# Multiplayer Networked Game Design

Volatile Wings — a team-based aircraft shooter game

Alexis Le Conte
lecontea@helsinki.fi
University of Helsinki

## 1 INTRODUCTION

In this document, we present a system design for a dynamic multiplayer networked game. The system scales up as the game becomes more popular and attracts a larger playerbase, and offers mechanisms for updating assets and game settings. The system strives to be reliable enough to provide a good experience to the players.

First, we introduce the game and its mechanics. The rest of this design document presents the system architecture — the system's components and the interactions between these components — and explains how it answers scalability and game-specific reliability requirements.

## 2 GAME DESIGN

### 2.1 Overview and General Rules

Volatile Wings is a team-based aircraft shooter game. Each player controls a fighter aircraft and tries to reach the highest possible score by destroying enemy players' vehicles. There are only two teams, but some original game mechanics allow players to leave their team and play against all, and even to switch teams!

Volatile Wings is a top-down two-dimensional game: the scene is viewed from above. Players' aircraft maneuver on an aerial battlefield that has no obstacles, but finite dimensions.

The game never ends — players can join and quit the game at any time — the only goal is to achieve the highest score possible. New players will be assigned to the team that currently has the less players.

When players get shot down, they automatically reappear somewhere on the border of the battlefield and can continue playing.

### 2.2 Scoring Mechanics

Players gain score points doing the following:

- Damaging an enemy player's aircraft
- Destroying an enemy player's aircraft

There is also a friendly fire risk: players lose score points if they damage or destroy an ally aircraft. Some scoring mechanics are designed to promote team-play:

- Players receive bonus points for helping allies destroying enemy aircraft
- Players receive bonus points for defending allies that are being attacked by enemy aircraft

### 2.3 Team-Switching Mechanics

Other game mechanics introduce more complex gameplay possibilities:

- Players who destroy too many friendly aircraft become renegades, and are no longer part of a team
- Players who belong to any team receive bonus points for damaging or destroying renegade players' aircraft
- Renegades receive more points for damaging or destroying any aircraft than regular players
- If a renegade player destroys several vehicles of the same team in a row, he will be assigned to the other team and immediately lose his renegade status

### 2.4 Aircraft Damage Model

To keep things simple, aircraft have a simple damage model: each player has a health bar, and if they get hit by a machine-gun bullet they will lose health points. The amount of damage inflicted by a bullet may depend on some parameters such as the angle of penetration and the distance travelled by the bullet before hitting the aircraft.

## 3 SYSTEM COMPONENTS AND INTERACTIONS

The system, modeled in figure 1, has a client-server architecture. This is mainly to prevent clients from cheating, as the authoritative room server instances will run a master game simulation, enforce game rules and correct clients deviations. If a clients cheats, it will only affect his experience, but not that of other players.

### 3.1 Roles of the Components

*3.1.1 Clients.* Clients are the player endpoints. They connect to a game server instance (a Room Server), accept player input and send it to the game server, run their own simulation of the game from the data sent by the game server and display the result on the screen.

*3.1.2 Main Server.* Its role is to act as a Directory Server for Clients to discover Room Servers. Clients willing to play the game will first connect to the Main Server and query the list of available game instances (Room Servers). Information such as the room name, the number of players and connection details (Room Server port and IP address) will be sent to the Client. The Main Server is also the origin for game updates distribution (this will be discussed in the System Update Mechanisms section).

*3.1.3 Room Server.* A Room Server runs a single game instance. It has to register to a Main Server to be advertised to players and fetch game updates. Regular health-checks will confirm Room Servers availability to the Main Server as well as send metrics such as the number of players connected to the game instance. A Room Server can accept up to a certain number of players at any point in time, and this limit can be set dynamically (this will be discussed in the Scalability section).

