Taller de programación I

(75.42/95.08) Course Veiga

Duck Game
Technical documentation

December 3, 2024

Leticia Figueroa 110510
Andrea Figueroa 110450
Josué Martel 110696
Candela Matélica 110641

Introduction

This documentation offers a clear and accessible yet thorough exploration of the project's overall structure. It includes a series of well-crafted diagrams, each accompanied by detailed explanations to ensure clarity and understanding. The documentation delves into several critical aspects of the project, such as the management of threads, highlighting how different threads interact and coordinate to achieve the desired functionality. It also provides an in-depth modeling of the most significant and frequently used classes, offering insights into their relationships, and roles within the system. Furthermore, the protocol's definition is carefully described, illustrating how communication between components is structured and handled. This combination of visual aids and explanatory text ensures a complete understanding of the project's architecture and functionality.

Project overview

The project involves recreating a simplified version of Duck Game, originally developed by Landon Podbielski in 2014. The system architecture follows a **client-server model**, where the server plays a central role by processing commands sent by clients and distributing real-time responses. This design ensures efficient coordination and a seamless flow of data among players.

On the other hand, the clients serve as interactive interfaces, enabling users to send commands to the server and view real-time information provided by the server, such as game state updates and player interactions.

The communication between clients and the server is established via IPv4 sockets over the TCP protocol, adhering to a custom communication protocol specifically designed for this project. Detailed descriptions of the designed protocol are provided in the subsequent sections of this documentation.

Thus, the consolidated system enables users to start a server and create multiple clients that can connect to it. These clients can join real-time multiplayer matches of the classic Duck Game or create a new match that other players can join while connected to the server.

Server-side

Server role

Before starting to delve into the specific design of the server, it is essential to establish what service the server must provide to its clients and what requirements this service implies, so that we can understand what should be able to be achieved globally on the server. and then, once this is done, pertinently establish what the main components of the server are and what role they play in it in order to constitute the system that achieves what is proposed.

In order to be able to provide clients with an experience similar to that provided by the classic Duck Game, we must be able to propose and build a server that acts as a central coordinator that allows us to simultaneously serve multiple players who connect, allow them to choose to join , or create multiplayer games and provide maintenance and support, during each game, to communications with each client, which must be carried out concurrently so that a notable difference is not perceived with respect to when the same message arrives. game state update one player and when it reaches another.

It is also essential that the server guarantees the isolation of active matches from each other, that is, once a client is a participant in a match, it is in this match thread that where player's commands will be processed and in no other, as well as this match will transmit its match state data (such as the movements, actions or results) to this client and the rest of the players participating in it, without these messages reaching other clients who do not care (and they shouldn't have to do it) about managing messages related to matches in which they are not even participants.

Finally, the server will also have to manage the release of the resources intended to achieve the previously mentioned, both when finishing a match, when noticing the disconnection of a client, and when receiving the indication to close the service offered by the server.

By achieving these requirements, the service provided will be able to guarantee a fluid, synchronized and highly interactive experience, allowing players to enjoy a dynamic and competitive environment with the flexibility to participate in multiple matches.

Concurrency mechanisms considered

Once the considerations of the service that the server must guarantee have been established, the need to carry out multiple tasks concurrently in a safe and efficient manner is palpable, if not evident.

So, in order to correctly understand the depth of the implemented scheme that we intend to carry out in this documentation, we find it necessary to mention in advance the concurrency mechanisms considered:

- Using multiple threads that will carry out specific tasks on the server
- Use of thread safe Queues with blocking push-pop methods to avoid Busy Waitings and non-blocking ones to keep the sequence of each gameLoop not blocked, that is, not blocking the passage of time in the worlds of each match.
- Use of mutexes that guarantee safe access to shared resources by multiple threads, as well as the implementation of monitor classes that make use of mutexes to guarantee thread-safe access to shared resources.

Server architecture and main components

The server architecture to offer a multiplayer video game service with multiple matches and the ability for each client to manage their participation in these is based on a flexible combination of an acceptor thread, communication threads dedicated to maintaining communication with a client and dynamic processor threads (match threads) that manage commands and updates from/for multiple clients.

