# Network Utilisation

## Network Architecture:

**Server**

Receive Packet

|

Calculate Logic

|

Send Packet to player(s)

**Client**

Send Packet

Receive Packet

|

Apply Packet Logic

The diagram shows a high level approach to how the client and server communicate. The lines indicate the direction of the packets (Who shall be sending and who shall be receiving them).

## Component Interactions:

There are several kinds of messages which are exchanged between the server and the client. This is because we have several client states which are looking for different kinds of packets. This section will cover these different states.

The first state is the menu state, where the client is looking for information regarding chat messages and session statuses. The client will be awaiting for the server to send information when a new session has been created, when a session status has updated or when another user has sent a message. At this stage the user will send packets when they have entered a message, joined or hosted a server.

Next we have the session state where clients have joined a game session with other players, or hosted one that players can join. The client will be looking for packets regarding the other connected players character selections or waiting for the sessions host to begin the game. The client will send a message when they’ve selected a character, left the session or (If they’re the host) begun the session.

Finally we have the in-game state, where the host has begun the session and the players are actually playing the game. Clients will be looking for messages when other players have moved or been hit. The user will also be sending the same kinds of messages that they are expected to receive.

There are also other handling packets for when a client has disconnected from the session or passing players details when a new client has connected.

# Server

## Data Structures

The server holds several objects for managing the different players and sessions. This section will go over each of the objects and explain their purpose.

First we have the Client object which, as the name suggests, holds details regarding each individual client. These details include their id, name, x and y positions and hit percentage. The server will use these details to distinguish which client their communicating with and manage their overall gameplay experience.

Next we have the Session object which holds information about individual sessions, such as their name, state, selected level and an array of all the connected players. The server needs these details to manage the overall session state and cut down the time required to find players who are part of that session.

The Message object is, as the name suggests, designed to hold messages which have been exchanged in the “chat” screen. It simply holds the name of the player who sent the messaged and the body of the text.

Next the Logic object holds several functions which calculate in-game events, such as checking collision, registering hit damage and calculating knockback effect. This object was designed to give the developers the ability to tweak how effective certain actions are or add additional logic later on into the game.

Finally we have the Debug object which runs several small tests on the Logic object (Although more could be added in the future) that get run whenever a client starts the server. This was designed as a mini Unit Testing class so developers could easily expand or alter how testing the server works.

## Code Structure

*Document the overall flow of the code and where it is located within the source file(s). Provide a table with one row for each of the functions that you have created within your server.*

<https://esdoc.org/>

Paste directory link to live Github page:

<https://malithium.github.io/Smashville/doc/index.html>

# Implementation Evaluation

*Provide a summary of the strengths and weaknesses of the technologies (platforms/frameworks etc) that you used to implement your game.*

Strength:

One major weakness of Smashville is that it’s partially Peer-to-Peer. This is because the program couldn’t run Phaser HEADLESS on the server side due to Node.js executing socket.io from the terminal (Not from a browser). There are better ways that this could be handled, such as using an engine rather than the Phaser library (Unity for example). The server could also have additional checks to prevent players “hacking” or passing across “broken” values. Due to the nature of these issues though there would need to be a lot of checks required to counter this.

Strength:

Weakness:

Strength:

Weakness: