliente!). Esta clase se encarga de construir los mensajes con Gameaction y los envía a una Queue<Gameaction> que comparte con el Sender del cliente.
- El Renderer renderiza un frame por loop y lee actualizaciones desde la **Queue<Gamestate> updates_feed** que comparte con el Receiver del cliente. utilizando clases de SDL2pp, Renderer y Window dibuja el mapa desde el TextureManager que posee como atributo, calcula el zoom segun la información de personajes que existen y muestra la porción de mapa correspondiente. A su vez, dibuja los personajes con distintos colores segun id, armas y proyectiles segun la información del estado.

Diagrama de queues principales cliente-servidor

El modelo de clases se basó en este gráfico obtenido de una clase de arquitectura cliente-servidor de la cátedra:



Otras clases game-wise

- Terrain, en servidor, puede obtener información de un mapa para calcular colisiones con personajes (patos), próximamente colisiones con proyectiles.
- Gun (en cliente y servidor): clases que encapsulan propiedades de distintas armas. En servidor se trata de una clase abstracta.
- Bullet
- StateManager (en cliente y servidor): clases con métodos estáticos para obtener información de Character y Duck respectivamente. En cliente, también se encarga de actualizar el estado de personajes y armas.
- Character en cliente, posee el estado de un personaje para sus distintos tipos de dibujo.
- Duck en servidor, contraparte de Character, posee el estado de un pato y varios métodos para modificarlo. Puede actualizar su propia posición.
- Otras clases de arms en servidor que heredan de Gun en servidor.