[redacted]
[redacted]
INF 221, Software Architecture
Homework Part 1 & 2
2/13/2013

Lunar Lander Architectural Modelling

Program overview: Our version of a "Lunar Lander" game that is **significant**, **multi-player**, and **networked** is conceived in the vein of a 2-dimensonal, top-down viewpoint multiplayer spaceship shooter game (working title: "*Super Asteroid Melee*").

This version of Lunar Lander is designed for a minimum of **2** players, and a maximum of **4** players. Players take on the role of Spaceship captains. Each Spaceship captain's objective is to use his ship's weapons to fire at and destroy the other captains' ships—the last man standing is the winner. To make the game more **"interesting"**, the game field will also be host to floating Asteroids, which the Spaceship captains must attempt to maneuver around while battling. Crashing into an Asteroid will damage the player's ship, bringing them closer to death.

Spaceships will be capable of firing multiple weapons, and will also have shield capabilities to protect them from other players' weapons and Asteroid collisions. Random powerup nodes will spawn on the battlefield during play. Picking up a powerup will reward the captain with a weapons upgrade, shield bonus, or extra health points (depending on the type of powerup).

Additional difficulty is introduced by adding Transportation vehicles (the *Magic School Bus*, for example). Transportation vehicles have no weapons and cannot damage players' spaceships. However, shooting an innocent Transportation vehicle will penalize the offending captain, resulting in a loss of health points. This element is primarily intended to keep players from constantly "mashing" their firing button(s).

Architecture description languages used: We choose to use the **xADL** architecture description language (because it was required), the **ACME** architecture description language (because it has better tool support than AADL—which was suggested for use, but we found to be difficult to work with), and **UML** (because we are most familiar with it).

High-level design decisions: We chose to model our Lunar Lander game as a browser-based game. This was based on our desire to create something fun, that also takes advantage of some of the newest capabilities of HTML5; something we both wanted to learn more about while doing this project.

As a browser-based game, Lunar Lander uses the architectural *style* of client-server. One of the defining characteristics of our initial architectural model is the choice to use the **WebSocket** API for communication (aka as a *connector*) between client(s) and server. WebSocket is an API specified by the W3C that enables Web pages to use the WebSocket protocol (defined by the IETF) for two-way communication with a remote host.[1]

[1] http://dev.w3.org/html5/websockets/

The motivation for using WebSockets is to avoid the traditional overhead associated with the request/response paradigm of HTTP, which generally makes it unsuitable for low latency applications with a significantly real-time component. In simple terms, a WebSocket creates a persistent connection between client and server, through which both parties can start sending data at any time.

Other strategies for dealing with latency-sensitive applications on the Web include **polling**, sending HTTP requests at regular intervals, or potentially one **long polling** session, where the client opens an HTTP connection to the server, and the server keeps the connection open until sending a response. However, any such workarounds that use HTTP suffer from one common limitation: the HTTP headers they use, which frequently contain unnecessary or duplicate information. WebSockets are superior in this regard; once the client establishes a connection with the server, information can be sent bi-directionally over a single "socket", full-duplex connection.

Using WebSockets as connectors therefore leads our architecture to also have elements of the *event-based* architectural style; JavaScript event handlers are used to determine the appropriate response to messages sent between the client and server, which may be received at any time while the connection is established.

xADL model: The architecture model in xADL is shown below. There are two main views: the first is a high-level view of the whole Web Application as a top-level component. Inside this component is a subcomponent showing the components of Server, and browser-based clients, and the connectors between them, the WebSocket connections. This view is very similar to the version of Lunar Lander shown in Figure 4-12 of the textbook; both are multiplayer applications in the client-server style.

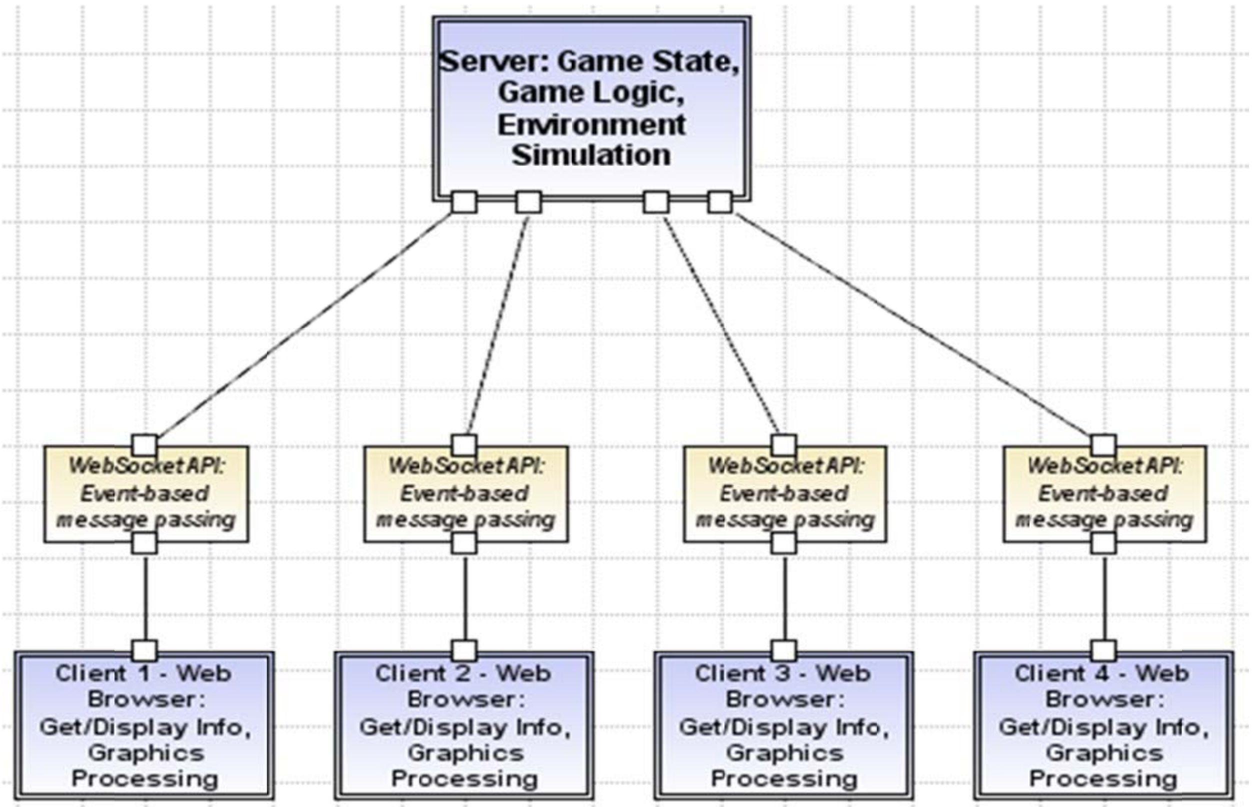

Server: Game State, Game Logic, Environment Simulation
WebSocket API: Event-based message passing
WebSocket API: Event-based message passing
WebSocket API: Event-based message passing
WebSocket API: Event-based message passing
Client 1 - Web Browser: Get/Display Info, Graphics Processing
Client 2 - Web Browser: Get/Display Info, Graphics Processing
Client 3 - Web Browser: Get/Display Info, Graphics Processing
Client 4 - Web Browser: Get/Display Info, Graphics Processing

Figure 1. Lunar Lander Web Application Modelled in xADL

This diagram shows the division of responsibilities for the game: the Server maintains the game state, implements the game logic, and simulates the battlefield environment (including the creation and movement of Asteroids and Transportation vehicles). The Clients get and display information from the Server, and are responsible for performing the graphics processing and presentation. Clients also send messages to the Server regarding their movement and weapon firing commands using the same WebSocket connection.

The second xADL view models the WebSocket connection between Client and Server (there will be between 2-4 such connections when the game is being played, depending on the number of players who have joined the game).

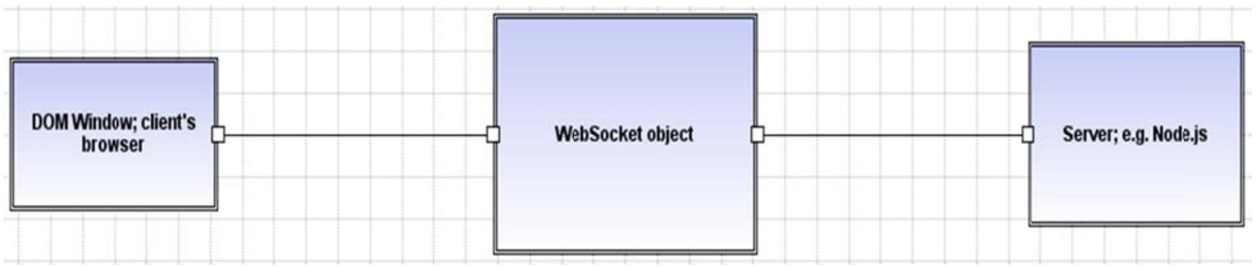


Figure 2. WebSocket connection modeled in xADL

This diagram shows the nature of the connection between client and server. The client is a DOM Window object, available to JavaScript code on a WebSocket-compliant browser (i.e. recent versions of Chrome, Firefox, IE 10 or Safari 6).[2]

The client initiates a connection by creating a WebSocket object, which will automatically attempt to open the connection to the server.[3] If the connection is successful, the client will be able to send asynchronous messages to the server. These messages can be sent as `strings, Blobs, or ArrayBuffers`. Complex messages can be created by encoding them using JSON.

