Hunt, Hill, Langham, Lonsway
CSCE 315-503

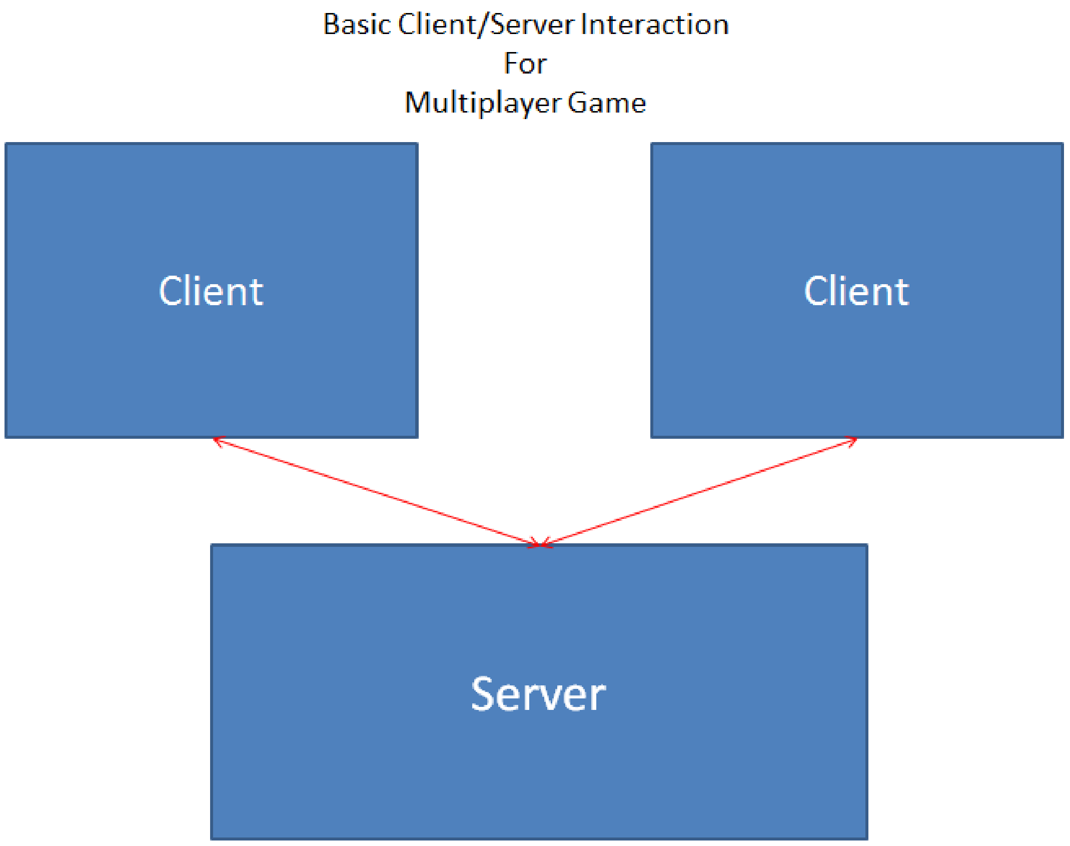
Space Escape Functional Specification

Purpose and Features:

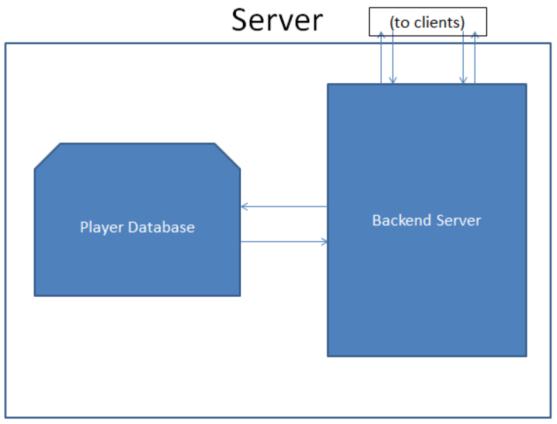
This design describes an original game which attempts to provide all of the aspects of meaningful play. It is designed as an online, multiplayer racing game with a unique control system application. The style targets a retro-aesthetic appeal, which is complimented by a simplistic control scheme. The game will support two modes: race and challenge. In race mode, the screen will scroll based on the movement of the player towards the edge of the screen. However, in challenge mode, the screen automatically scrolls at a predetermined speed regardless of the position of the player on the track. The player will then have to able to keep with the screen's speed.

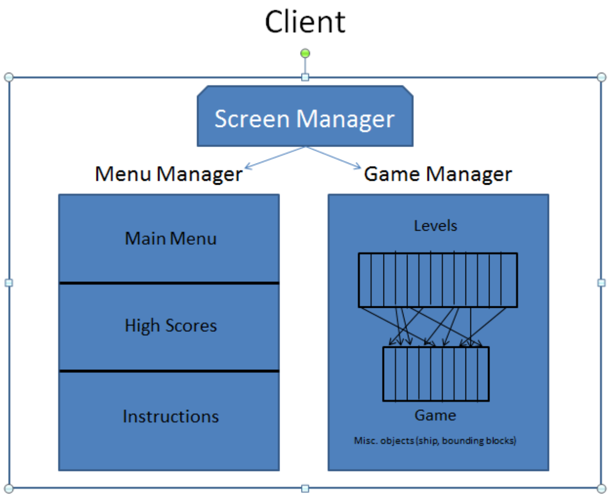
High-level Overview:

High Level Entities:



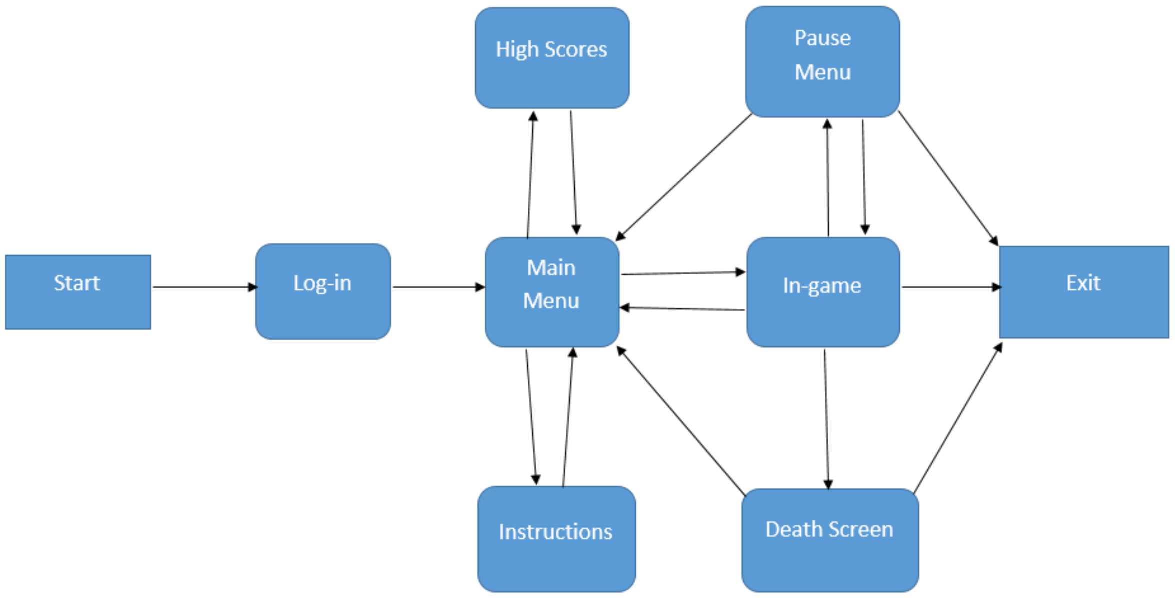
In the game, we have:





The primary high-level interaction in this design is between a client and a server, since the game is run in a multiplayer environment. Each player runs a local client on their own webpage, which communicates with a server, which in turn sends updates to another client, and vice versa. Furthermore, each of the high level entities can be further subdivided (especially the client side).

Flowchart:



Client:

The client will be written in JavaScript, CSS, and HTML. Originally, the plan was to use the Easeljs framework for the graphics portion of the game. However, after developing a very simply prototype and doing some research, it was decided that it would be easier and more straightforward just to use the built-in graphics instead of an external library.

This client consists of a number of different aspects, which in turn create the game:

- Stage: hosts all of the local graphics. This is the graphical framework that we use to draw on the canvas.
- Screen Manager: passes control to the appropriate control manager with parameterized function calls. The two managers that it can pass control to are the Menu Manager, and the Game Manager
    - Menu Manager: Contains all information about menus, and has access to the server to retrieve high score information. Deals with any screen involving text only. Handles mouse inputs for these screens as well. Queries server for online players.
    - Game Manager: Contains all information about the ongoing game. Included in the game manager is a list of all possible levels, as well as a list of the current game set up. The game manager contains a final tier of objects.
        - Level: A numerical description of the various block-obstacle arrangements.
        - Ship: The ship itself which the player controls, along with its various variables.
        - Level Layout: The randomly generated list of levels that make up a course, as well as the endpoints.
        - Collision Detector: An object to create simple collision detection. ***Notice that this is hosted by each client separately.***
        - Animator: Object to manipulate objects on the screen, mainly being the ship. Handles events such as a crash and respawn.

Server:

The server is a much smaller application. The server implementation will be achieved using JavaScript along with node.js and socket.io. It will be hosted on the Computer Science Department's compute server, although it could just as easily be hosted on a local machine. The server's main job will be mostly of relaying messages between the two clients. The main objects in the server are:

- Database: This is a database containing all of the users of the game, and their rating and password protection information.
- Server State Manager: A manager that keeps track of who is online, and who is making game requests. This is useful when a client wants to know what games are available.
- Server Game Manager: A manager that relays updates from a client to another client, and keeps track of the current game layout.

Finally, we have one shared object, being an Update object. An update is the JavaScript object which the client sends to/receives from the server. It contains all of the necessary information that the other client needs to track the progress of the other player. The details of the Update object are described in the client/server communication section below.

Low-level Overview:

Client:

In describing the low level design, let us start with the client side. As stated earlier, the client primarily consists of a stage, which is where all of the objects are drawn. Simply stated, it is the parent to everything else. A stage is the basic graphical object, from which objects can be attached and detached in order to display and remove them. The stage does little more than this, to provide an entity on which we can draw. The functions that are used with the stage are:

- detach (o) – detaches an object from the screen.
- attach (o) – attaches an object from the screen.

Screen Manager:

From there, we move on to the screen manager. This is really no more than a state which describes if we are in game mode, or in menu mode. This is useful for when the server makes calls to the client, as well as applying event handlers and other similar aspects. But in essence, this is really just a design abstraction that makes the flow easier to understand.

Menu Manager:

So first, let us address the menu manager. Whenever the menu manager is opened, a mouse event handler is added to handle clicks on the text, which redirect the user. The text will be drawn again by using built-in graphics, mainly the Canvas Text objects. These can be easily added to the stage and drawn. The main control flow of the screen manager is run through the function:

function displayScreen(index);

The indices of possible screens (highscore, main menu, instructions) are pre-declared as constants in an enumeration-style beforehand. Therefore, this function can be used to call any menu (i.e., at startup the main function can call displayScreen(MAIN_MENU) to easily and clearly start the main menu). All other function calls through the menu manager are through the graphical framework, via creating text objects to display and attaching/detaching them from the stage.

From the menu manager, the program can also call startGame(type, host), which starts a game using enumeration-style constants: SINGLE_TIME, SINGLE_CHALLENGE, MULTI_RACE, MULTI_CHALLENGE. This is passed as a parameter, which determines which type of levels to generate. The host is a Boolean value, which is passed for multiplayer game modes, and determines if the current client or the other client will generate the levels. This is not to be confused as saying that the client itself hosts the gameplay.

There will also be sub-menus for difficulty and level of gravity. These will be small boxes that pop up over the main menu. There will be three difficulty levels and three different strengths of gravity. When one of these menus is up, the main menu may be visible in the background, but won’t be clickable.

Game Manager:

Now we move on to the game manager itself. This is where the game mechanics and objects begin to come into play. The game manager primarily contains information on level design and the current layout of the levels. In addition, it contains functionalities for collision detection, as well as the ship itself, and the top and bottom bounding blocks. As explained above, the menu manager transfers control to the game manager by calling function startGame(type, host). If the client is the host, it generates a level layout which will be explained later, and if not, then the client can do nothing but wait for the server to send the game layout generated by the other client.

*As a quick side note, the rationale behind generating the level design in the client is thus; the client already needs a level generator for single player, unless we want the client to have communication with the server while in single player mode. So from there, it would be repetitive and unnecessary to put this in the server code as well.

It should also be noted that the game will include a short tutorial mode. This will launch automatically when a user first plays the game. It will give the player a brief overview of the game mechanics and controls, allow them to get accustomed to them, and then start them in a time trial game. The goal of the tutorial mode is to quickly introduce the player to the game before allowing them to jump right into the action. After the first time, the tutorial will then be available from the instructions screen off the main menu.

Level Layout:

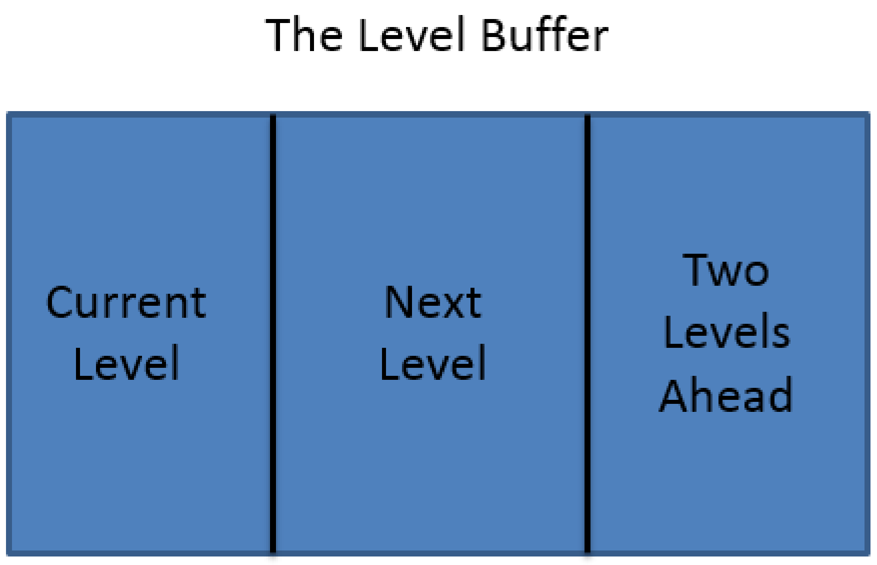
To generate a level layout in race/time trial mode, the program calls:

function generateLevelLayout() – this returns an array of levels.

> Taking this a step deeper, levels themselves are hardcoded in as predesigned arrays of arranged blocks. Blocks themselves are merely just an array of points (later we explain the addition of lines to these blocks for collision detection). So for each level, we have a function stored in an array called:
>
> function generateLevelX(index) - The index describes how many levels are before it in the level layout. Practically, that means that for each block, we need to offset it by (levelWidth)*(index). levelWidth is constant across all level designs to aid in the simplicity of the design, and furthermore it is set to the width of the screen (which is set to a constant value of 1000).

So stepping back up to the function at hand, generateLevelLayout(), the functionality takes a series of non-repetitive random numbers in the set [0..NUM_LEVELS-1]. We then call the function from the array of functions described above, call it, and add the returned level to the array of levels which make up the level layout. This is repeated from 1 to LAYOUT_LENGTH which describes the number of levels which make up a layout. The layout also stores the value representing the finish line. This will be used to determine a winner, as well as to stop scrolling once the finish line is on the screen.

Generating a level layout in challenge mode is slightly more complicated. Because we do not know when the game will end, we cannot compile the list of levels beforehand. Instead, what the program does is constantly store 3 levels. One is the current level, and then it buffers the next two. Any time the ship passes to the next level, a new level is generated. The low level design of this is an array of 3 levels called levelBuffer, and an integer describing the current level. Thus, whatever we pass as the new level, we generate a new buffered level, store it in levelBuffer[currentLevel], and then increment current level.



Finally, levels can of course be repetitive, but will not be allowed to repeat more than once every three levels. This is very easy, since when generating a new level in challenge mode, it can simply check that the level generated is not one already stored in the levelBuffer. The functional specification of this would be:

> function generateChallengeLevel(index) – index again describes how many levels have become before it. Thus the level generation occurs in the same way, and this function just picks a random level. This repeats until that level is one not in the buffer, and then returns the compiled level.

Ship and Collision Detection

The next aspect of the gameManager is the ship itself, which is stored using a number of values to determine velocity and position. Specifically, the ship contains an array of three points for the vertices of the triangle, a rotation value, and numerical velocities in the X and Y directions.

A primary place where these values are used is in the CollisionDetector. This is a specified static object which simply tells the client if the ship has collided with any blocks. The program does this computation by passing in all of the blocks from the current level, as well as the bounding blocks. The collision detection system then uses intersections to see if the player has died. In order to speed this process up, Line variables are added to blocks, which contain information describing a line, with a slope and intercept. We then call:

function intersect(l1, l2) – returns the point of intersection between the two lines.

Of course, for every frame we do have to compute the line functions for the ship object, however this turns out to be a relatively cheap operation. However, if we determine to use this sort of linear representation of a line, we need to take into account vertical lines. This is done using a quick check in the collision detector to see if the x values are the same. If they are, the convention we made is to simply have that constant x value stored in the line’s b-value. Thus, this actually speeds up the intersection calculation, since we just plug in that constant X-Value to the ship’s linear function.

Animator:

A final aspect of the game manager is what we separate as the animator. Because we do the visuals primarily using vector graphics, we want to add some manual animations upon death. So first, whenever a player hits something, the ship spawns three separate lines, which move away from the point of collision in what looks like the ship splitting apart. Following a specified amount of time (say 1 second), the lines move back to the beginning of the current level and reform the spaceship. Before calling the animation object, the ship controls are turned off (in single player, the user can still pause). So this serves the dual purpose of being a neat user interface, as well as providing game mechanics to give a further time penalty for death. The function used to call this is:

function animate(deathPoint, respawnPoint, xV, yV) – deathPoint and respawnPoint are fairly self-explanatory; just the point location of death and respawn. The other two values describe the velocity the ship had upon collision, which can be used to make a smoother animation. Notice that this will also have to have access to the stage in order to do the screen scrolling (scroll back to beginning of level). There will also be sounds to accompany these animations.

Client Representation of Other Player:

A parallel ship and level layout is maintained for the opposing player, and is drawn along with the client players' ship and level layout.

In terms of what the game manager does:

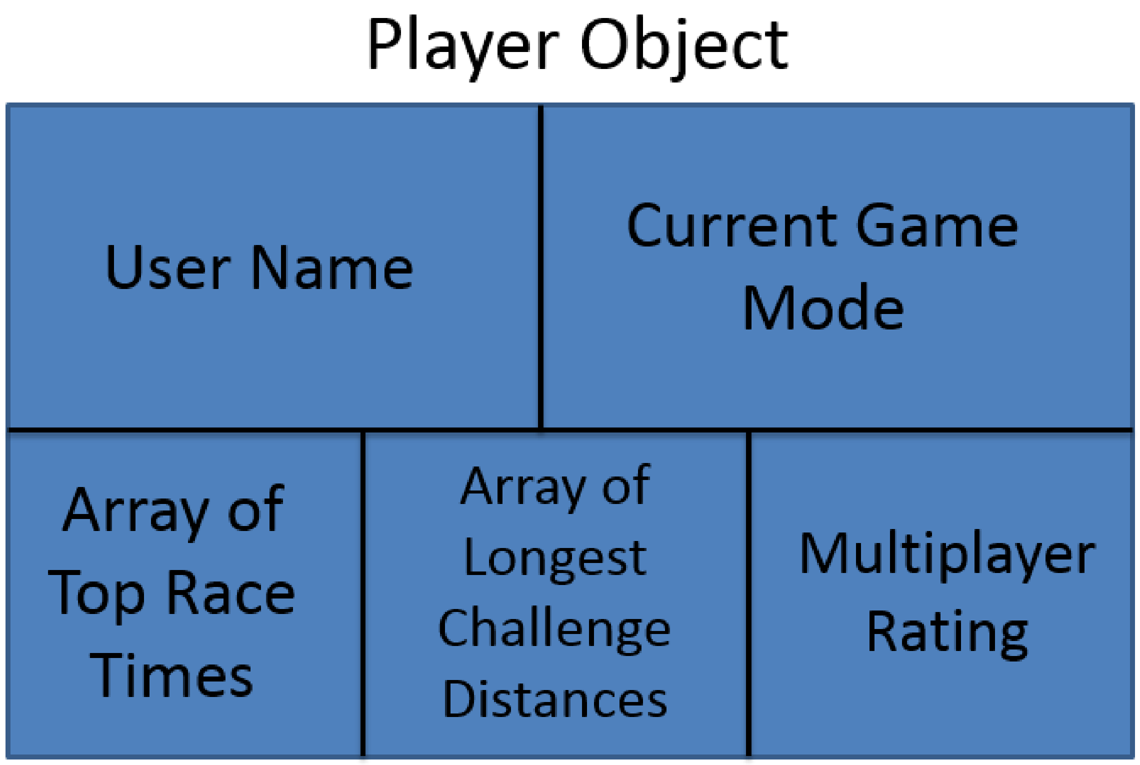
function update() – updates the current state based on currently pressed keys and position of the ship in relation to the layout. This is where scrolling is handled. If dead, most of this functionality is disabled while the animation takes place. This also updates and sends an Update object to the server for the other client.

function handleUpdate() – an event handler for an incoming update request. This then updates values for the opposing player's ship to be drawn at the next update().

Server:

Next we move on to the server design. The server has a much simpler design, and is primarily used to communicate between clients. As stated above, the three main aspects of the server are: database, server state manager, and server game manager.

The database is a fairly simple design. It is simply an array of player objects. The data members of a player object are: username, current game mode, top race times (for each time trial difficulty and for online race), longest challenge-mode distances and multiplayer rating. For the best times and distances, the top five of each are stored in their own arrays.



The client communicates down to access a certain player's profile to either receive permission to log in as that player, or rather to just query the database for top ratings. A user profile is linked to a specific player object; therefore, it contains all of the data members discussed above and shown in the player object diagram.

After log-in, the client sends game mode information to the server every time the player selects a new game mode. This is helpful when the server needs to store high score information about the client. Also, when a user selects a multiplayer game mode, they are placed in a waiting list associated with that mode until there is an opponent ready. Lastly, the database receives updates of new scores and multiplayer rating updates when a client finishes a race.

The server state manager is slightly more complex. This is an object that keeps track of all users currently logged into the system, and tracks who is trying to begin each type of game. The primary use of this is to show the user if and who is currently waiting for an online multiplayer game. The design of this is simply through two arrays, one for race and one for challenge. When a user selects the corresponding mode, the server checks the correct array (which serves as a waiting list). If the array is empty, then the user is added to it; however, if there is already someone in the waiting list, then that person is removed and the two users then begin a game.

Finally, we have the server game manager. The server creates one of these for each game, and the server then has constant contact with each of the target clients. The server receives updates from each client, and passes them on to the other. The functionality of this is fairly simple, and is run by installing an event listener to a web socket for a function:

function handleUpdate(update, client) – takes the update, and sends it to the other client. If the hasWon value in update is true, the server updates players' rating values and closes the current server game manager.

While this is the main function, we also need a way to transfer the level design mentioned earlier between clients, so chronologically, the server first installs a handler:

function handleLevelDesign(level, client) – sends the level design to other client

After this function is called, the event handler is removed, and the handleUpdate function is added.