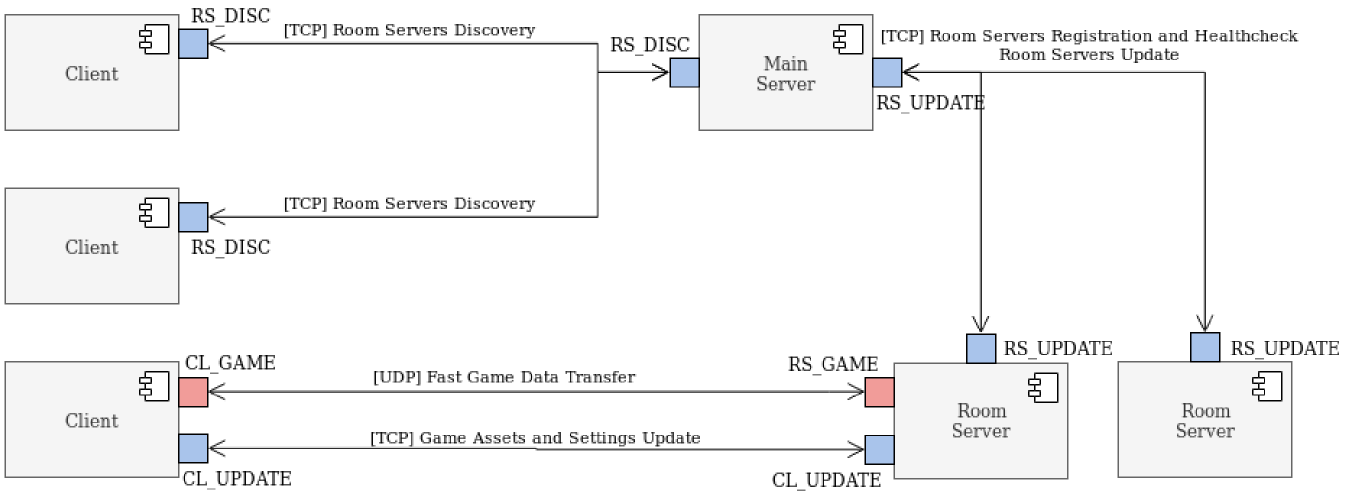
Client
RS_DISC
[TCP] Room Servers Discovery
Client
RS_DISC
[TCP] Room Servers Discovery
RS_DISC
Main Server
[TCP] Room Servers Registration and Healthcheck
Room Servers Update
RS_UPDATE
Client
CL_GAME
[UDP] Fast Game Data Transfer
RS_GAME
CL_UPDATE
[TCP] Game Assets and Settings Update
CL_UPDATE
RS_UPDATE
Room Server
RS_UPDATE
Room Server

**Figure 1: System components and interactions**

## 3.2 Interactions between Components

In this section, we define the exact interactions between the components, that is the kind of data that will be sent through network connections defined in figure 1. These interactions have their own requirements that we use to choose relevant reliability mechanisms, described in section Network Reliability Mechanisms.

*3.2.1 Room Servers Discovery.* Here the Client does not need to send information to the Main Server. Upon connection, the server replies with a list of Room Servers. The amount of data to send to the client depends on the number of Room Server instances running, but should not get very large (64 bytes of data per room should be more than enough). There is no extreme latency nor throughput requirement here, it is acceptable to wait a couple of seconds for connecting to the game and retrieving a room list. However, it is required that the data sent and received (room name, server port and IP address) is correct — no packet corruption. Discovery updates need to be sent every few seconds by the Main Server to the Client.

*3.2.2 Room Servers Update.* At startup, Room Servers will connect to the Main Server and check for updates. Before sending game data, the Room Server and Main Server will exchange hashes of their game assets to identify if they have the same version of game files. As game assets and configuration will be very lightweight files, there is no need for a fast update. Moreover, Room Server startup is a mechanism that is totally hidden from the end users, and will not affect them in any way. However, it is required that the data is transmitted entirely and without corruption!

*3.2.3 Room Servers Registration and Healthcheck.* Once updated, Room Servers need to signal to the Main Server that they are still alive and send some game-specific data such as the number of active players in the game instance. Then again, there is no extreme latency nor throughput requirements here, it does not matter if some packets get lost, as long as enough of them make it through. If the Main Server detects that a Room Server is unhealthy — meaning no packets were received from the Room Server since a certain number of seconds — it will simply stop advertising it to the clients until the Room Server sends new packets.

*3.2.4 Client-Side Game Assets and Settings Update.* Upon connection to a Room Server, Clients may need to update their game settings and Assets according to the room's game version. This is very similar to Room Server Updates. As we have already discussed, game assets are lightweight files, and it is acceptable to wait a few seconds for connection and download time when connecting to a game server. However, downloaded files should be an exact copy of the Room Server files.

*3.2.5 Fast Game Data Transfer.* This is the most demanding data flow in terms of reliability, because it needs to be as fast as possible while allowing only a small margin of error for packet loss. The Client will only send very small packets (a few bytes of data) containing its keyboard state and a timestamp. Keyboard state packets are send every time the player presses or releases a key, or when a timeout is reached. These packets are used by the server to update its master simulation, and also to check which Clients are alive. It is absolutely necessary that this packets arrive to the Room Server to ensure a good player experience, because if keyboard states are lost, the system will not account for player actions. The system needs to reduce the risk that a critical keyboard state packet — typically a packet containing an ephemeral order, such as opening fire — is lost. What's more, if keyboard state packets are lost, it is not conceivable to re-send them, because packet-loss detection is very time-expensive, and packet re-transmission would require simulating back several game steps to take into account the lost packet, which may decrease the overall performance and affect negatively every other player, as their actions depend on what they saw of other player's actions.