The server's replies are handled via the event-based WebSocket API by delivering a "message" event to the client's `onmessage` function. Two-way communication can thus continue until the client closes the WebSocket connection by calling `close()`.

UML Model: The architecture model in UML is shown below. There are three UML views: the Class Diagram view, which shows the elements of the game software in an Object-Oriented paradigm, a Sequence Diagram, which shows the sequence of operations and interactions between components involved in starting and playing the game, and a Use Case Diagram, which shows at a glance the high-level functionality provided by the software.

The UML Class Diagram is shown below. Note the SpaceShip class, which represents the game object controlled by players (Spaceship captains) via the browser. The GameEngine class represents the

[2] https://developer.mozilla.org/en-US/docs/WebSockets
[3] https://developer.mozilla.org/en-US/docs/WebSockets/Writing_WebSocket_client_applications

portion of the Server responsible for maintaining game state and logic. Part of the environment simulation includes the Asteroid and Transportation classes. The BoosterPack class and its' specializations represent the powerup items that spaceship captains may pick up to boost their firepower and defenses; the Armory class models the stockpile of weapons a player has accumulated.

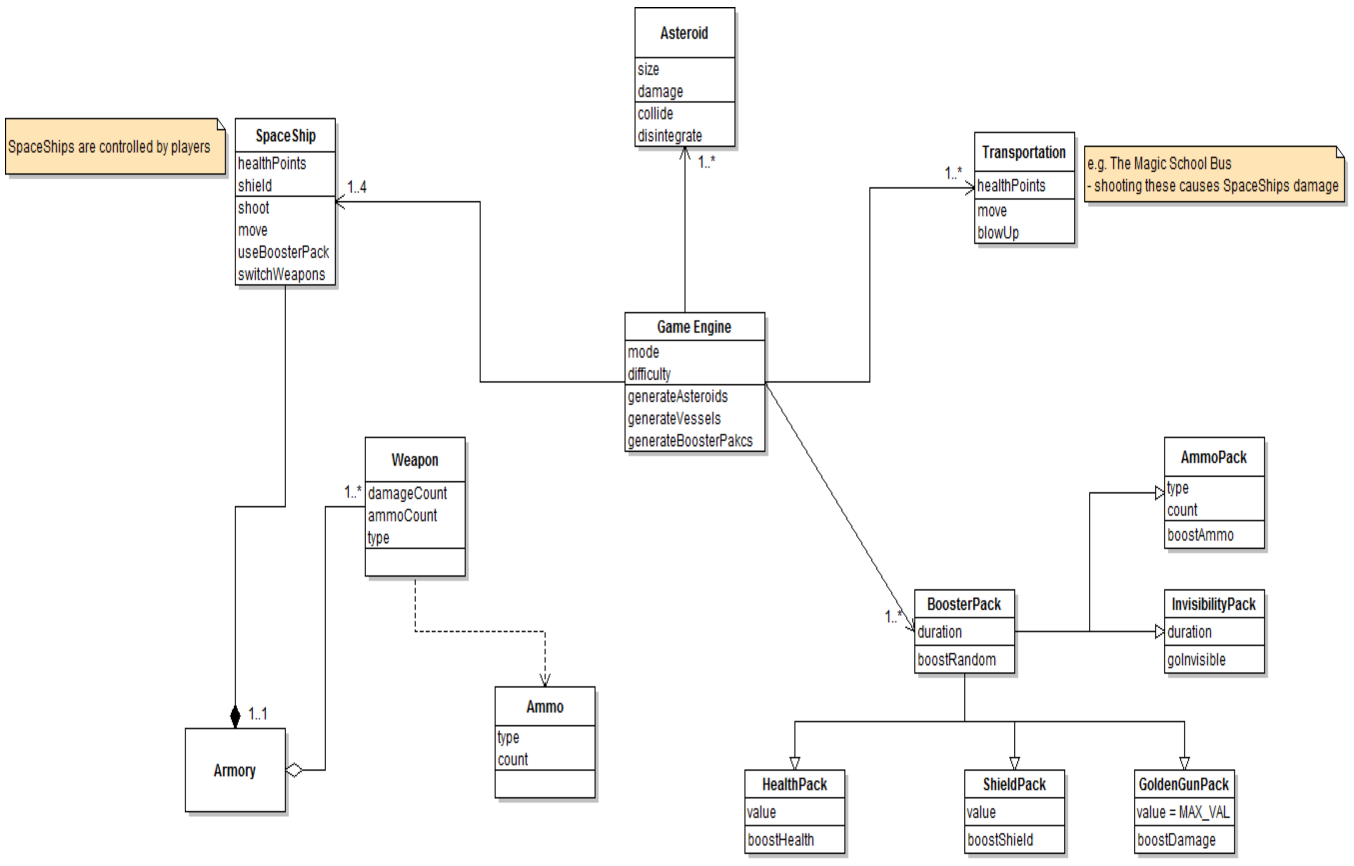


Figure 3. Lunar Lander Class Diagram

The UML Sequence Diagram is shown next. This shows the sequence of actions and communications required to start up and play the game. First, players connect to the Server and choose to join the game. The game supports 2-4 players. Only a single game can be played at a time; if a player attempts to join a game that is already in progress, the request will be denied and the player disconnected from the server.

Once the players have joined the game, a notification is sent to each client to begin the game and enable players to enter movement and combat commands. These commands are sent over the WebSocket connection to the Server and processed by the Game Engine; the Server then updates the game state based on the previous game state, the game logic, and the received player commands. The updated game state is pushed back to the clients' browsers, which then update the player's view.

This same basic sequence continues until the game ends – when only one spaceship captain is left alive!

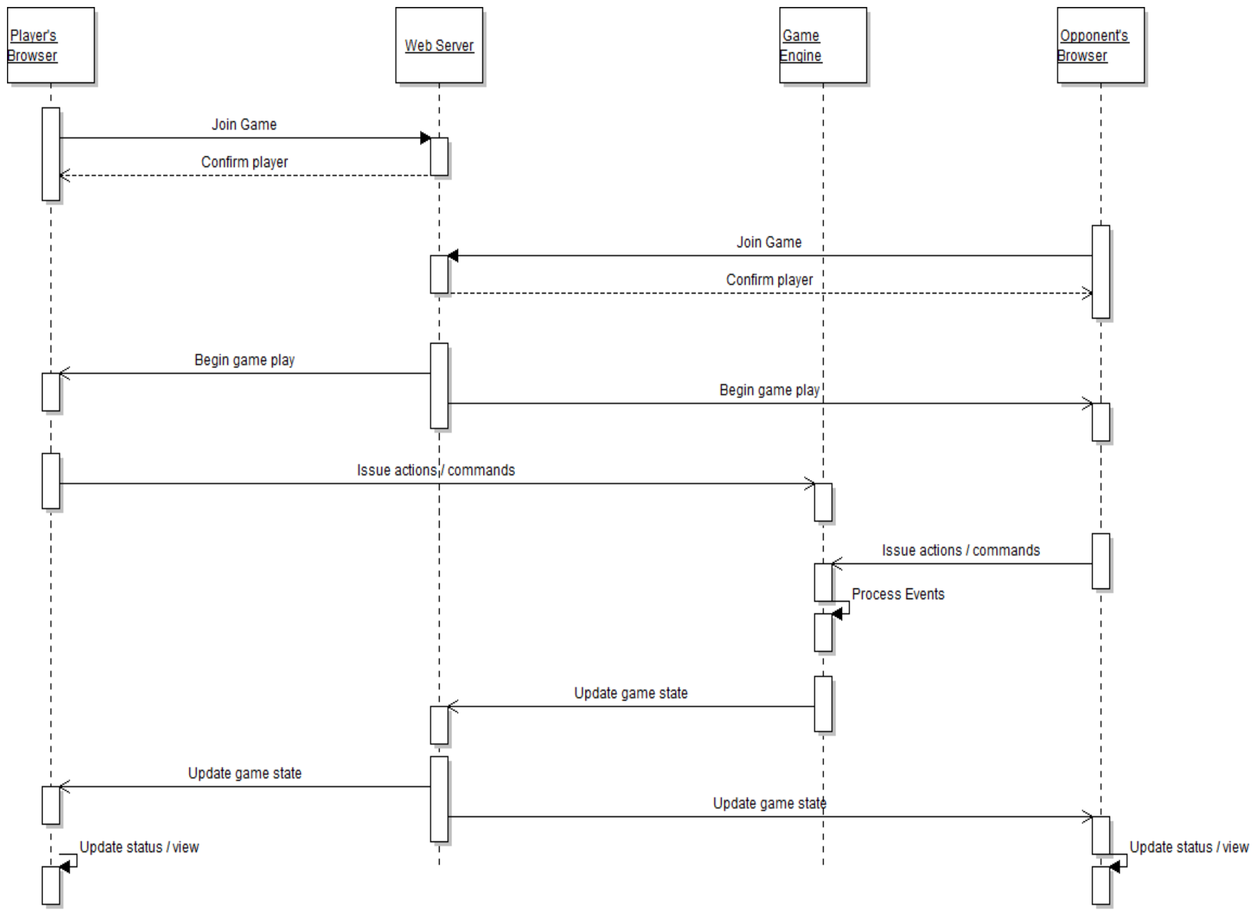


Figure 4. Lunar Lander Game Play Sequence Diagram

Finally, the UML Use Case Diagram is shown. The Use Case Diagram is used to show all of the system functionality at a glance. This diagram does not show the detailed step-by-step sequences of actions required to realize each of the Use Cases. Its purpose is to show high-level descriptions of the system's functionality, as well as the relationships between "actors" involved in each operation.