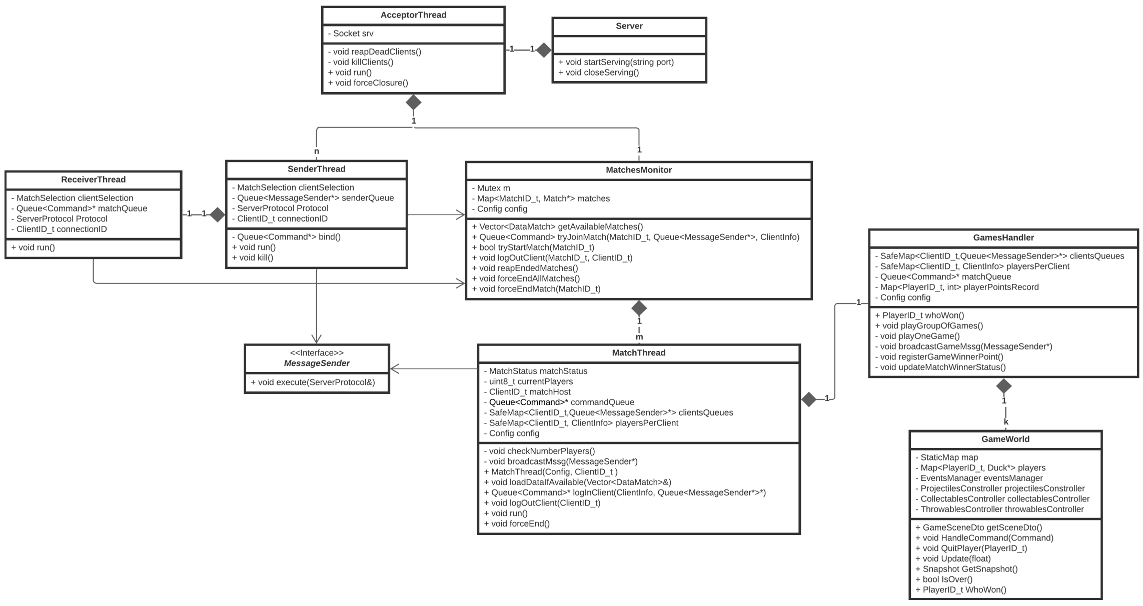

AcceptorThread
- Socket srv
- void reapDeadClients()
- void killClients()
+ void run()
+ void forceClosure()
Server
+ void startServing(string port)
+ void closeServing()
1
1
1
n
1
ReceiverThread
- MatchSelection clientSelection
- Queue<Command>* matchQueue
- ServerProtocol Protocol
- ClientID_t connectionID
+ void run()
1
1
SenderThread
- MatchSelection clientSelection
- Queue<MessageSender*> senderQueue
- ServerProtocol Protocol
- ClientID_t connectionID
- Queue<Command*> bind()
+ void run()
+ void kill()
MatchesMonitor
- Mutex m
- Map<MatchID_t, Match*> matches
- Config config
+ Vector<DataMatch> getAvailableMatches()
+ Queue<Command> tryJoinMatch(MatchID_t, Queue<MessageSender*>, ClientInfo)
+ bool tryStartMatch(MatchID_t)
+ void logOutClient(MatchID_t, ClientID_t)
+ void reapEndedMatches()
+ void forceEndAllMatches()
+ void forceEndMatch(MatchID_t)
1
m
<<Interface>>
MessageSender
+ void execute(ServerProtocol&)
MatchThread
- MatchStatus matchStatus
- uint8_t currentPlayers
- ClientID_t matchHost
- Queue<Command>* commandQueue
- SafeMap<ClientID_t,Queue<MessageSender>*> clientsQueues
- SafeMap<ClientID_t, ClientInfo> playersPerClient
- Config config
- void checkNumberPlayers()
- void broadcastMssg(MessageSender*)
+ MatchThread(Config, ClientID_t)
+ void loadDataIfAvailable(Vector<DataMatch>&)
+ Queue<Command>* logInClient(ClientInfo, Queue<MessageSender*>*)
+ void logOutClient(ClientID_t)
+ void run()
+ void forceEnd()
1
1
GamesHandler
- SafeMap<ClientID_t,Queue<MessageSender>*> clientsQueues
- SafeMap<ClientID_t, ClientInfo> playersPerClient
- Queue<Command>* matchQueue
- Map<PlayerID_t, int> playerPointsRecord
- Config config
+ PlayerID_t whoWon()
+ void playGroupOfGames()
- void playOneGame()
- void broadcastGameMssg(MessageSender*)
- void registerGameWinnerPoint()
- void updateMatchWinnerStatus()
1
k
GameWorld
- StaticMap map
- Map<PlayerID_t, Duck*> players
- EventsManager eventsManager
- ProjectilesConstroller projectilesConstroller
- CollectablesController collectablesController
- ThrowablesController throwablesController
+ GameSceneDto getSceneDto()
+ void HandleCommand(Command)
+ void QuitPlayer(PlayerID_t)
+ void Update(float)
+ Snapshot GetSnapshot()
+ bool IsOver()
+ PlayerID_t WhoWon()

Figure 1: Diagram of main component classes on the server

Acceptor thread

The server has a central acceptor thread that manages incoming connections from clients and establishes an initial communication channel, in this way, once a connection is accepted, it will allocate and assign communication threads dedicated to said client that follow up on communication with it. So that the acceptor thread can continue focusing mainly on listening on a specific port for new connections, creating and launching communication threads and, incidentally, fulfilling the task of providing access to the existing group of games (processor threads), as well as also. periodically, clean the communication threads created that no longer handle an active connection (communication threads dead).

Communication threads:

As stablished above, once a connection is accepted, the acceptor creates the Sender thread, which establishes the first stage of communication with the client (Binding), which has the purpose of determining which of the existing matches the client wishes to join or whether the client wishes to create a new one. and to make this link concrete, the Match (existing or created) is asked to log In this client in the match, which is, add to the list of participant queues to broadcast the sender queue, as well as give access to the command queue that will process the Match.

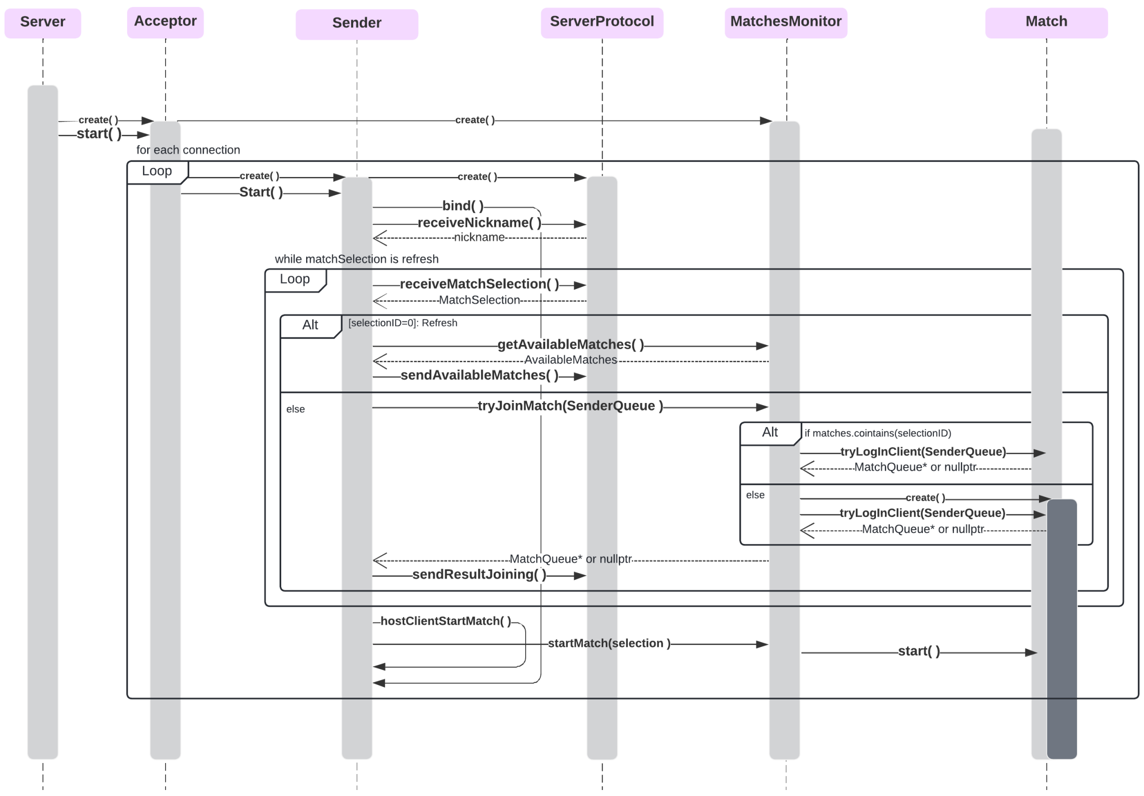


Figure 2: Sequence diagram of binding phase

Once this first linking stage has been successfully completed, a new stage of communications begins, which includes the sending of messages related to the selected Match and the reception of commands/actions to be sent to the same Match.

For this, the sender thread, having access to the command queue of the Match Thread as a result of the previous stage, creates the receiver thread providing access to it and starts this new independent thread.

Now the two communication threads are established into a send loop to send messages, and a receive loop to receive them, this within the duration of the match.

