## Contents

# Multirally Documentation

## I. Intro

Multirally is a real time multiplayer game written in python. The game uses the snapshot interpolation approach for multiplayer. In this approach, the server sends snapshots of the game state to the client for rendering at a fixed rate. The game state mostly consists of the positions of all the objects in the world. This rate is usually slow (10Hz – 128Hz). A lower rate uses less bandwidth and processing power. A higher rate is smoother and more responsive. The client interpolates between each snapshot to smooth the movement of objects in the game. The biggest challenge when programming a multiplayer game is decoupling the networking code from the game code. Programmers should not have to worry about the network code when adding new features to the game (New enemies, Pickups, etc). This project aims to decouple the "netcode" from the game logic code. The methods used in the project are based on the methods used in Valve's source engine. (https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking). Each entity has a client and server version. The client version of the entity has code that is only run on the client. Entities contain "network variables" which are variables that would be shared by both the server and the client and need to be sent over the internet. The network variable is a class of variable that contains various attributes for network communication.

The other main functions in the game engine are client prediction, hit registration lag compensation and delta compression. One big advantage of snapshot interpolation is that the approach is server authoritative, so cheating is difficult.

In shooter multiplayer games, where players are shooting at objects and other players, latency is a big issue. If your client is lagging behind, then you are always shooting at where objects were in the past. To compensate for this, when you shoot at an object, the client sends its latency to the server. Then the server rewinds the game to the snapshot that the client would be seeing. After rewinding, the server checks if you hit the player. The code for the server rewinding can be found in world.py and player.py. When the shoot function is called, the player calls the world rewind function to check if the shot hit.

The client prediction code can be found in client2.py. The client predicts the players current position based on the snapshot from the server and the calculated latency. This is to make the game responsive even when the client has a high latency.

## II. Integrated Design

The game logic and netcode must be integrated in such a way that game logic programmers do not need to worry too much about the networking code. In order to achieve this, a class-based approach is used.