For Lunar Lander, the Use Case Diagram shows at a high level only the "Client" and "Server" actors, and the functions they are involved in.

The Client can create a WebSocket object to communicate with the Server (or display a connection error in the event of an unsuccessful connection attempt), send messages to the server, process event messages received from the server, and close the connection to the server.

The Server can receive and connect the WebSocket connection, process messages from the client(s), and return messages to the client(s).

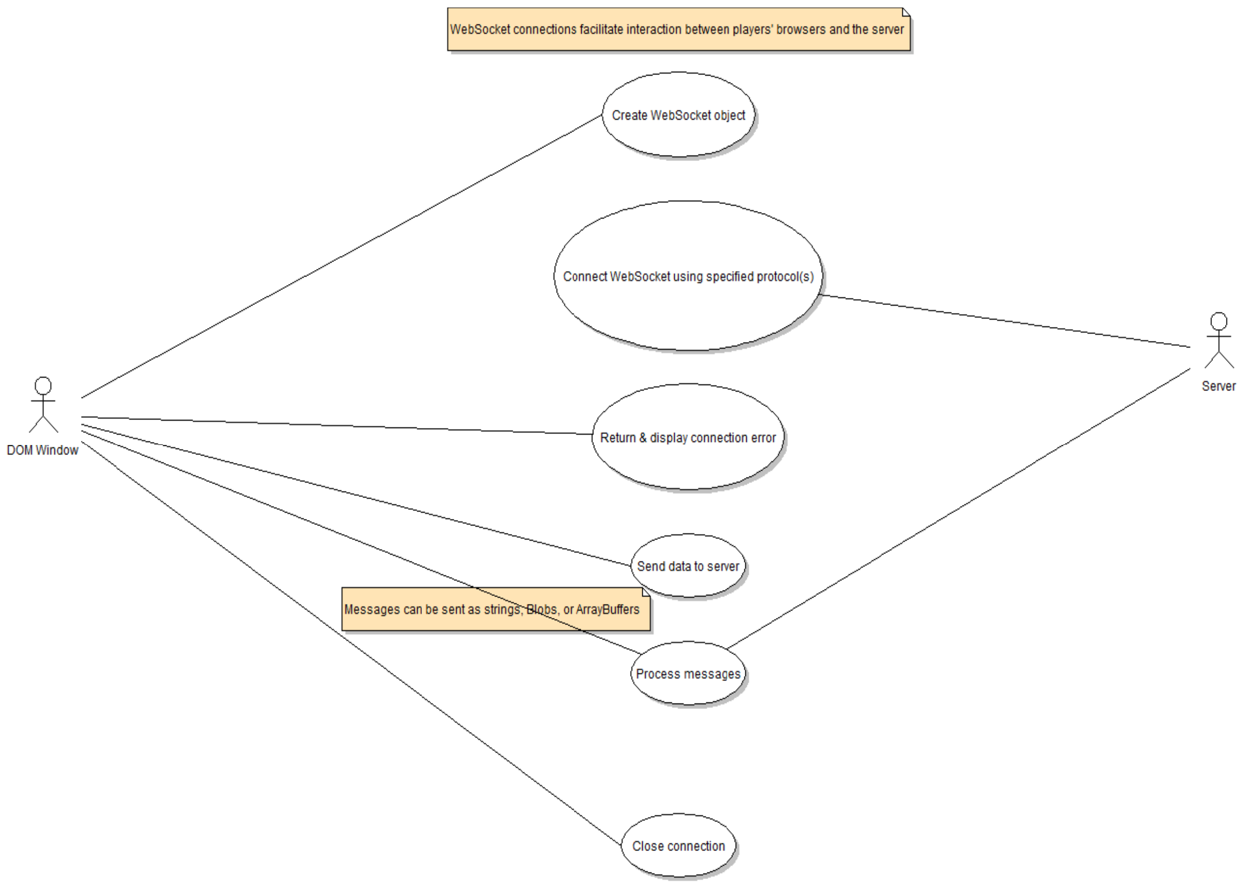

WebSocket connections facilitate interaction between players' browsers and the server
Create WebSocket object
Connect WebSocket using specified protocol(s)
Server
DOM Window
Return & display connection error
Send data to server
Messages can be sent as strings, Blobs, or ArrayBuffers
Process messages
Close connection

Figure 5. Lunar Lander UML Use Case Diagram

ACME model: The ACME model is shown below. There are three ACME views: the high-level view, which shows client and server, as well as the connection between them, a more detailed view of the client component, and a more detailed view of the server component.

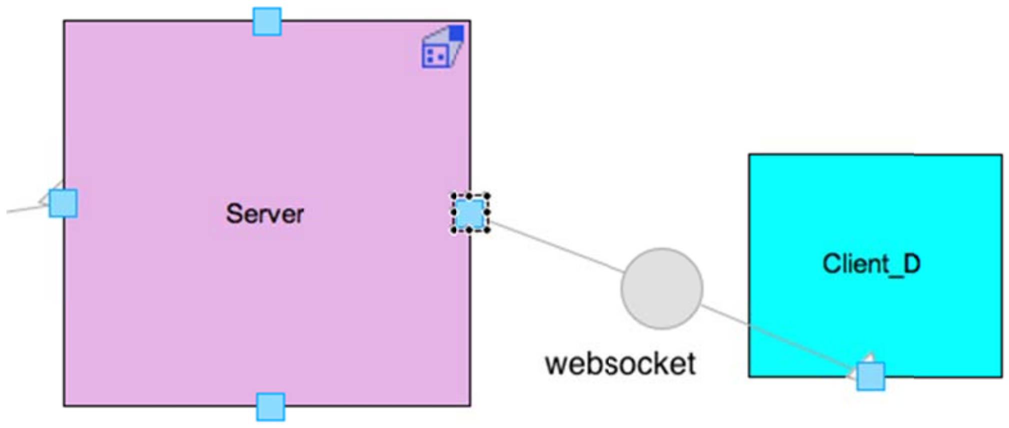


Figure 6. Client-Server connection view Modeled in ACME

The ACME client-server connection view is shown above. This view focuses on the connection between the server and a single client; note the single connector between them (the WebSocket connection) and its consistency with the other architectural views.

More detail about the nature of the client side of the application can be seen in the ACME client view below. This view is used to model and show the nature players' interaction with the game; as spaceship captains, they use the keyboard and mouse of their computer to send input commands to the browser. Layout engine software is used by the browser to allow the browser to render and show the game to the player: WebKit[4], shown in the example below, is used by the Safari and Chrome browsers, while IE uses Trident[5] and Firefox uses Gecko[6].

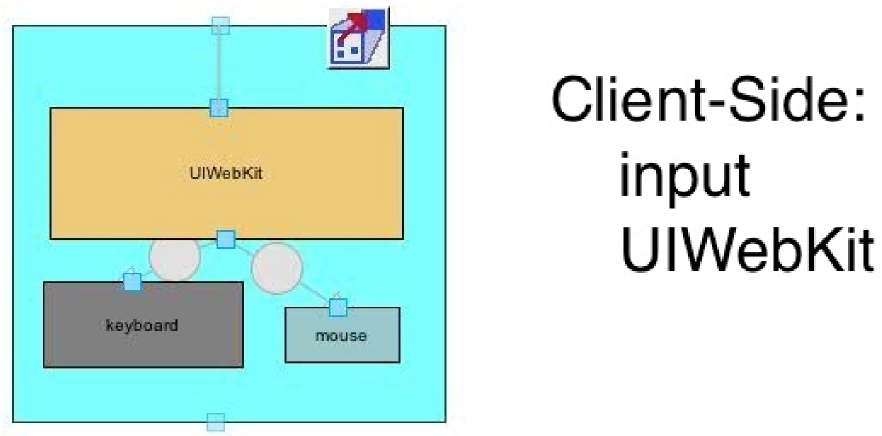
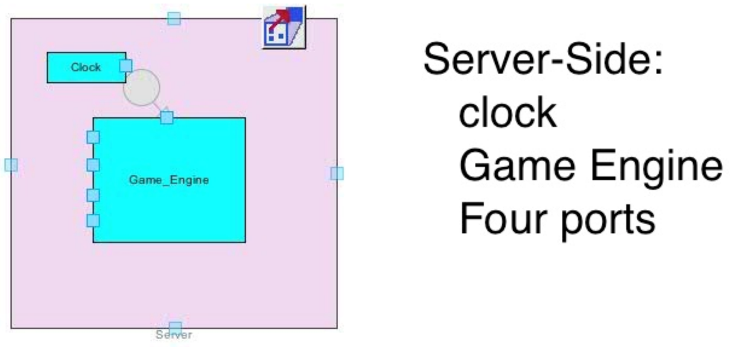


Figure 7. Client-side view Modeled in ACME

Similarly, more detail about the nature of the server side of the application can be seen in the ACME server view. This view shows the nature of the server: four ports, each of which can be used to connect a single WebSocket connection with a player, a Game Engine component, to maintain the game state and rules / logic, and a clock to synchronize the system. The internals of the Game Engine component shown in this view make up much of the UML Class Diagram shown above.



[4] http://www.webkit.org
[5] http://msdn.microsoft.com/en-us/library/aa741317.aspx
[6] https://developer.mozilla.org/en-US/docs/Gecko

Assessment of the modeling experience: Initially, this process seemed somewhat foreign to us. As software developers, often our first instinct when embarking on a new project idea is to put together *something* in code, no matter how small, to start to think about how it works. Then, generally after some time "building" in code, thinking about how things fit together, we might try to engage in activities more readily described collectively as "software architecture". Matt says, "In the past, I used to dive into coding projects and proceeding in a linear manner—coding the first view that users would see, then the next, and so forth. For this assignment, it was a new experience to design an architecture from a high level first.

Forcing ourselves to think in a top-down manner and not get too far ahead in planning specific implementation details was difficult. This is probably primarily due to a lack of comfort and familiarity in engaging in this process. Reflecting on this exercise, we feel more comfortable with what the concept of software architecture *is*, because we have experienced some small engagement with it, rather than just reading one of the many definitions available in textbooks and on the Web. It is also fair to say that we are both sure that we will approach our *next* software architecture project with increased confidence, based on the experience of engaging with this one.

Being forced to use three different architecture description languages and associated modeling tools, we can see that it's almost certainly true that no one language or tool is perfect for every project (or even *perfect* for a single project). Each has its own strengths and weaknesses. Just as is the case in selecting a programming language, it's OK to have a "favorite" or a go-to choice, maybe the one you are most familiar with, but it also seems prudent to be open to the possibility that perhaps something new might be better suited to the particular task at hand. You shouldn't code everything you do in Java just because you are "a Java guy", and you probably shouldn't *only* use UML to do your software architecture activities just because it's the only thing you know. Also, while three architecture description languages seemed like a lot to have to consider when we started the project, it's clear that there are many more that have developed and are being actively used – we switched from AADL to ACME, but we know other students used Rapide, we learned about C2 from the textbook, and we found an even longer list of ADLs posited as potential candidates for "best" ADL on Wikipedia.[7]

UML was the only one of the ADLs we used for this assignment that we had previous experience with. When someone in class asked if we had to use all 13 of the UML diagrams available for this, we were wondering *exactly* the same thing. This, I think is one of those two-edged sword aspects of UML specifically. It's great that UML 2.0 has been expanded to the point where almost any aspect of a software system you could wonder about can be described by at least one of the standard UML Diagrams. But at the same time, it becomes an issue because no one seems to agree on when any of the specific diagrams should be *required* for a particular system, or *how many* diagrams is "enough", even though people tend to have expectations about these things. This creates issues in communication focused on these external issues, and draws focus away from the software system itself. This leads us to the reflection that accepting that UML (or any other ADL) may not be able to, or might not be the best way to comprehensively describe or show everything there is to know about a software system may be

[7] http://en.wikipedia.org/wiki/Architecture_description_language

the best course. Thinking about our modeling efforts in this way made it easier to accept that **all** of our models are designed to be *complementary*, and "The Software Architecture" is really described by the totality of the models, not any one model individually.

One thing we found reassuring when starting to do this write-up about our experience was our desire to include a section talking about and explaining the *principal design decisions* we made about our system. We felt that if this concept was as important to understanding what software architecture is as the first lecture suggested, **we must be on the right track**. Of these, our intent to use the WebSockets API as our software connector between the client's browser and the game server really stood out as the **most** important and fundamental property of our system. Before we started the project, we decided to use a Google Doc to keep track of the notes we made during our joint brainstorming sessions and to share notes about our individual experiences. From this documentation, we can trace the origin of this key design decision:

> "For our multiplayer game, we quickly arrived at the client-server model since it has proven to be an effective model in the past for many successful games. Given that it was a multiplayer game, one significant aspect is updating the views when one player does something on his screen; all the other screens should be updating as quickly and efficiently as possible, so the other players may respond appropriately in the game. As a result, we discovered HTTP requests were not good enough or even designed to handle live and frequent information exchange. Our investigation led us to techniques that others have used, such as polling or long polling and server-side event-driven techniques. Still, these techniques and workarounds had severe limitations. For instance, polling at regular intervals assumes that information comes in at the same time, but that is not the case. Long polling connections may get terminated after a time-out, too.
>
> Luckily, HTML5 is breaking through and one of its native features is WebSockets. This was the perfect solution for our connectors and communication. Because of WebSockets, the size of data packets is reduced from kilobytes to 2 bytes and latency can be reduced from 150ms to 50ms."

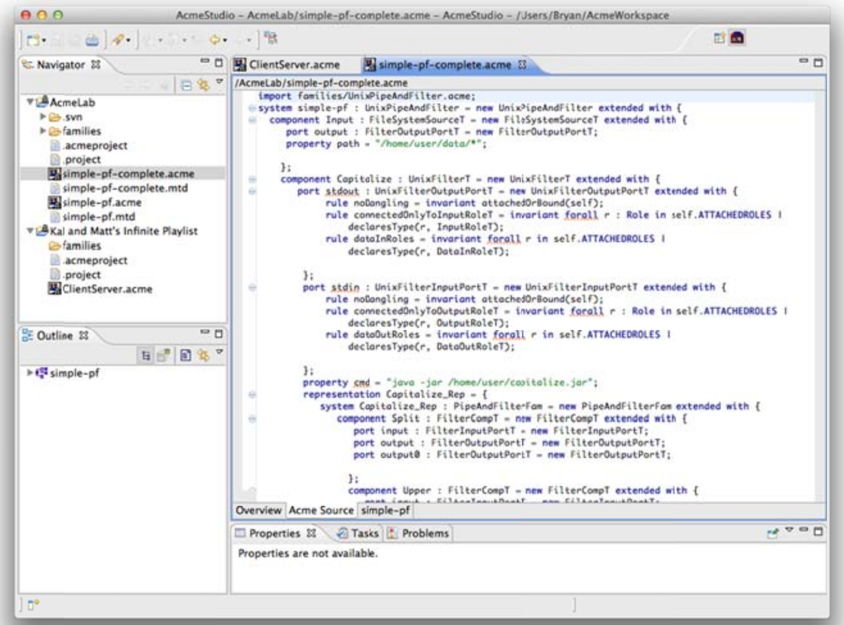
Having this source of documentation made it easier to produce this report, and we think makes it easier to trace our thinking over time. Looking back at what we wrote when we first decided to use WebSockets makes it easier to assess now whether we still think that's the right decision after engaging in the modeling exercise (we do).

Tackling the latency issue reminded Matt of a lecture he attended at Twitter, Inc. last summer. The talk was about the architecture of Twitter and how they deal with billions of tweets. Specifically, the speaker talked about Lady Gaga and her 34 million followers. When Lady Gaga tweets about her day, the architecture cannot afford to push the tweet directly to the 34 million followers. As a result, the Twitter architecture is built such that the tweet only gets pushed to her followers when they next log in via the web or mobile apps. This prevents inactive or casual Twitter users from consuming large amounts of bandwidth unnecessarily. This example highlights how we always relate architectures we build to things we already know about.

Of course, we experienced a few quibbles with the tools we used to do the modeling. For modeling UML, we used the Violet UML editor. Violet is nice because it is a very small Java program that can produce decent-looking UML diagrams quickly and easily. It is simple to get started with and learn. However, it is somewhat limited on features. For example, there is no capability to draw dashed lines around a group of classes in the Class Diagram to indicate which host they might run on.

We used ArchStudio 5 to model using xADL; this required the extra step of downloading a newer version of Eclipse (4.2 Juno, previous install was 3.7 Indigo). One thing that was unclear when first using ArchStudio was how to actually *connect* links. Initially, links added to an Archipelago diagram using just components *look* like they are connected; it was only after adding interfaces to the model that it became clear how to connect them. The green highlighting used in Archipelago was helpful for reinforcing this idea.