Take a look of this 3 ***Sequence diagram of during-match communications***

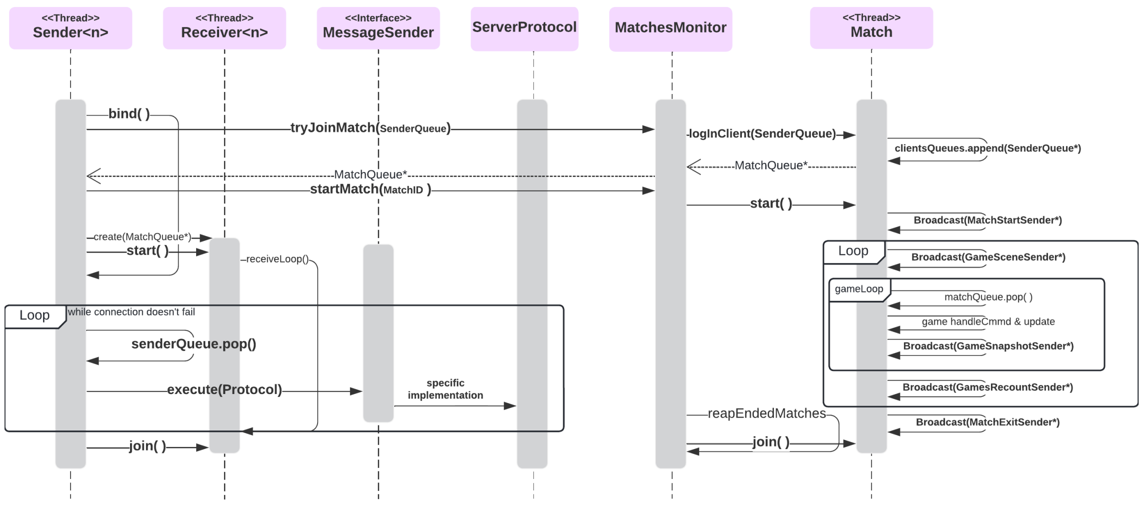


Figure 3: Sequence diagram of during-match communications

In this way, decoupling communication operations that make direct use of blocking sockets from the rest of the server's operations achieves greater modularity and ensures that messages are handled in a timely manner, even when the server is busy processing the logic of the current game, for example applying gravity to the characters in the model, or even other processing which are done between games.

In addition, the communication threads ensure that each client receives updated information about their match without conflicts, using thread-safe structures to avoid concurrency problems, i mean, the communication between threads is done through thread-safe Queues.

Finally, it remains to be clarified how the state of the communication threads and the state of the selected Match are managed in the event of an unexpected disconnection of the client, for which, to begin, it is necessary to make it clear that both the sender thread and the receiver thread communicate with the client through the same socket, in this way, the recv and send operations of the socket will cause the unexpected interruption of both communication loops, with the Receiver thread being responsible for, outside of the receive loop, accessing the Match (through the monitor) to which had linked the client and make the explicit log out request, so that the Match thread can reflect the absence of this client on the model (which consists on removing all related to every character that was playing through that connection) as well as remove the queue of this client from the container of the clients to whom it will be broadcast.

Processing threads:

First, let's establish the main idea that allows us to have multi-matches: Make the main logic of each match be managed by a processor thread (worker), representing an active match instance of the video match in progress.

Now, as previously mentioned, the information on existing matches or even the editing of this set (since clients can decide to create new matches) must be accessed/carried out by the clients' communication threads during the binding stage, which makes it clear that we have: concurrent reading and writing access to the shared resource, which is the set of existing matches.

It is for this reason that it is necessary, in order to avoid Race Conditions, to incorporate into the model a Matches Monitor entity that manages the synchronization of access to the existing worker threads on the server through the use of mutexes, this would be a monitor of the " "set" of worker threads.

Furthermore, since the number of workers is dynamically scaled based on the clients' decisions in the Binding stage and the current server load (it is not allowed to create an unlimited number of macthes), the service must allow each client to join to a processor thread associated with an existing match or request the creation of a new one if he/she have decided to start his/her own match, it ensures that the server can adapt to the dynamics of a multiplayer environment where the number of matches and participants can vary significantly.

This design, where each match is associated with its own processor thread, ensures the previously mentioned goal of isolating active matches from one another. By encapsulating each match within its own active object, the Match Thread, the system effectively processes commands received exclusively from players who have successfully logged into that specific match. Additionally, broadcasts are sent only to those same players, maintaining the integrity and independence of each match.

Finally, we highlight the decoupling of the game logic from the worker thread managing it. In this design, the logic encapsulated within the GameWorld class is unaware of and independent from the threading architecture. This approach ensures flexibility, allowing the described architecture to be replaced with another without impacting the game logic.

Detail of the communication of different types of messages during a match

Since the type of messages that need to be communicated to the client via the protocol depends on the "stage" of the match's state, the message-sending mechanism was implemented using an approach inspired by the double dispatch design pattern. This allows for extensible handling of multiple message types by adding them under a common MessageSender interface to the senders' queue. Each concrete MessageSender has a specific implementation detailing how its content is sent using the protocol.

Refer to the static relationships between the processor threads (which create concrete MessageSenders), the sender threads, and how is the ServerProtocol used , in the 5

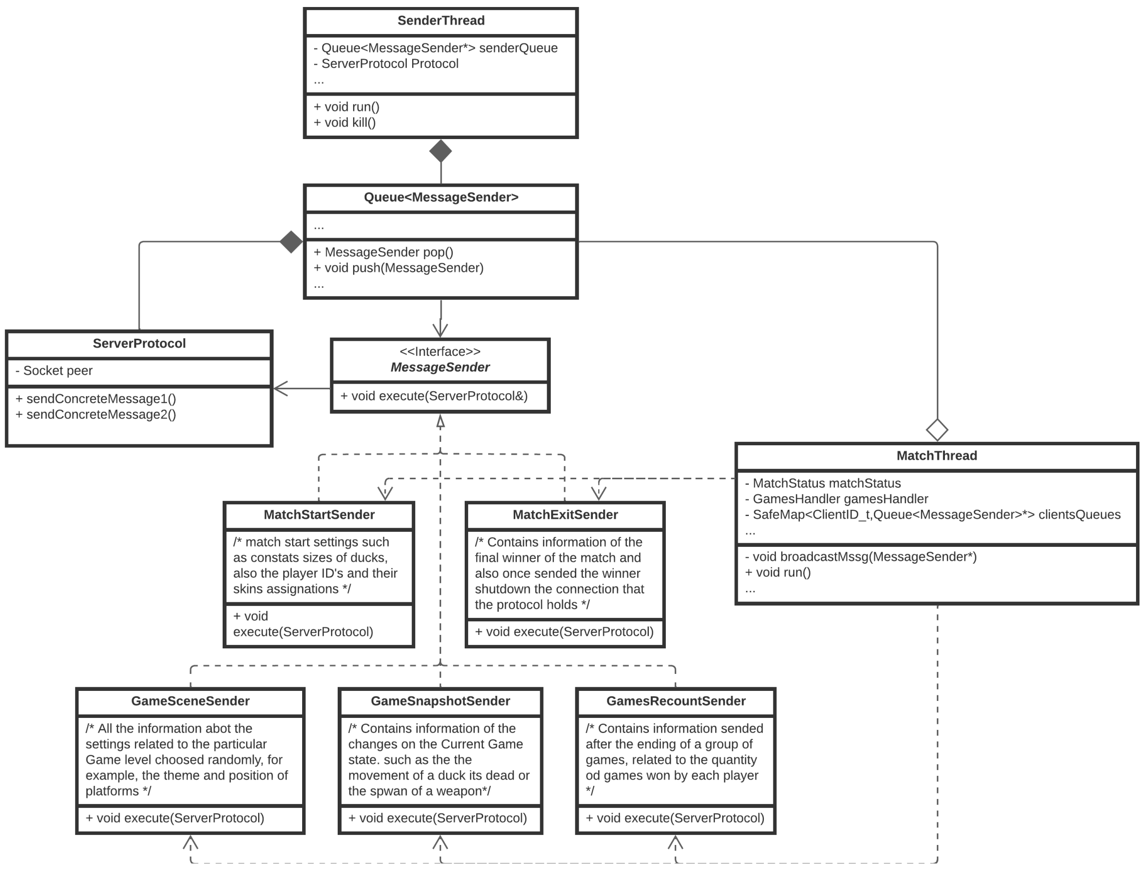

SenderThread
- Queue<MessageSender*> senderQueue
- ServerProtocol Protocol
...
+ void run()
+ void kill()
Queue<MessageSender>
...
+ MessageSender pop()
+ void push(MessageSender)
...
ServerProtocol
- Socket peer
+ sendConcreteMessage1()
+ sendConcreteMessage2()
<<Interface>>
MessageSender
+ void execute(ServerProtocol&)
MatchThread
- MatchStatus matchStatus
- GamesHandler gamesHandler
- SafeMap<ClientID_t,Queue<MessageSender*> clientsQueues
...
- void broadcastMssg(MessageSender*)
+ void run()
...
MatchStartSender
/* match start settings such as constats sizes of ducks, also the player ID's and their skins assignations */
+ void execute(ServerProtocol)
MatchExitSender
/* Contains information of the final winner of the match and also once sended the winner shutdown the connection that the protocol holds */
+ void execute(ServerProtocol)
GameSceneSender
/* All the information abot the settings related to the particular Game level choosed randomly, for example, the theme and position of platforms */
+ void execute(ServerProtocol)
GameSnapshotSender
/* Contains information of the changes on the Current Game state. such as the the movement of a duck its dead or the spwan of a weapon*/
+ void execute(ServerProtocol)
GamesRecountSender
/* Contains information sended after the ending of a group of games, related to the quantity od games won by each player */
+ void execute(ServerProtocol)

Figure 4: Diagram of classes relating to sending different types of messages

Analysis of thread management

In this section, the goal is to provide a final overview of the architecture implemented for the server. To achieve this, we will present a summary of the number of active threads in the server, detailing when they are triggered and how and when their execution terminates.

- **Main server program thread**: This single thread listens for input until the character 'q' is entered via the keyboard, signaling an intent to forcibly shut down the server. Upon this input, the thread will invoke the appropriate procedures to terminate the program cleanly, ensuring no resource leaks.

- **Acceptor thread**:
  - **Quantity of threads**:There is a single acceptor thread.
  - **Launching**: The acceptor thread is started at the beginning of the program: Just after some initializations such as the server configurations read from the corresponding configuration files, the explicit call is made to the server method that starts this thread.
  - **Ending of its execution**: Given that it executes a loop performing blocking operations, a "forceful" mechanism is required to exit the loop (preventing starvation of the main thread during the join operation on the acceptor thread). This is achieved by closing the socket designated for listening to connections, which raises an exception, effectively breaking the loop. This exception is caught and logged as an "unpolite termination" of the accept loop. The process then proceeds to enforce the termination of all ongoing matches. Matches, having access to the queues used by their participants'

sender threads (performing blocking pop operations), handle the cleanup by terminating all threads associated with them (directly or indirectly) and releasing their resources.

Sequentially, the accept loop performs the following actions:

Calls the match monitor to forcefully terminate all active matches (each causing the termination of their participants' communication threads).

Calls the monitor method to clean up resources allocated for all inactive worker threads, which by this point should be all of them. Explicitly invokes the kill method on all communication threads linked to clients that are still in the binding stage. Finalizes resource cleanup for all communication threads in the server.

- **Communication threads**:
  - **Quantity of threads**: Let BC be the number of clients that successfully completed the binding stage to join a match (Bound Clients), and UC the number of clients that did not. Thus, the server maintains the following quantity of comunication threads: UC sender threads (for unbound clients) and BC $\times$ 2 threads (sender and receiver pairs for bound clients).
  - **Launching of the threads**: Sender threads are launched immediately upon accepting a client's connection, while receiver threads are created and launched once the client completes the binding process.
  - **Ending of their executions**: Execution termination for these threads can occur in several cases:

    An unexpected client disconnection—regardless of the communication stage—causes the threads to exit their communication loops due to unsuccessful usages of the socket operations through the protocol.

    Server-triggered termination occurs in two scenarios:
    * **Match conclusion**: When a match ends due to its internal state (e.g., a final winner is determined), the match instance ensures no communication threads remain alive for clients who are neither disconnected nor part of an active match. This is achieved by closing its command queue, broadcasting the final MessageSender (which asynchronously terminates the sender threads after delivering the final message to the clients), and closing the MessageSender queues used by the participants' sender threads.
    * **Server shutdown**: As previously mentioned, the acceptor thread invokes the monitor's method to expedite the termination of all matches, resulting in the same cleanup effect as described above.

- **Match threads**:
  - **Quantity of threads**: As previously detailed, the number of match threads is dynamic, depending on user decisions during the binding stage.
  - **Launching of the threads**: After a client completes the binding stage and their match is determined, the sender thread enters a brief loop to receive messages regarding the client's intent to start the match. The match can only begin if a minimum number of players is reached. Once this condition is satisfied, the match thread is launched, and the match officially starts.
  - **Ending of their executions**: Match threads terminate in two predictable scenarios:
    * Natural match flow: The match concludes when a final winner is determined, following the above described steps.
    * Server shutdown: This triggers the same logic as a natural match conclusion but anticipates the termination process.

Model Layer

In this section, we will present the key aspects of the design implemented to model the game entities, which is strongly rooted in object-oriented programming. Following that, we will provide a brief explanation of how the game snapshots are generated. These snapshots are transmitted over the internet to the players, using a mechanism that heavily relies on event-driven programming.

Principal game entities

As previously mentioned, the server's architecture is designed with a layered structure that houses instances of the model for each game, accessed through the GameWorld class, which serves as the entry point. This class is tasked with handling player commands at a consistent rate, simulating the passage of time within the game world. To achieve this, the class delegates its main distinguishable responsibilities to various controllers. These include the Throwable Items Controller, Collectable Items Controller, Projectiles Controller, and Boxes Controller.