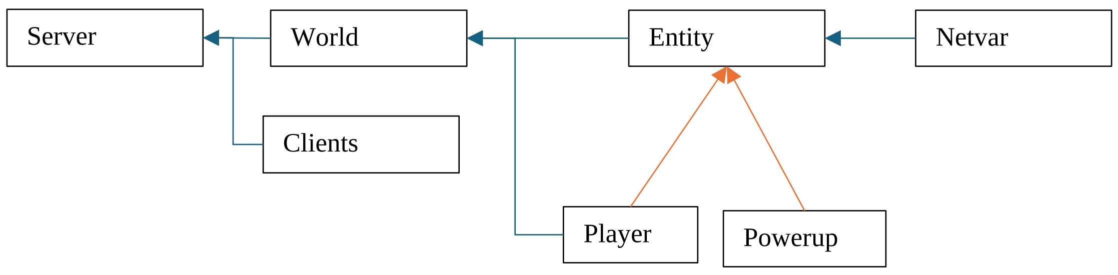

**Figure 1: A simplified view of the server class attributes**

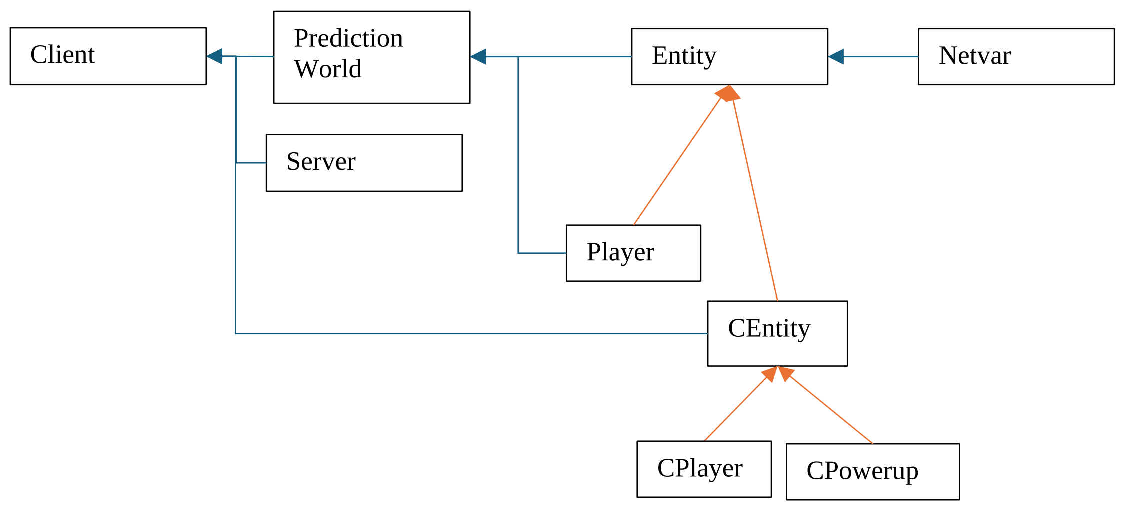

**Figure 2: A simplified view of the client class attributes**

As shown in Figure 1, the server contains a world, which contains entities. Entities are objects with a unique ID that interact with the world. The world is in charge of updating the entities. Entities that interact with the client, for example an entity that makes noise or is drawn on the screen, have a CEntity counterpart. Entities and CEntities contain Network variables, which are to be sent across the network to each client when required. Each network variable for the entity has a unique ID. The entity contains logic that is calculated on the server and the CEntity contains functions that are calculated on the client such as the draw function. The server also contains a socket which connects to the client.

Figure 2 shows the attributes of the client class. It is similar to Figure 1. The key difference is that the world is also used for prediction of the server state and the client contains a dictionary of the CEntities.

## III. Network Variable Class

The network variable is instantiated in both the Entity and CEntity objects and is to be sent over the network to the clients. Essentially, the variable and value is shared between the Entity and CEntity. The *NetworkVar* class takes several arguments to initialise.

### *A. Args*

- *ent*: (Entity) This is the entity that the networkVar object belongs to
- *var*: (integer/float) This is the starting value for the variable
- *_id*: (integer) This is the unique ID value for the networkVar, it should match the corresponding value in the CEntity
- *lerp*: (bool) The client will interpolate the value if true
- *slerp*: (bool) The client will use spherical interpolation if true. This is used when rotation must be smoothed.
- *quantise* (Int) The number of digits to round the value before sending across the network. -1 means full precision.
- *only_send_to_owner* (bool) This variable is useful if the network variable is owned by a player entity. If true, only send this variable to the client for that player. Useful for secret variables only the client should know like health.

### *B. Attributes*

- *var* (integer / float) This is the value stored in the NetworkVar

- *oldvar* (integer / float) This is the previous value of the var, used to check if the var has been updated.

Upon initialisation, the network variable adds itself to the entities data table. The network variable keeps track of if it is updated or not with the *updated* attribute. This tracking is used for the delta compression, so unchanged variables do not need to be sent across the network.

## IV. Entity Class

Entity classes are instantiated in the world object and are the building block of the world. An entity is an object in the game that can interact with the world. Its state can be shared with the client and server by using *NetworkVars*. The entity class does by default does not take arguments to initialise, however some child classes may require arguments.