Finally, we initially tried to use AADL to create one of our models, but struggled with installation issues and the fragmentation of AADL support and examples. Matt is working on getting a refund for the AADL textbook he bought. Instead of using OSATE and AADL, we opted for Acme and the AcmeStudio software developed at Carnegie Mellon. The user interface was friendlier and the tutorials were clear and easy to follow.



```
import families/UnixPipeAndFilter.acme;
system simple-pf : UnixPipeAndFilter = new UnixPipeAndFilter extended with {
  component Input : FileSystemSourceT = new FileSystemSourceT extended with {
     port output : FilterOutputPortT = new FilterOutputPortT;
     property path = "/home/user/data/*";

    };
    component Capitalize : UnixFilterT = new UnixFilterT extended with {
       port stdout : UnixFilterOutputPortT = new UnixFilterOutputPortT extended with {
            rule noDangling = invariant attachedOrBound(self);
            rule connectedOnlyToInputRoleT = invariant forall r : Role in self.ATTACHEDROLES |
                declaresType(r, InputRoleT);
            rule dataInRoles = invariant forall r in self.ATTACHEDROLES |
                declaresType(r, DataInRoleT);

        };
        port stdin : UnixFilterInputPortT = new UnixFilterInputPortT extended with {
            rule noDangling = invariant attachedOrBound(self);
            rule connectedOnlyToOutputRoleT = invariant forall r : Role in self.ATTACHEDROLES |
                declaresType(r, OutputRoleT);
            rule dataOutRoles = invariant forall r in self.ATTACHEDROLES |
                declaresType(r, DataOutRoleT);

        };
        property cmd = "java -jar /home/user/capitalize.jar";
        representation Capitalize_Rep = {
           system Capitalize_Rep : PipeAndFilterFam = new PipeAndFilterFam extended with {
              component Split : FilterCompT = new FilterCompT extended with {
                 port input : FilterInputPortT = new FilterInputPortT;
                 port output : FilterOutputPortT = new FilterOutputPortT;
                 port output0 : FilterOutputPortT = new FilterOutputPortT;

                };
                component Upper : FilterCompT = new FilterCompT extended with {
```

Figure 9. AcmeStudio User Interface on Mac OS X platform

[redacted]

[redacted]

INF 221, Software Architecture
Homework Parts 3, 4 & 5
3/6/2013

Super Asteroid Melee Implementation

Program update: Our version of a "Lunar Lander" game that is **significant**, **multi-player**, and **networked** was originally conceived in the vein of a 2-dimensonal, top-down viewpoint multiplayer spaceship shooter game (working title: "*Super Asteroid Melee*").

In the process of implementing *Super Asteroid Melee* we made an important (*architectural, even)* decision that slightly changed the nature of the game itself: to save ourselves the trouble of implementing a large amount of code to deal with the orientation of the spaceships, and the trajectory of the projectiles they fired, we redesigned the core game concept, but kept the same theme.

Instead of having players pilot spaceships and attempt to kill each other, we changed *Super Asteroid Melee* so that players join the game as *teammates*, forming a squadron of spaceship captains to coordinate their efforts and destroy waves of asteroids that fall towards them.

What this did for us was simplify the physics we would have to simulate.
All player spaceships are now oriented in a north-south orientation, facing the top of their browser window. When they fire projectiles, their weapons are released and travel in a straight line pattern.
The asteroids fall from the top of the screen toward the squadron of players.
Shooting an asteroid with a projectile will destroy both the asteroid and the projectile.
If a player collides with an asteroid, his ship explodes and he dies.

An extra level of difficulty and fun is added by having the players cooperate as teammates. We added a game mechanic whereby players can be destroyed not just by collisions with asteroids, but by colliding with player-fired projectiles as well. This creates the concept of *friendly-fire* deaths.
We liked this idea because it can act as a deterrent to the player from always using the strategy of firing constantly – after we started testing the game we felt that limiting players' firing rate or available ammunition just wasn't as much fun, so the friendly-fire concept creates another level of difficulty and addresses this concern. Of course, once players encounter the friendly-fire idea, we fully expect that some of them will adopt a strategy of *intentionally* firing on their "teammates", trying to kill them "for the lulz". This is great because it opens up the game to support multiple play styles.

This original conception of our game was designed for a minimum of **2** players, and a maximum of **4** players. This seemed like a good number to limit an overhead-battle scenario to, and we felt like it would limit the level of difficulty in implementation for us.

One of the things that using HTML5, WebSockets, socket.io and Node.js bought us though is simplicity in coding for additional players joining the game, using our top-to-bottom screen orientation. Because of this, the final version of *Super Asteroid Melee* has **no** software-limited cap on the number of players: theoretically, and number of players can continue to join the game and play simultaneously.

We tested with as many as six players by running the client code in separate browser tabs during development.

A single-player version of *Super Asteroid Melee* is actually possible, as implemented. There is no queuing or "lobby" mechanism that requires two players to join the game before it can begin. Players join the game by pointing their web browser to the page hosting the game, and are connected immediately. The other players already in the game are notified of the presence of the new player by means of event propagation through **socket.io** – a JavaScript software library designed to be used as a module added to a server running **Node.js**, and implementing the HTML5 WebSocket API. A similar chain of events handles player disconnects.

The level of difficulty for the squadron of spaceship captains is scaled automatically: as more players join the game, the number and frequency of asteroids that rain down on them increases. This is accomplished by having each connected client run a loop asking the server to create an asteroid on a timer. So while a single player can play the game alone, it is actually **much** more interesting to play with a few friends!

Some of the concepts from our original design were pared down, due to time constraints. Instead of keeping track of how much "damage" has to be done to asteroids or players to destroy them, all collisions result in instant death.
The random powerup nodes were dropped as well, in the interest of time.
They could be added without major difficulty given additional time. The logic for spawning, moving, and collecting them would be very similar to the logic for the asteroids, but would produce different effects.

The idea of the Transportation vehicles (the *Magic School Bus*, for example) as a way to keep players from constantly "mashing" their firing button(s) was replaced with the friendly-fire concept – the possibility of shooting and killing teammates acts as a similar deterrent (or motivation, depending on the player's mindset) instead.

Implementation languages and tools used: The main web page that hosts the game is written in **HTML 5**. The server component runs **Node.js**, a server-side software system designed for writing scalable Internet applications, notably web servers. Programs are written on the server side in **JavaScript**, using event-driven, asynchronous I/O to minimize overhead and maximize scalability.[1] The **socket.io** library implements the **WebSocket** API and provides additional features like heartbeats, timeouts, and disconnection support.[2] Clients connect to the server by visiting the web page hosting the game using a web browser; that HTML page loads and invokes JavaScript files that contain the client's portion of the game logic.

High-level design decisions: We recap here our most important design decisions from Parts 1 + 2 for reference, and because these are the components that we chose to focus on when trying to provide evidence that the implementation and the model(s) are consistent.

1. The architectural *style* of client-server.

[1] http://en.wikipedia.org/wiki/Node.js
[2] http://socket.io/#faq

2. The use of the **WebSocket** API for communication (aka as a *connector*) between client(s) and server. WebSocket is an API specified by the W3C that enables Web pages to use the WebSocket protocol (defined by the IETF) for two-way communication with a remote host.[3]
3. An *event-based* architectural style; JavaScript event handlers are used to determine the appropriate response to messages sent between the client and server, which may be received at any time while the connection is established.

xADL model: The updated architecture model in xADL is shown below. There are two main views: the first is a high-level view of the whole Web Application as a top-level component. Inside this component is a subcomponent showing the components of Server, and browser-based clients, and the connectors between them, the WebSocket connections. This view is very similar to the version of Lunar Lander shown in Figure 4-12 of the textbook; both are multiplayer applications in the client-server style.

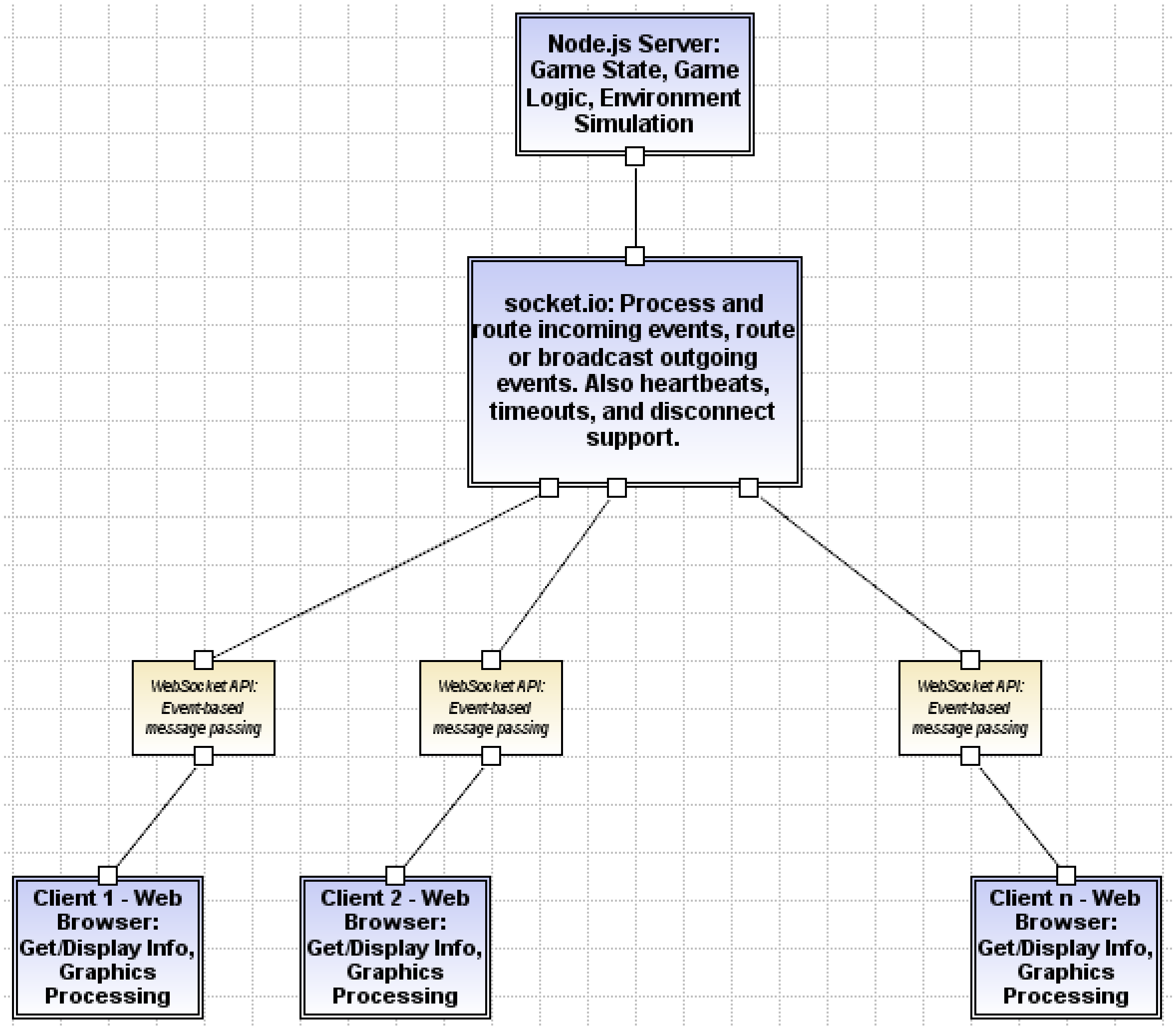

Node.js Server: Game State, Game Logic, Environment Simulation
socket.io: Process and route incoming events, route or broadcast outgoing events. Also heartbeats, timeouts, and disconnect support.
WebSocket API: Event-based message passing
WebSocket API: Event-based message passing
WebSocket API: Event-based message passing
Client 1 - Web Browser: Get/Display Info, Graphics Processing
Client 2 - Web Browser: Get/Display Info, Graphics Processing
Client n - Web Browser: Get/Display Info, Graphics Processing

Figure 1. Updated – Super Asteroid Melee Web Application Modeled in xADL

[3] http://dev.w3.org/html5/websockets/

This diagram shows the division of responsibilities for the game: the Server maintains the game state, implements the game logic, and simulates the battlefield environment (including the creation and movement of Asteroids). The Clients get and display information from the Server, and are responsible for performing the graphics processing and presentation. Clients also send messages to the Server regarding their movement and weapon firing commands using the same WebSocket connection.

The diagram has been updated by adding a new component, the **socket.io** library. Socket.io is installed as a module for the Node.js server, and acts as an intermediary between the clients and the server. Its main function is to process and route the event messages exchanged between the clients and the server. Socket.io provides a simple interface to enable the server component to route an event message to a single connected client, or to "broadcast" a message to all other connected clients. Socket.io also provides some additional functionality, in addition to the WebSocket API: it also handles heartbeats, timeouts, and automatic disconnection for the server.

Finally, this diagram has been updated to reflect the variable and unbounded number of clients supported. Instead of Client 1 through Client 4, Client 3 has been removed, and the final Client component labeled as "client n" to indicate this.

The second xADL view models the WebSocket connection between Client and Server (instead of just 2-4 such connections when the game is being played, there are now any number of these connections, one for each player who has joined the game).

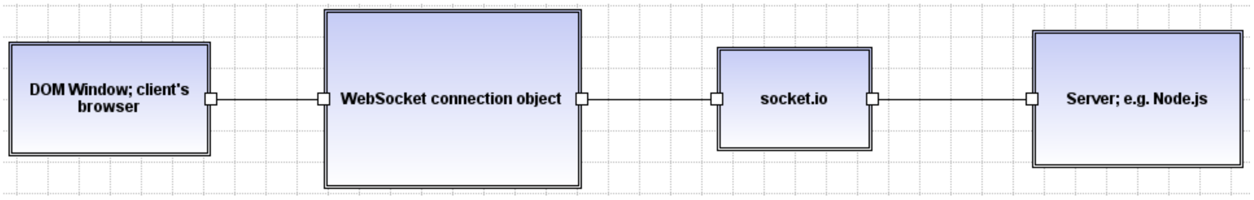


Figure 2. WebSocket connection modeled in xADL

This diagram shows the nature of the connection between client and server. The client is a DOM Window object, available to JavaScript code on a WebSocket-compliant browser (i.e. recent versions of Chrome, Firefox, IE 10 or Safari 6).[4]

The client initiates a connection by creating a WebSocket object, which will automatically attempt to open the connection to the server.[5] If the connection is successful, the client will be able to send asynchronous messages to the server. These messages can be sent as `strings, Blobs, or ArrayBuffers`. Complex messages can be created by encoding them using JSON.

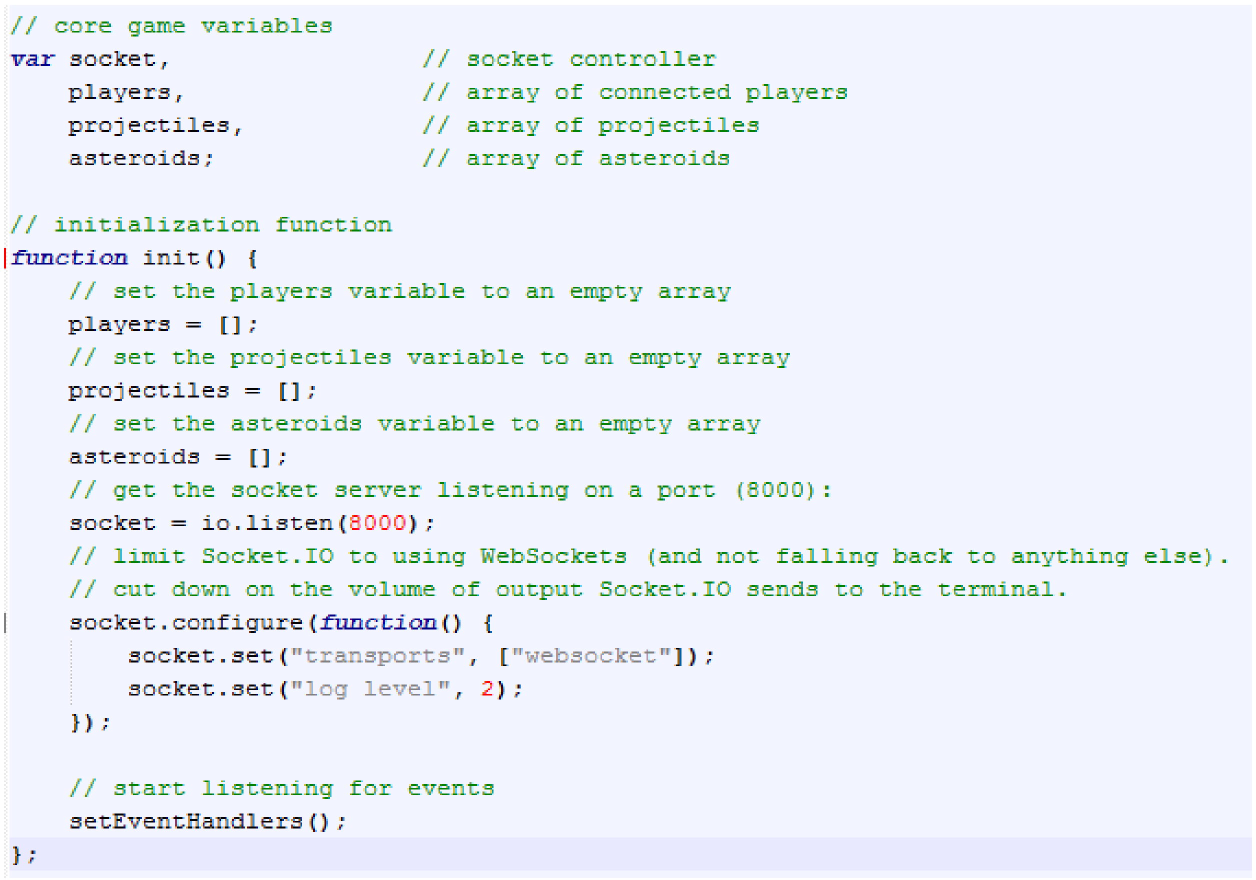
The server's replies are handled via the event-based WebSocket API by delivering a "message" event to the client's `onmessage` function. Two-way communication can thus continue until the client closes the WebSocket connection by calling `close()`.

This diagram has been updated to reflect the addition of socket.io, which acts as a wrapper around the WebSocket API and an intermediary in this communication.

[4] https://developer.mozilla.org/en-US/docs/WebSockets
[5] https://developer.mozilla.org/en-US/docs/WebSockets/Writing_WebSocket_client_applications

Consistency in the xADL Models: The important aspects of the system captured by the xADL models are
1. The architectural *style* of client-server, and
2. The use of the **WebSocket** API for communication between client(s) and server.

The bulk of the server-side game logic is contained in the **game.js** file.



```
// core game variables
var socket,                 // socket controller
    players,                // array of connected players
    projectiles,            // array of projectiles
    asteroids;              // array of asteroids

// initialization function
function init() {
    // set the players variable to an empty array
    players = [];
    // set the projectiles variable to an empty array
    projectiles = [];
    // set the asteroids variable to an empty array
    asteroids = [];
    // get the socket server listening on a port (8000):
    socket = io.listen(8000);
    // limit Socket.IO to using WebSockets (and not falling back to anything else).
    // cut down on the volume of output Socket.IO sends to the terminal.
    socket.configure(function() {
        socket.set("transports", ["websocket"]);
        socket.set("log level", 2);
    });

    // start listening for events
    setEventHandlers();
};
```

Figure 3. game.js server initialization code

In the initialization function of **game.js**, we see how the server sets up to listen for incoming client requests on a specific port (port 8000) using a call to **socket.io** (io.listen). The server is then able to initiate all future communication with its clients through the **socket** variable, which acts as a socket controller.

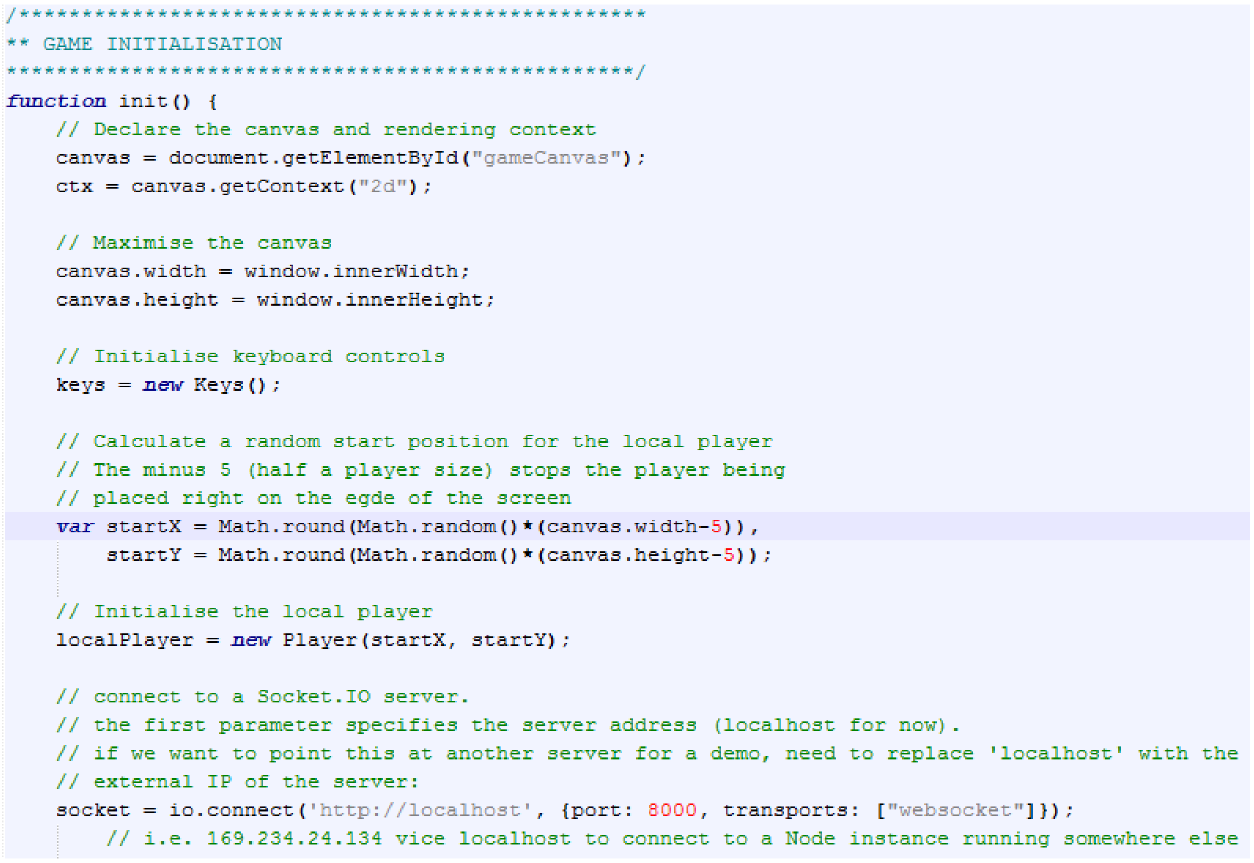
The configuration of the socket controller also restricts communication to ONLY using the WebSockets transport (protocol). If a client attempts to connect and interact with the server through the socket controller and does not support WebSockets (i.e. a mobile browser or older version of Internet Explorer), the communication requests will fail, and the web page will not load.

Although Socket.IO supports multiple transports, and will automatically fall back to Adobe Flash sockets, JSONP polling, or AJAX long polling if the client supports those transports[6], we have chosen to **explicitly** configure our socket controller to **only** support WebSockets. This supports our architectural modeling and our architectural ideal of ensuring low latency for our game, which requires a near-real-time

[6] http://en.wikipedia.org/wiki/Socket.io

component that can be provided by the lower overhead associated with WebSockets, when compared to other transport protocols which operate on top of HTTP and require full HTTP headers.

The bulk of the client-side operations are also contained in a matching **game.js** file.



```
/****************************************************
** GAME INITIALISATION
****************************************************/
function init() {
    // Declare the canvas and rendering context
    canvas = document.getElementById("gameCanvas");
    ctx = canvas.getContext("2d");

    // Maximise the canvas
    canvas.width = window.innerWidth;
    canvas.height = window.innerHeight;

    // Initialise keyboard controls
    keys = new Keys();

    // Calculate a random start position for the local player
    // The minus 5 (half a player size) stops the player being
    // placed right on the egde of the screen
    var startX = Math.round(Math.random()*(canvas.width-5)),
        startY = Math.round(Math.random()*(canvas.height-5));

    // Initialise the local player
    localPlayer = new Player(startX, startY);

    // connect to a Socket.IO server.
    // the first parameter specifies the server address (localhost for now).
    // if we want to point this at another server for a demo, need to replace 'localhost' with the
    // external IP of the server:
    socket = io.connect('http://localhost', {port: 8000, transports: ["websocket"]});
        // i.e. 169.234.24.134 vice localhost to connect to a Node instance running somewhere else
```

Figure 4. game.js client initialization code

In the initialization function of **game.js**, we see the other side of the WebSocket connection being set up. The client sends a connection request to the server on the specified port (port 8000) using a call to **socket.io** (io.connect). The client is then able to initiate all future communication with the server through the **socket** variable, which acts as a socket controller.
The optional `transports` parameter specifies only the WebSocket transport protocol should be used for this connection. Thus we have ensured that our client and server can communicate, and that they can communicate exclusively using WebSockets.

UML Model: The architecture model in UML is shown below. There are three UML views: the Class Diagram view, which shows the elements of the game software in an Object-Oriented paradigm, a series of Sequence Diagrams, which show the sequence of operations and interactions between components involved in starting and playing the game, and a Use Case Diagram, which shows at a glance the high-level functionality provided by the software.

The updated UML Class Diagram is shown below. The significant changes include removing the BoosterPack class and its specializations, since we decided not to include Booster Packs in the game due to time constraints. The Transportation class was also removed. The class we called GameEngine in the original UML model is now called the Game class, but largely serves the same purpose – maintaining game state and simulating the environment.