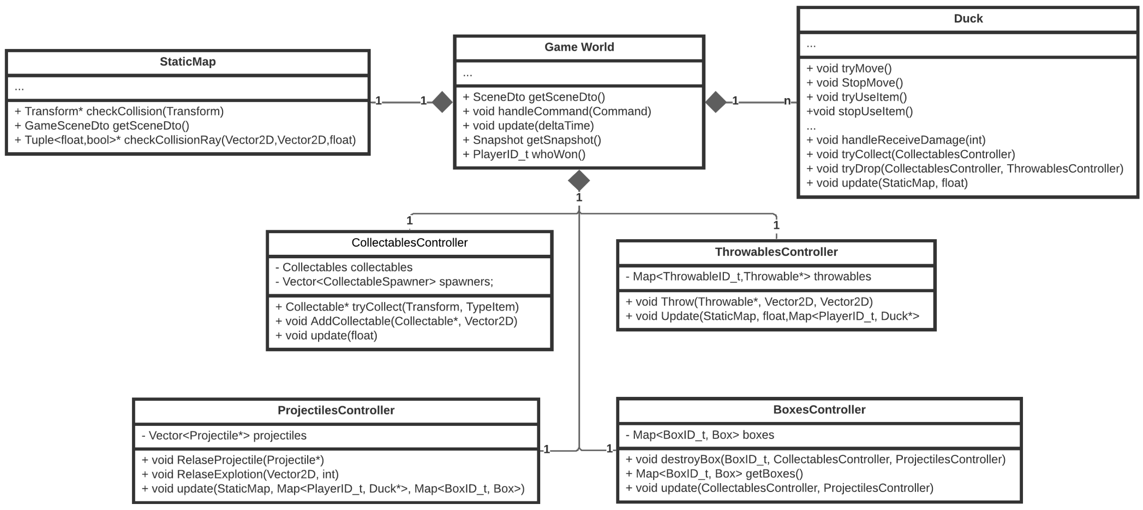
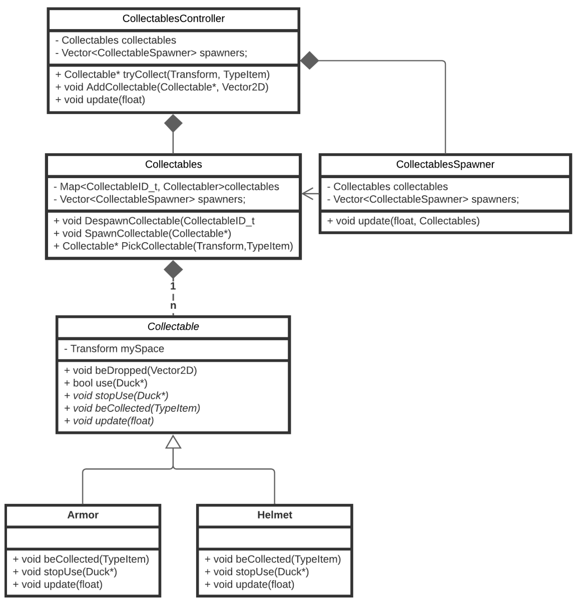

StaticMap
...
+ Transform* checkCollision(Transform)
+ GameSceneDto getSceneDto()
+ Tuple<float,bool>* checkCollisionRay(Vector2D,Vector2D,float)
1
1
Game World
...
+ SceneDto getSceneDto()
+ void handleCommand(Command)
+ void update(deltaTime)
+ Snapshot getSnapshot()
+ PlayerID_t whoWon()
1
n
Duck
...
+ void tryMove()
+ void StopMove()
+ void tryUseItem()
+void stopUseItem()
...
+ void handleReceiveDamage(int)
+ void tryCollect(CollectablesController)
+ void tryDrop(CollectablesController, ThrowablesController)
+ void update(StaticMap, float)
1
1
1
CollectablesController
- Collectables collectables
- Vector<CollectableSpawner> spawners;
+ Collectable* tryCollect(Transform, TypeItem)
+ void AddCollectable(Collectable*, Vector2D)
+ void update(float)
ThrowablesController
- Map<ThrowableID_t,Throwable*> throwables
+ void Throw(Throwable*, Vector2D, Vector2D)
+ void Update(StaticMap, float,Map<PlayerID_t, Duck*>
ProjectilesController
- Vector<Projectile*> projectiles
+ void RelaseProjectile(Projectile*)
+ void RelaseExplotion(Vector2D, int)
+ void update(StaticMap, Map<PlayerID_t, Duck*>, Map<BoxID_t, Box>)
1
1
BoxesController
- Map<BoxID_t, Box> boxes
+ void destroyBox(BoxID_t, CollectablesController, ProjectilesController)
+ Map<BoxID_t, Box> getBoxes()
+ void update(CollectablesController, ProjectilesController)

Figure 5: Entry point of the model and main controllers diagram of classes

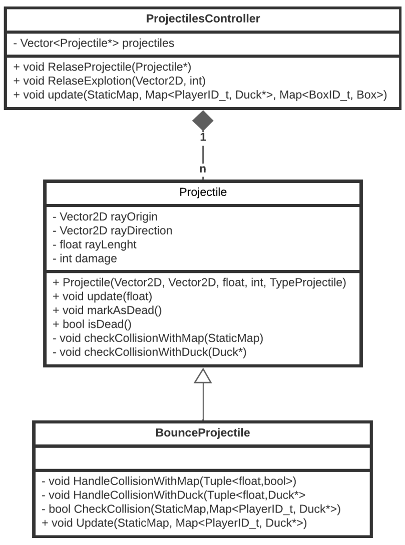
Below are diagrams representing the main participants in a game, along with the players' characters (Ducks). These diagrams highlight the inheritance relationships implemented to abstract the characters from the specific behaviors of certain items, even when those items share the same "type" of interaction. This abstraction is crucial for streamlining command handling, as the commands sent by each character consist only of basic details: who wants to perform the action and what action they want to perform. For instance, "Player with ID xxx wants to Use Item in Hand." Consequently, the implementation of command execution adheres to this logic, abstracted from the specific item being used.

Event-driven programming for snapshots

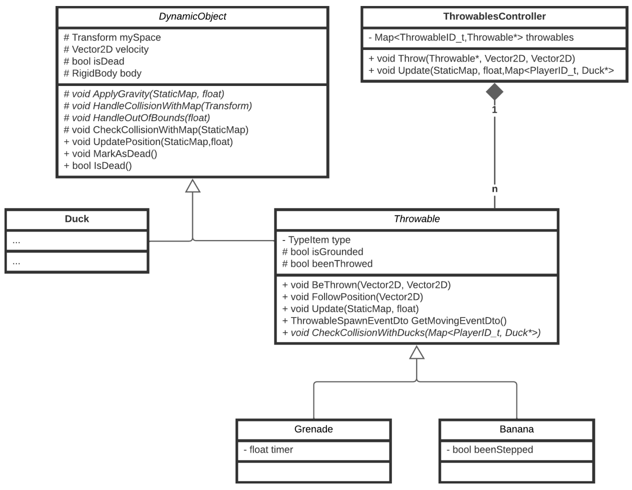
For tracking and communicating in-game changes to the client, we have chosen an event-driven design. This approach ensures that only new information, not already known to the client, is sent. This minimizes the logic's intrusiveness in the game's model while maintaining efficiency and clarity.


CollectablesController
- Collectables collectables
- Vector<CollectableSpawner> spawners;
+ Collectable* tryCollect(Transform, TypeItem)
+ void AddCollectable(Collectable*, Vector2D)
+ void update(float)
Collectables
- Map<CollectableID_t, Collectabler>collectables
- Vector<CollectableSpawner> spawners;
+ void DespawnCollectable(CollectableID_t
+ void SpawnCollectable(Collectable*)
+ Collectable* PickCollectable(Transform,TypeItem)
CollectablesSpawner
- Collectables collectables
- Vector<CollectableSpawner> spawners;
+ void update(float, Collectables)
1
n
Collectable
- Transform mySpace
+ void beDropped(Vector2D)
+ bool use(Duck*)
+ void stopUse(Duck*)
+ void beCollected(TypeItem)
+ void update(float)
Armor
+ void beCollected(TypeItem)
+ void stopUse(Duck*)
+ void update(float)
Helmet
+ void beCollected(TypeItem)
+ void stopUse(Duck*)
+ void update(float)

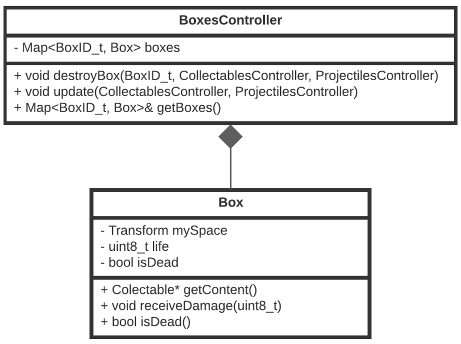
(a) Collectables management and inheritance


ProjectilesController
- Vector<Projectile*> projectiles
+ void RelaseProjectile(Projectile*)
+ void RelaseExplotion(Vector2D, int)
+ void update(StaticMap, Map<PlayerID_t, Duck*>, Map<BoxID_t, Box>)
1
n
Projectile
- Vector2D rayOrigin
- Vector2D rayDirection
- float rayLenght
- int damage
+ Projectile(Vector2D, Vector2D, float, int, TypeProjectile)
+ void update(float)
+ void markAsDead()
+ bool isDead()
- void checkCollisionWithMap(StaticMap)
- void checkCollisionWithDuck(Duck*)
BounceProjectile
- void HandleCollisionWithMap(Tuple<float,bool>)
- void HandleCollisionWithDuck(Tuple<float,Duck*>
- bool CheckCollision(StaticMap,Map<PlayerID_t, Duck*>)
+ void Update(StaticMap, Map<PlayerID_t, Duck*>)

(b) Projectiles management and inheritance


DynamicObject
# Transform mySpace
# Vector2D velocity
# bool isDead
# RigidBody body
# void ApplyGravity(StaticMap, float)
# void HandleCollisionWithMap(Transform)
# void HandleOutOfBounds(float)
# void CheckCollisionWithMap(StaticMap)
+ void UpdatePosition(StaticMap,float)
+ void MarkAsDead()
+ bool IsDead()
ThrowablesController
- Map<ThrowableID_t,Throwable*> throwables
+ void Throw(Throwable*, Vector2D, Vector2D)
+ void Update(StaticMap, float,Map<PlayerID_t, Duck*>
1
n
Duck
...
...
Throwable
- TypeItem type
# bool isGrounded
# bool beenThrowed
+ void BeThrown(Vector2D, Vector2D)
+ void FollowPosition(Vector2D)
+ void Update(StaticMap, float)
+ ThrowableSpawnEventDto GetMovingEventDto()
+ void CheckCollisionWithDucks(Map<PlayerID_t, Duck*>)
Grenade
- float timer
Banana
- bool beenStepped

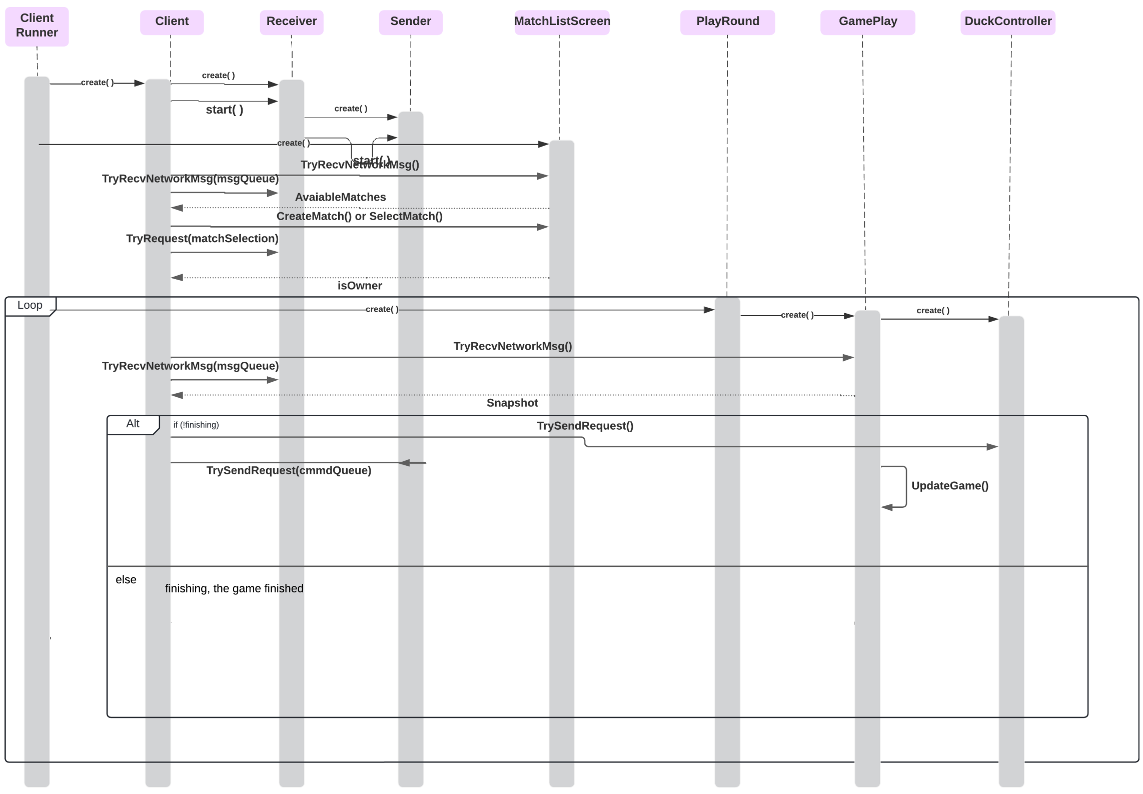
(c) Dynamic Objects: Throwables, throwables management and ducks


BoxesController
- Map<BoxID_t, Box> boxes
+ void destroyBox(BoxID_t, CollectablesController, ProjectilesController)
+ void update(CollectablesController, ProjectilesController)
+ Map<BoxID_t, Box>& getBoxes()
Box
- Transform mySpace
- uint8_t life
- bool isDead
+ Colectable* getContent()
+ void receiveDamage(uint8_t)
+ bool isDead()

(d) Boxes controller

Client-side

In the following section we are going to show the main sequence diagram which shows the main dynamic of multiple threads on the client app: And also we shoe the main abstraction that allows the client received different types of messages: Being this type of messages which the protocol returns, and also the ones that are pushed to a Queue that the main client thread will pop.