### A. *Attributes*

- *_id* (int) Unique id number for the entity, this is used to correspond a client entity with a server entity over the network.
- *class_id* (int) Class id number for the entity, this is sent over the network to tell the client to create a new entity with the right subclass. A table corresponding the *class_id* to each subclass is initialised in *entity_table.py*
- *ent_destroyed* (Bool) When set to true, this tells the world to destroy the entity when possible.
- *data_table* (Dict) This is a table containing all the network variables the entity owns. The table corresponds the id of the network variable to the variable object.
- *updated* (bool) Used so that the world can keep track of when the entity state has changed. This is for delta compression. When any of the entities' networkVars update, they set the entity *updated* to *True*.
- *actor* (bool) This variable flags the entity as an actor. Actor entities are entities that have a location in the world co-ordinate space. Actor entities take extra xpos and ypos arguments before they are instantiated.
- *shootable* (bool) The variable flags if the entity is shootable, meaning it must have a bounding box that blocks ray casts and a function to handle getting hit called *get_shot()*. The bounding box must be returned by the *get_collision_bounds()* function.
- *owner* (Client) Used for player entities to keep track of their owners.
- *snapshots* (Dict) For linear interpolation. A history of the state of the object. Only keeps track of network variables. In the form *{_id: [a,b,c..]}*
- *snapshots_xvals* For linear interpolation. Stores the time corresponding to each value in the snapshot history.

### B. *Data tables*

The entities' *data_table* stores the state of the entity. This state must eventually be sent over the network to the client(s).

Sending the original data_table over the network is unideal because some network variables do not need to be sent every frame and some variables do not need a great level of precision. The *prepare_data_table* function returns a new version of the data table. Any network variables that have not changed since the last time the data was sent are removed and variables are rounded to the desired precision based on the *quantise* attribute.

### C. *Apply data table*

#### 1) *Args:*

- *tick* (int) The tick number of the client minus the delay caused by the lerp buffer, essentially the x value of the interpolation.
- *Delay* (int) Lerp requires a delay time so that there can be a buffer. This is the delay (measured in ticks) parameter for the lerp.

The *apply_data_table* function handles linear interpolation. This function is mainly used for client entities (CEntities). The function iterates through all the netvars. If the netvar has lerp enabled, then the function will estimate the y value based on the given "tick" x value using linear interpolation.

### D. *Update*

All entities have an update function. The update function is called each tick during the update phase of the game loop. The world must be input into the update function so that the entity can interact with other objects in the world.

### E. *Client Entity (CEntity)*

The client entity inherits from the server entity and includes extra functions for drawing to the screen.

### F. *Player entity*

The player entity inherits from the server entity but is treated differently to regular entities. The player entity is defined in the *player.py* file. The player's update function takes an extra *actions* argument. The *actions* is a dictionary of the keyboard inputs from the client controlling that player. The player also has some extra required arguments to initialise.

#### 1) *Args:*

- x (float)
- y (float)
- *angle* (float) The angle of the player when it spawns
- *name* (string) The display name for the player
- *owner* (Client Address) The client that possesses the player

## V. WORLD

The world is in charge of the game logic and holds the game state. The world handles updating of the entities, adding new entities and deleting entities. The world handles loading of the level. The world passes on entity information to the server class to send over the network.

### A. *Args:*

- *level_name* (String) The name of the level to load

### B. *Attributes:*

- *player_table* (Dict {PlayerObject : Client Addr}) Table in the format
- *entdict* (Dict {id: Entity}) Table to lookup *Entity* ID number and get entity object
- *static_entities* (List Entity) List of static entities (Entities with no netvars) loaded from the level
- *level* (Dict) Dictionary containing level data in level format. More information on level format found elsewhere in the document
- *client_world* (Bool) Set to true if this world is for client prediction
- *dt* (Float) Delta time for world update
- *snapshots* (List) A history of *entdict*s that remembers the state of entities for a specified amount of frames. This is specified by the *MAX_SNAPSHOT_HISTORY* constant.
- *snapshot_number* (Int) Increments each game loop.
- *create_ents* (list [ents]) List of ents that need were just added to the entdict and need to be sent to the client
- *delete_ents* (list [_id]) List of the ids of the ents that needs to be deleted.
- *collision_sectors* () [NOT IMPLEMENTED]
- collision_sector_size () [NOT IMPLEMENTED]
- *entities_to_spawn* (list [ent]) Entities to spawn in the next frame

### C. *Functions*

#### 1) *add_new_player(client, name)*

*a) Args:*

- client (string) The ip address of the client.
- Name (string) The name of the player

This function spawns in a player for a newly connected client

#### 2) *destroy_player(client)*

*a) Args:*

- *client* (string) The ip address of the client

This function is used when the client disconnects and the player class needs to be removed from the memory.

3) *spawn_entity(entity)*

   a) *Args:*

- *entity* (Entity) The entity object to spawn into the world.

add a new entity in the world on the next frame

4) *add_new_entity(entity)*

   a) *Args:*

- *entity* (Entity) The entity object to add into the world

This function is called during the update loop if entities need to be spawned in. The function assigns the entity object with a unique id number. Then the function adds the entity to the entdict and to the create_ents list. Newly created entities are stored in the create_ents list to track which new ents are not yet sent to the clients.

5) *add_new_static_entity(entity)*

   a) *Args:*

- *entity* (Entity) The static entity object to add into the world

Add a static entity object. Static entities do not need an id number as they are not sent over the network.

6) *destroy_entity(id)*

   a) *Args:*

- *id* (int) The id number for the entity to delete

Deletes the entity from the entdict and appends id to the delete ents list so that it can be sent over the network

7) *Update_existing_entity(entid, newent)*

   a) *Args:*

- *entid* (int) The id number for the entity to change
- *newent* (Entity) The new entity object to replace with

This is an unused function that replaces an entity object. Does not alert the client so the entity object will stay the same on the client end.

8) *get_entity_from_id (id)*

   a) *Args:*

- *id* (int) The id number for the entity to get

Returns the entity belonging to the id number.

9) *get_camera_for_player(client)*

   a) *Args:*

- *client* (string) The address of the client

Returns the id number for the player entity belonging to the client

10) *send_entire_gamestate(client)*

   a) *Args:*

- *client* (string) The address of the client to get the gamestate for

Returns the data table to send to the specified client. The data table is a snapshot of the entire game state.

11) *rewind_to_snapshot(game_state)*

   a) *Args:*

- *game_state* (dict) The snapshot to apply.

Sets the game state according to the state in the snapshot. The function is used to rewind to a specific snapshot in history.

12) *rewind_to_snapshot_index(snapshot_index)*

a) *Args:*

- *snapshot_index* (int) The snapshot number to rewind to.

Sets the game state according to the state in the snapshot index. The function is used to rewind to a specific snapshot in history.

13) *update(client_input_table)*

a) *Args:*

- client_input_table (Dict) The inputs of each client for this frame.

The update function is called every tick. The update function performs several tasks in the following order.

1. The update function first checks if any new entities need to be created by checking the *entities_to_spawn* attribute. If new entities need to be spawned, the *create_new_entity* function is called.
2. The update function iterates through all the entities in the *entdict* and calls their *update* function. If the entity is a Player type, then the respective client's input is passed into the *update* function. Then the entity's *destroyed* attribute is check. If *destroyed* is true, the *destroy_entity* function is called on the entity.
3. The function iterates though all the static entities and calls the *update* function.
4. The data table for each client must be prepared. The data table is a dictionary in JSON format that is to be sent over the internet. The dictionary keys are the id numbers of each entity and the values are all the updated *netvar* ids and values for the entity. The *client_data_table* is the variable that keeps the data table to send to the respective clients.

   To prepare the data table, the function iterates through every entity. The function calls the entity's *prepare_data_table* function which returns a dictionary containing the *netvar* ids and values. If the entity is an actor, then the data table is only required if the entity's x and y position is within the client's screen. Then the entity's data is added as a value to the *data_table* corresponding to the entity id. Then the *data_table* for each client is added to the *client_data_table.* Each client gets a different data table because different things will be sent to each client depending on their position in the world and what variables they are allowed to see.
5. After creating the data tables for each client, a snapshot of the game state is saved to the *snapshots*
6. Entities in the *delete_ents* variable are deleted from the *ent_dict*

The update function returns the *client_data_table* for the server to send.