Client/Server Communication:

Finally, we need to address the Update object, being the main degree of communication between the clients. An Update is a list of all applicable information that one client needs to know about the other. The information we store is as follows:

shipX – X position of opposing ship

shipY – Y position of opposing ship

shipRotation – rotation of opposing ship

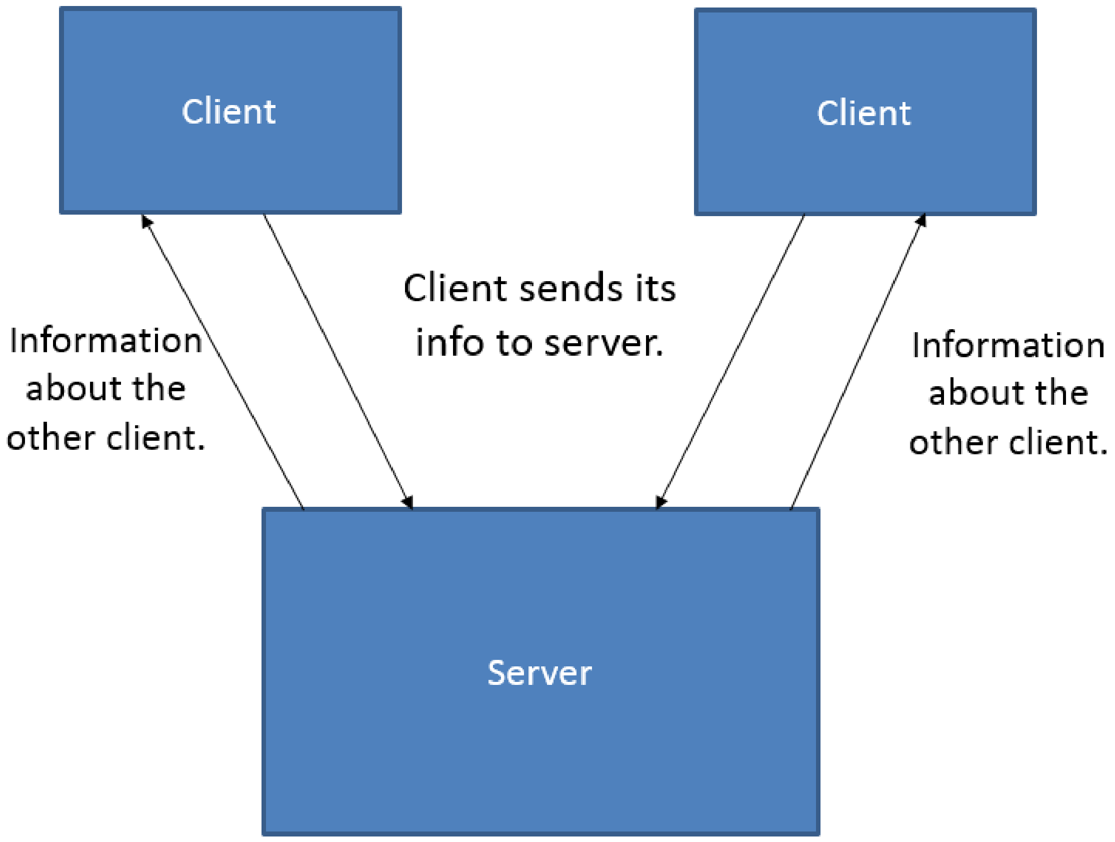
screenOffset – the X scrolling offset of the opposing client. Otherwise, we would need to calculate if the other client has stopped scrolling or not.

level – this is the current level of the opposing player. This aids in the drawing of the other player's position on the bottom half of a client screen.

There are two other significant messages that are sent during a game. The first, wonGame, is sent only during a multiplayer race. This tells the server and the other client that this player has won. Therefore, the game is over. Conversely, in challenge mode, the lostGame message is sent when a player dies. This allows the game to end properly, the other player to know that he/she has won, and for the server to make the proper updates.

There will also be some other communication between clients and the server, some of which has already been referenced previous in this document. These include:

- Log-in/User information
- Game mode choice
- High Score information
    - High scores can be requested as either overall scores or just a player’s personal best
- Progress percentage when player crashes


Client
Client
Client sends its info to server.
Information about the other client.
Information about the other client.
Server

All communication will be done using websockets (via socket.io and node.js). The data itself will be passed using the built-in JSON functionality. This will not require any additional libraries because both client and server will be written in JavaScript.