The Room Server sends back to the Clients acknowledgement of keyboard states along with the master simulation's version of the player state to allow client-side reconciliation if both simulations diverge. The Room Server also sends game objects updates (aircraft and bullets positions) as long as scoreboard updates and notifications. These packets allow more room for error: the player will probably not notice low packet loss rates, because game objects positions will be predicted and interpolated client-side and thus can be seamlessly corrected.

## 4 SYSTEM SCALABILITY

In this section, we discuss how the system can be scaled up when the number of player increases.

As it was mentionned above, Room Servers define a maximum number of players at any point in time. This is to satisfy game-specific requirements:

- Players should be able to travel through the entire battlefield (from one end to the other) in less than one minute
- The battlefield must be large enough to avoid concentrating all the action in a single point, allowing for more complex gameplay
- As this is a score-based game, all players should appear on the scoreboard and be visible on the in-game scoreboard screen at the same time

These constraints limit the number of players within a game instance. The maximum number of players should not exceed 16, which seems a reasonable enough trade-off decision.

As the number of player within a game instance increases, the performance of this particular instance can decrease. More players make more game objects to manage: more collision detection, and more game objects updates to broadcast to the Clients. The two potential bottlenecks of the system are the simulation time, which grows at least linearly in terms of the number of players (depending on the data-structures used for collision detection, it could even grow in $O(n \times k)$ where $n$ is the number of players and $k$ is the maximum number of in-flight bullets), and the outgoing network bandwidth used to send game state updates to the Clients. However, with appropriate data-structures like quad-trees, only 16 players, and with policies aiming at limiting the maximum number of in-flight bullets (limiting the fire-rate and bullets time-to-live), this should hopefully not be noticed by the Clients.

The system provides another way to scale the system horizontally, allowing more players to play — but not in the same room. It is indeed possible to start new Room Servers as the number of player increases. Game instances are run in independent Room Server instances, which may create some overhead, but allows to easily distribute rooms on different physical servers, depending on the available CPU and network bandwidth resources. Also, game instances are resilient and not affected by failures in other application-level parts of the system once a game is launched and players are connected.

Because a client-server architecture can only allow a finite number of connections, it is only possible to scale the system up to a certain extent. At some point, money will be the answer to scaling the system up.

## 5 SYSTEM UPDATE MECHANISMS

The Main Server stores game assets — mainly a set of few textures to draw aircraft and background textures — and updates of these files need to be done first on the Main Server. When a Room Server connects to the Main Server, it sends a list of its own asset files hashes. The Main Server is then able to compare these hashes with its own file hashes. Sending hashes allows to spare network bandwidth, even though game assets are very lightweight files. Updates are then propagated to the Room Server if needed. The same mechanism holds when Clients connect to Room Servers.

This mechanism, along with the horizontal scalability possibilities, allows to update the entire system without service interruption. First, an update is done on the Main Server. Then, a new Room Server is deployed, and fetches the update. Meanwhile, an already running Room Server is scaled down: the maximum number of players it accepts is dynamically set to 0, which means players can continue playing but no one new can join, so the room will eventually die and can be killed. This can be repeated until all Room Servers are up to date. A more aggressive policy can be to immediately scale down all servers and progressively reintroduce up-to-date servers.

## 6 NETWORK RELIABILITY MECHANISMS

In this section, we take a look at all the network interactions between the system components, and how end-to-end reliability trade-offs will be implemented to satisfy their specific constraints and requirements, that were elicited in section Interactions between Components.

### 6.1 Room Servers Discovery

For this purpose, as the data flow can be relatively slow but must not suffer corruptions, TCP is a good candidate. It satisfies all requirements and is simpler to implement than UDP, because no additional reliability mechanisms are needed on top of TCP. The main drawback of this design is that if a really huge number of players connect at the same time to the Main Server, it will not be able to scale up and open so many TCP connections. However, as players will spend more time playing than looking at the list of available rooms, we can close TCP connections as soon as a Client has chosen a Room Server.

### 6.2 Room Servers and Client Update

The discussion here is quite similar. TCP will allow for a reliable file transfer and satisfies our requirements. However, application-level end-to-end reliability must be implemented on top of TCP, to ensure that files were correctly written on the file system. This can be done by reading the updated files, computing their hash values and comparing them the to hash values of the files sent by the server.

### 6.3 Room Server Registration and Healthcheck

As a TCP connection already exists between the Master Server and the Room Server, it seems sensible to keep it and use it for health-checks. Moreover, TCP connections allow trivial disconnection detection at the application level. The downside of this approach is that many TCP ports will be open on the side of the Master Server, and it will only be able to scale up this much. However, if the number of players grows so much that this becomes an issue, a scalability solution will be to add one more level to servers hierarchy: one central server could administrate several Main Servers and become the starting point for system updates... And Main Servers could be hidden behind a load-balancer.

## 6.4 Fast Game Data Transfer

We have already argued that packets re-transmission was not an option for Client-Room Server communications, because simulating back several steps to improve the experience of one player was expensive (particularly in the case of high packet loss), and would not be fair to all the other players.

For transmissions from the Room Server to the Client, re-transmissions are not relevant, because the Room Server will send game state updates regularly anyways, and as entities positions are interpolated client-side (see section Entity Interpolation), packet loss should not be too noticeable because entities positions can be smoothly corrected.

Therefore, the system strives to reduce the probability of unfortunate events such as packet loss and data corruption. Detecting packet loss will be of little use, but detecting data corruption and correcting data is a priority.

There is no point in playing a networked multiplayer over a network that suffers of too much packet loss. Here, we decide that the maximum acceptable network packet loss for the system is 30%.

*6.4.1 Client to Room Server.* For communications from Client to Room Server, throughput utilization is clearly not the bottleneck: only very small packets are sent. We can afford a high overhead to ensure that keyboard state packets will reach the server, and will be readable. As a reminder, keyboard state packets contain information about keys pressed as well as a timestamp which acts as a packet id.

Because of the small size of the packets, data corruption is already unlikely, but we need forward-error-correction mechanisms to recover our data. We will use a combination of triple-redundancy (if all you have is a hammer, everything looks like a nail[1], but in this case we can actually afford such overhead, it is easy to implement and it is efficient enough!) to prevent from packet loss, and Hamming Codes ($Hamming(7,4)$) to be able to correct corrupted data even from a single received packet. According to previous course work, triple redundancy is enough to recover 95% of the original data in the case of 30% random network loss rate, which is satisfactory.

We decide not to implement packet ordering on the server side because it is time-expensive in the event of packet loss, and argue that packet ordering can effectively be avoided here. The server will simply discard any packet reading a timestamp that is before the timestamp of the last received packet, with one exception to be discussed later. Consider we require that the Client application runs at 30 frames per second, and that each frame displayed to the screen is the result of a single simulation. There is a 33 milliseconds delay between each simulation, which means there is also a 33 milliseconds delay between the moments we register and send each keyboard state change. If we send keyboard states several times over the network, there is a high chance that the fastest keyboard state packet of simulation $n$ will arrive before the fastest packet of simulation $n + 1$ if the round-trip-time is not too long. In the rare event a packet is discarded because of bad ordering, this should not affect the player too much because we expect consecutive keyboard states to be similar in most cases. Furthermore, missing one keyboard state is only critical if the packet contains very ephemeral information such as the order of opening fire, which is solved by the exception mentioned above: if a packet received in the wrong order contains such orders, they will be executed by the server in the next simulation.

*6.4.2 Room Server to Client.* Like Room Servers, Clients will not implement packets ordering. Indeed, it is too expensive to wait for delayed packet, and in any case, it is already too late to simulate step $n$ for a game object if data for step $n + 1$ has already been received. Therefore, Clients will ignore game objects updates for packets sent with too early timestamps.

We discuss here how to minimize the impact of packet loss from the Client's point of view. First, game object updates will be sent in several packets to minimize the loss of data in case of packet loss or data corruption, even though this design will introduce more overhead due to protocol headers space. Second, against data corruption, packets will use the Reed-Solomon block codes, where a symbol is a byte of data. At this point, we consider that a few application messages are buffered before being sent to the Clients. The different application messages to send will have to be concatenated inside Reed-Solomon blocks. As blocks have a fixed size, this will result in even more overhead because some padding may be needed. This can be partly solved using an optimization algorithm to dispatch application messages cleverly into fixed-size blocks, such that padding is minimized.

## 7 GAME-LEVEL RELIABILITY

Articles [2] and [1] discuss well-known mechanisms to ensure networked games reliability and compensating latency issues. The system's design is inspired by such mechanisms.

## 7.1 Client-Side Prediction

To ensure a smooth player experience and high system responsiveness, the player aircraft position will be predicted according to player input. Keyboard-state packet feedback from the Room Server will contain the server-side position of the player at the time the keyboard-state was sent, allowing to correct the client-side player position in case of deviation (which can happen if packets were lost for instance).

## 7.2 Entity Interpolation

Every entity that is not controlled by the player will be interpolated client-side, to avoid instability issues in case of packet loss. The entities will be displayed according to linearly interpolated data received from steps $n - 1$ to $n$.

## REFERENCES

[1] VALVE development community. [n. d.]. Source Multiplayer Networking. Retrieved October 4, 2019 from https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking#Lag_compensation

[2] Gabriel Gambetta. [n. d.]. Fast-Paced Multiplayer. Retrieved October 4, 2019 from https://www.gabrielgambetta.com/client-server-game-architecture.html

[1]Abraham Maslow, Law of the Instrument