## VI. SERVER

The server is in charge of accepting connections from clients, keeping track of connected clients, sending / receiving data reliably and calling the world update function. The server uses UDP sockets. Packets are tagged with a packet number for reliability.

### A. *Args:*

- *ip* (string) The ip address for the server socket
- *port* (int) The port that the server socket should run on
- *lvl* (string) The name of the level to initialise the world

### B. *Attributes:*

- MAXPLAYERS (unused)
- *world* (World) The world class that runs the game logic.
- *IP* (string) The ip address for the server socket
- *PORT* (int) The port that the server socket should run on
- *ADDR* (tuple (string, int)) Is the ip and port in a tuple
- *sock* (Socket) The server socket using UDP protocol

- *net_server* (Network) The network class contains lower level functions for parsing JSON data and sending / receiving across the network
- *client_input_table* (Dict {client:inputs}) Tracks inputs sent to the server from the client
- *client_last_action_number* (Dict {client:action}) Tracks the last id number of the last input sent by the client
- *data_table* (Dict) The table stores the data to send to each client as JSON. Keys in the dictionary are the clients. Values are the data tables.
- *new_clients* (List) Stores a list of clients that just connected
- *connected_clients (List)* A list of clients that are connected to the game
- *start_time* (float) The time since the end of the last update
- *tick_number* (int) Counts how many ticks have passed since server start
- *message_number_for_client* (Dict) The latest message number received from that client
- *client_unacked_message* (Dict) Save messages that have not been acked in case the need to be sent later
- *client_last_acked_message* (Dict) Tick number of the last acked message
- read_thread (Thread)

### C. *Functions:*

#### 1) *Update*

The update function runs every tick. The update function first iterates through each connected client. For each newly connected client, the server sends them the entire game state and the id of the player class that the client is in control of.

If the server has a message to send to the client, the server will add the *client_last_action_number*, *message_number_for_client, client_last_acked_message* and *tick_number* to the message. Then the server checks  Any unacked information is added to the message. Then the message is sent using the net_server.send_msg function. After that, the world is updated with the *world.update* function.

#### 2) *read_client_messages*

*a) args:*

- *msg* (DICT) Message to send to client
- *addr* (string) Address of client to send message to

The *read_client_messages* function is run in parallel with the update function. The function is constantly reading incoming messages from any address. If a message from a new address is received, the server assumes that the client wishes to connect to the server. The server tells the world to add a new player entity and adds the client to the *connected_clients* list.

If the server receives a message from a client that is already in the *connected_clients* list, then the message is assumed to be an input and is added to the client input table.

#### 3) *send_message_to_client*

Sends a message to the client and updates the *message_number_for_client* and *client_unacked_messages* variables

#### 4) *only_important_info*

*a) args*

- *msg* (Dict) Input message for client

*b) returns*

Returns the important info from the message.

## VII. CLIENT

The client class is responsible for rendering the world based on the information provided by the server, reading key inputs and sending the input to the server.

### *A. Snapshot interpolation*

The client and server use the snapshot interpolation method, which means the client must keep a buffer of snapshots sent from the server. Then the client interpolates between each value based on the time. Only *Networkvars* with the interpolation flag set to true are interpolated. The interpolation method begins in the client class *process_server_message* function. Each entity holds a buffer containing its state at different time intervals. The *entity.snapshots* attribute holds a dictionary of y values corresponding to the state of each network variable. The *snapshots_xvals* attribute holds a dictionary containing the time each corresponding state was received from the server.

Then, when the next frame is ready to be rendered, the client's *update_entity_table* function is called. This function calls the *entity.apply_data_table* function for each entity in the world. The *entity.apply_data_table* function updates the entities' state based on the current time. This function calculates the interpolation using the numpy *np.interp* function.