Custom communication protocol

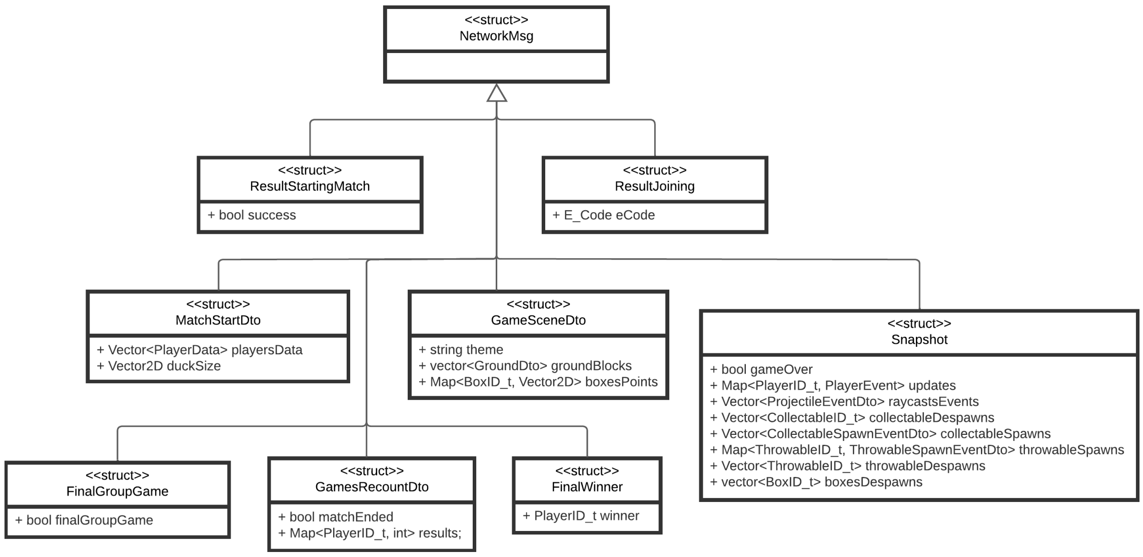
Multiplayer games with complex dynamics, such as this remake of Duck Game, require custom communication protocols to efficiently manage the flow of interactions between the server and clients. This tailored approach is crucial for handling the unique aspects of the game, including dynamic matches, adaptive rules, and sequential groups of levels. Unlike generic protocols, a custom-designed protocol allows the server to dynamically modify the sequence of messages that clients expect during runtime, ensuring a smooth and coherent gaming experience.

Design of the protocol

In the design of the Duck Game protocol, special attention was given to determining when it is appropriate to send specific information to clients. This was essential to prevent cheating and ensure that no client can access details about events, circumstances, or contexts that are intended to remain hidden until they are strictly necessary. By deferring the transmission of such messages, the protocol maintains fairness and preserves the integrity of the game while adapting to its evolving state.

The server's flexibility to adjust the game flow in real-time is a standout feature of this protocol. For instance, if the server detects that only one player remains connected, it can alter the usual rules, even if the standard victory condition—such as achieving a specific number of wins—has not been met. Through this protocol, the server can send messages containing specific flags to guide clients, indicating whether a new set of games will follow or if the current game is the last one.

Additionally, the protocol effectively manages transitions between levels or groups of games. At the conclusion of a group, the server sends messages detailing the number of victories for each player, enabling clients to determine whether there is an overall winner or if more levels are to be played. This structure, built on flags and sequenced messages, ensures that the game flow can be adjusted seamlessly in real-time while safeguarding the fairness and consistency of the experience for all players.


<<struct>> NetworkMsg
<<struct>> ResultStartingMatch
+ bool success
<<struct>> ResultJoining
+ E_Code eCode
<<struct>> MatchStartDto
+ Vector<PlayerData> playersData
+ Vector2D duckSize
<<struct>> GameSceneDto
+ string theme
+ vector<GroundDto> groundBlocks
+ Map<BoxID_t, Vector2D> boxesPoints
<<struct>> Snapshot
+ bool gameOver
+ Map<PlayerID_t, PlayerEvent> updates
+ Vector<ProjectileEventDto> raycastsEvents
+ Vector<CollectableID_t> collectableDespawns
+ Vector<CollectableSpawnEventDto> collectableSpawns
+ Map<ThrowableID_t, ThrowableSpawnEventDto> throwableSpawns
+ Vector<ThrowableID_t> throwableDespawns
+ vector<BoxID_t> boxesDespawns
<<struct>> FinalGroupGame
+ bool finalGroupGame
<<struct>> GamesRecountDto
+ bool matchEnded
+ Map<PlayerID_t, int> results;
<<struct>> FinalWinner
+ PlayerID_t winner

Stages of server-client communication

From the moment a client connects to the server until the match they joined concludes, two main stages of communication were identified. These stages have been referenced and discussed in previous sections. Here, we will formally define these two stages, describe the types of messages exchanged between the client and server during each, and detail the concrete implementation of this communication protocol.

Binding-stage

The first stage covers all communications between the client and the server from the moment a client joins a game until the game, through either an existing or new match, reaches its conclusion. The logic of this stage is relatively simple:

- **Set Client Profile:** The client sends the first message in the communication by submitting its nickname (`string`, following the format established by the `Protocol Assistant`), and the server responds by sending back a unique identification number (2-byte unsigned integer) that allows the client to identify its local connection.
- **Game Selection:**
  a. The next message expected is from the server which must send a list of available matches (matches still waiting for players to start) with their essential information such as the current number of players, the maximum number of players, the name of the game lobby and their own MatchID.
  b. The client must send either `00 00` (2 bytes) to request updated information about the available games or provide a valid MatchID/connection ID as a 2-bytes unsigned number (since the ID of each match is the same as the connection ID of the client that created it) followed by a 1-byte unsigned integer indicating the number of local players that will join through that connection. If the client sends its local ID, the server creates a game "on behalf" of that client. Otherwise, sending a different ID directly corresponds to the match the client intends to join.
  c. If `00 00` was sent previously, the communication flow returns to step `a`. Otherwise, the next message expected is a response from the server regarding the client's attempt to join or create a game. This response is represented by an error code (1-byte unsigned integer), indicating various outcomes (defined in `data/errorCodesJoinMatch`). These codes specify whether the client successfully joined the game or why it failed to do so.

d. If the error code indicates that the client failed to join a game, the communication flow returns to step `b`. If the last code describes no error, the subsequent steps (regarding *binding*) depend on whether the client decided to create or join a game:
   i. If *binding* was not successful, it concludes here.
   ii. If successful, communication flow temporarily halts until the game creator ensures that at least two players (from different connections) join the match. For this, the server sends a `header 06` when `00 00` was sent, indicating insufficient players. A `01` header is sent when the game can officially start.
   iii. The last message of this step indicates whether the *binding* was successful. If not, step `b` repeats.

During-match stage

Protocol Assistant

We will finalize the explanation of the concrete implementation of the protocol by mentioning a key entity, present in both the server and client protocols, which allowed the implementation to be coded in a readable and higher-level format, abstracting the server and client protocols from the precise endianness and size of the data, and also made the variety of messages that the client and server could exchange much more extensible. This entity has the title stablish, is the protocol assistant:

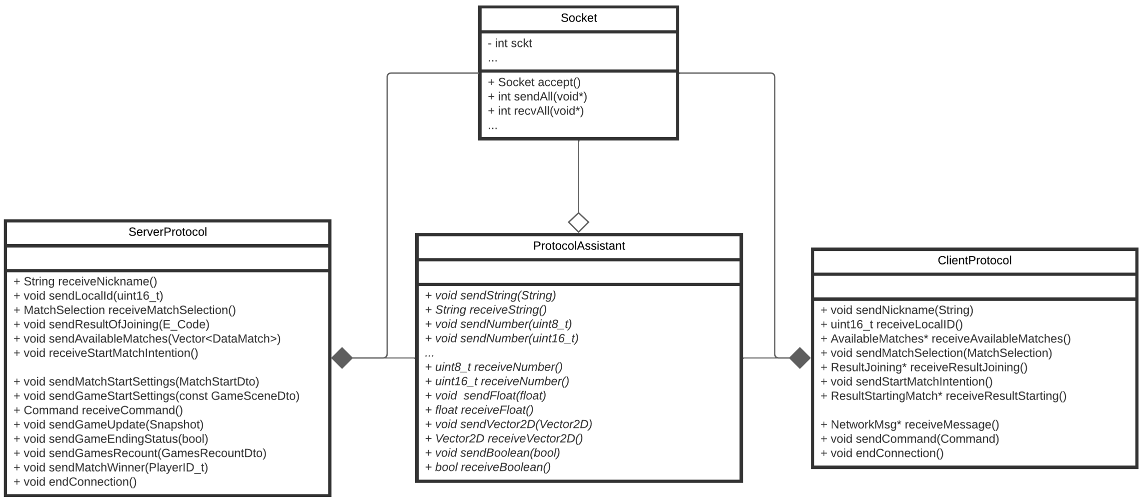

Socket
- int sckt
...
+ Socket accept()
+ int sendAll(void*)
+ int recvAll(void*)
...
ServerProtocol
+ String receiveNickname()
+ void sendLocalId(uint16_t)
+ MatchSelection receiveMatchSelection()
+ void sendResultOfJoining(E_Code)
+ void sendAvailableMatches(Vector<DataMatch>)
+ void receiveStartMatchIntention()
+ void sendMatchStartSettings(MatchStartDto)
+ void sendGameStartSettings(const GameSceneDto)
+ Command receiveCommand()
+ void sendGameUpdate(Snapshot)
+ void sendGameEndingStatus(bool)
+ void sendGamesRecount(GamesRecountDto)
+ void sendMatchWinner(PlayerID_t)
+ void endConnection()
ProtocolAssistant
+ void sendString(String)
+ String receiveString()
+ void sendNumber(uint8_t)
+ void sendNumber(uint16_t)
...
+ uint8_t receiveNumber()
+ uint16_t receiveNumber()
+ void sendFloat(float)
+ float receiveFloat()
+ void sendVector2D(Vector2D)
+ Vector2D receiveVector2D()
+ void sendBoolean(bool)
+ bool receiveBoolean()
ClientProtocol
+ void sendNickname(String)
+ uint16_t receiveLocalID()
+ AvailableMatches* receiveAvailableMatches()
+ void sendMatchSelection(MatchSelection)
+ ResultJoining* receiveResultJoining()
+ void sendStartMatchIntention()
+ ResultStartingMatch* receiveResultStarting()
+ NetworkMsg* receiveMessage()
+ void sendCommand(Command)
+ void endConnection()

Figure 7: interaction between classes: protocol assistant - server protocol - client protocol

As shown in Class Diagram 7 ***Interaction between classes protocolAssistant- ServerProtocol-ClientProtocol*** , the Protocol Assistant encapsulates low-level details such as endian conversion and provides functionality for sending and receiving common data types. In our case, this includes 2D Vectors with decimal components. Additionally, by leveraging the overloaded SendNumber signature and the fact that C++ is a strongly-typed language, it offers polymorphism, allowing different types of numbers (e.g., 1-, 2-, or 4-byte IDs) to be sent using the same method name.

Configurations

Avaiable Levels

This YAML file contains a list of available level names for gameplay. It is used by the configuration class to inform the server about which levels are available and how many there are. Additionally, it is used by the editor to display the list of levels available for editing.

Levels Format

The YAML files that store the level data have a simple format. First, the THEME is declared, representing the background and the type of tiles used for rendering. Next, there are lists of dictionaries that specify different positions: one for player spawn points, another for collectible spawn points, and a third for box positions. Following this, there's a list of strings containing the names of all platforms in the map. Each platform entry includes its center position, size, and which of its edges are drawn.

Basics Grounds

This YAML file contains all the available ground options that can be selected in the editor. Each ground option includes the minimum size and a unique combination of edges. The format of these platforms is identical to the format used when reading a platform from a level YAML file.

Duck Config

This YAML file contains the basic information about a duck, including its mass, size, speed, and health.

Match Config

This YAML file contains all the configuration settings for the match. You can specify how many players are allowed and how many are needed to start the match. Additionally, you can configure the number of rounds required to win the game.

Object Config

This YAML file contains information about different objects:

- **Box:**
  - **Size:** Specifies the dimensions of the box.
  - **Destruction Requirement:** Indicates the number of shots needed to destroy the box.
- **Collectable Objects:**
  - **Size:** Defines the dimensions of the collectable object.
  - **Respawn Time:** Specifies the time interval after which a collectable object reappears.

Themes

This YAML file contains a list of available themes to choose from. Each theme includes information about the selected background, the tiles to be used, the associated image, and the corresponding YAML file that is used for rendering.

Weapon Attributes YAML Description

This YAML file contains information about various weapon attributes. It includes details on:

1. **Cooldowns:** Defines the time between actions or shots. Types include:
   - Medium
   - Minimum
   - Large
   - Grenade explosion time
2. **Damage:** Specifies the damage inflicted by a weapon. Categories include:
   - Minimum
   - Medium
   - Short-range
   - Long-range
3. **Dispersion:** Describes the spread pattern of shots. Levels include:
   - No dispersion
   - Long dispersion
   - Short dispersion
   - Super dispersion
   - Hyper dispersion
4. **Shooting Inclination:** Indicates shooting accuracy or behavior. Modes include:
   - Basic
   - Laser rifle
5. **Projectiles per Shot:** Defines the number of projectiles fired with each shot. Types include:
   - Basic
   - Fragment for grenade
   - Fragment for box

Additionally, the file lists all available weapons. Each weapon is defined by its **name** as the key and includes attributes such as:

- **Ammunition count**
- **Scope or range details**