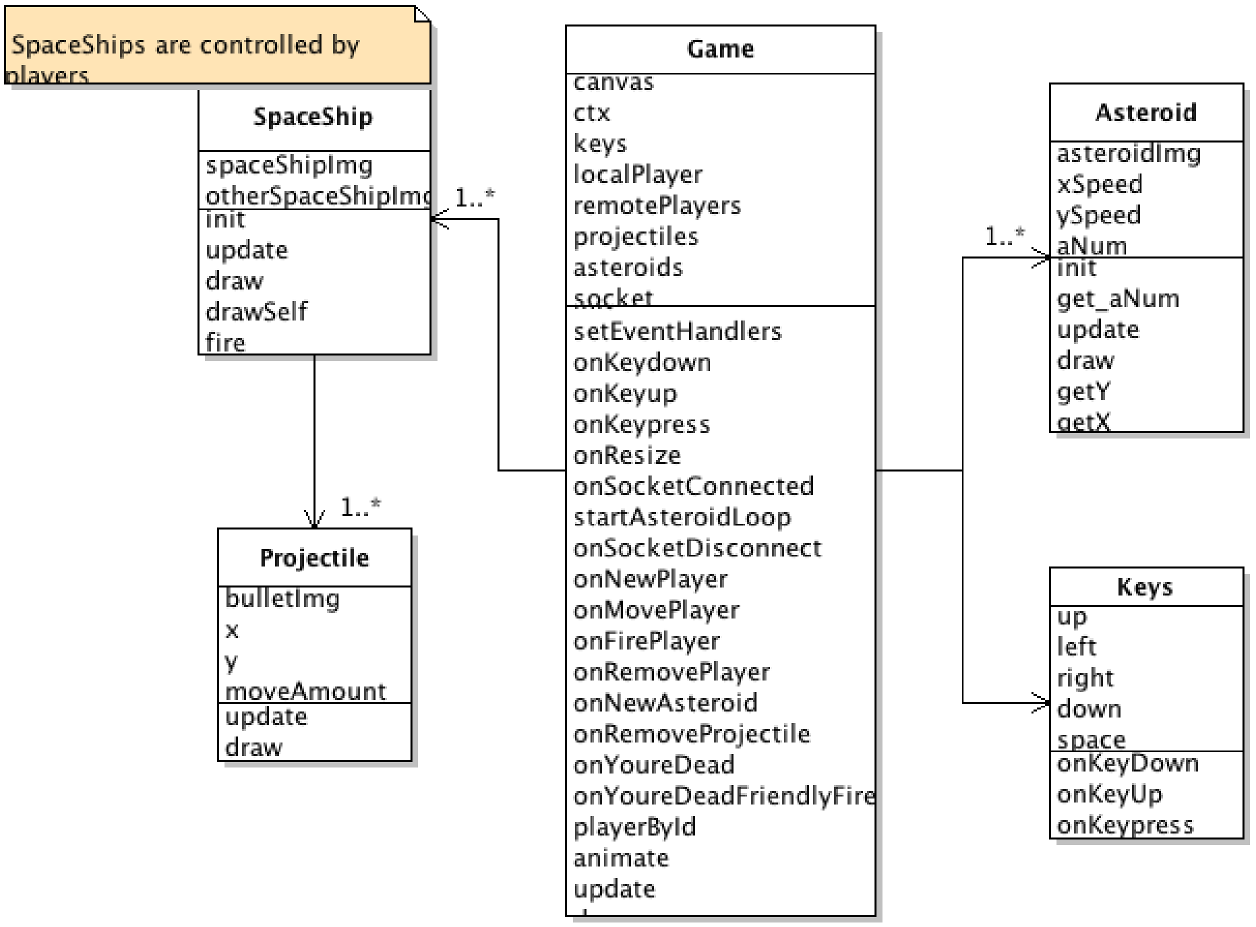


Figure 5. Super Asteroid Melee Class Diagram

The UML Sequence Diagrams are shown next. The original sequence diagram showed the sequence of actions and communications required to start up and play the game, but the implemented functionality is quite different than originally modeled. Here is the updated sequence diagram.

Note: We have left out the analogous Client WebSocket controller Object Lifeline for the Opponent's Browser (shown on the right) from these diagrams for the sake of simplicity and readability. In reality, there **are** such WebSocket controller objects for each client in the system, but they perform the same message-routing duties as the controller connected to the Player's Browser (shown on the left).

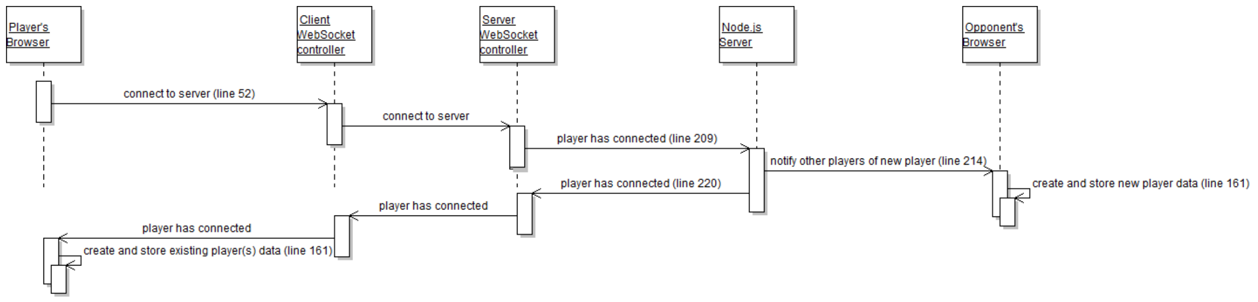

Player's Browser
Client WebSocket controller
Server WebSocket controller
Node.js Server
Opponent's Browser
connect to server (line 52)
connect to server
player has connected (line 209)
notify other players of new player (line 214)
create and store new player data (line 161)
player has connected (line 220)
player has connected
player has connected
create and store existing player(s) data (line 161)

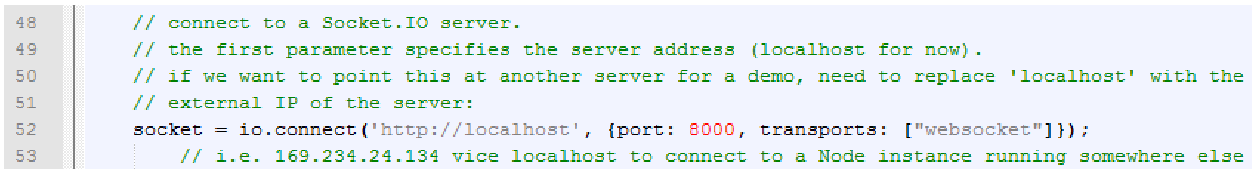
Figure 6. Game Join Play Sequence Diagram

Consistency in the model and implementation:

To trace the consistency between this UML model and the implementation, we can trace the sequence of commands and the associated source code that execute them:

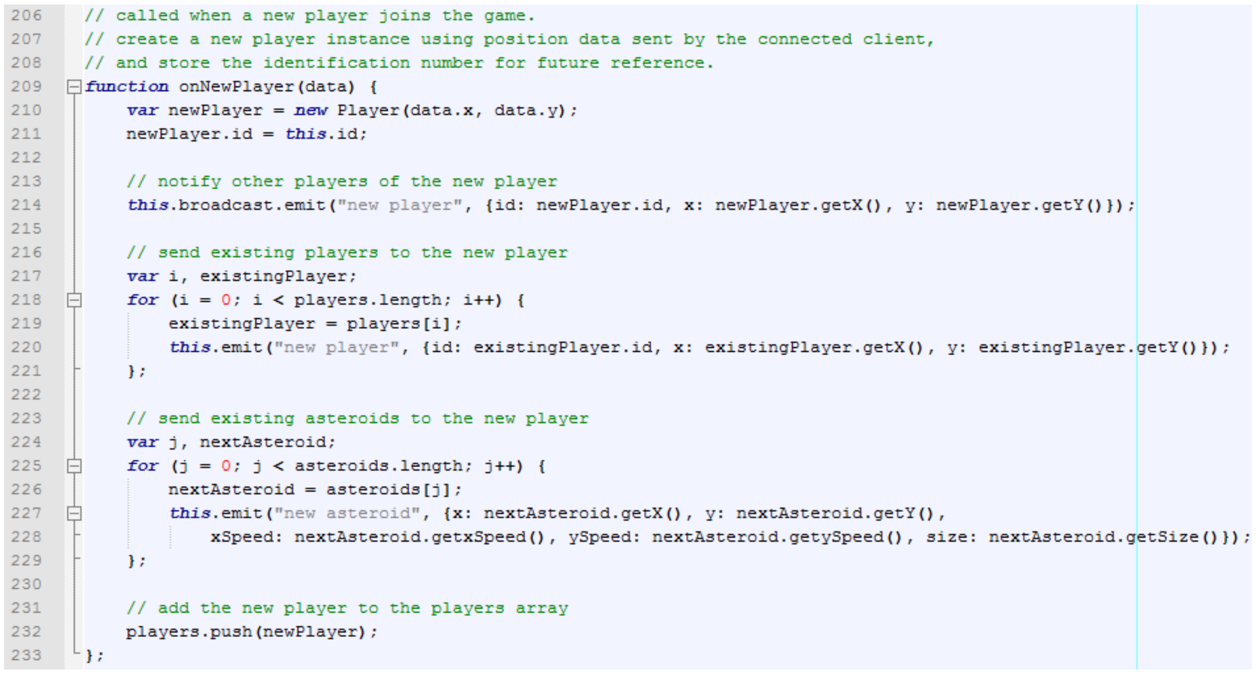
1. The player's browser sends a request to connect to the server when loading the `index.html` web page.

This request is executed on line 52 of the client's `game.js` file:



```
48      // connect to a Socket.IO server.
49      // the first parameter specifies the server address (localhost for now).
50      // if we want to point this at another server for a demo, need to replace 'localhost' with the
51      // external IP of the server:
52      socket = io.connect('http://localhost', {port: 8000, transports: ["websocket"]});
53          // i.e. 169.234.24.134 vice localhost to connect to a Node instance running somewhere else
```

2. The client's WebSocket controller routes the message to the server's WebSocket controller. This triggers the server's `onNewPlayer` callback (line 209 of the server's `game.js` file):

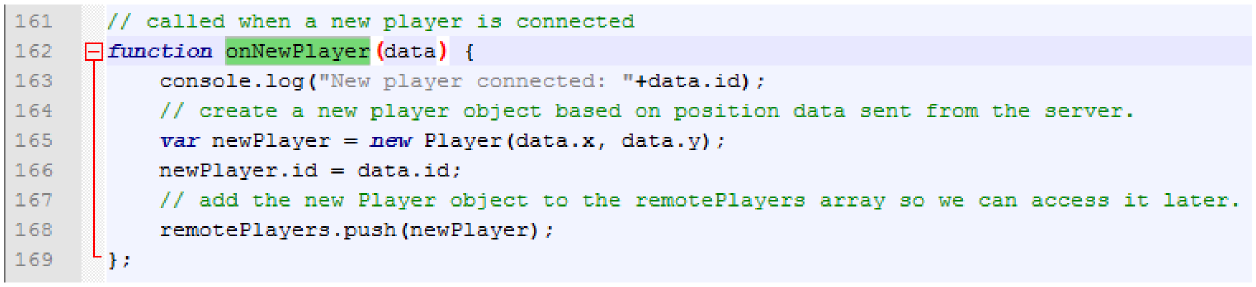


```
206 // called when a new player joins the game.
207 // create a new player instance using position data sent by the connected client,
208 // and store the identification number for future reference.
209 function onNewPlayer(data) {
210     var newPlayer = new Player(data.x, data.y);
211     newPlayer.id = this.id;
212 
213     // notify other players of the new player
214     this.broadcast.emit("new player", {id: newPlayer.id, x: newPlayer.getX(), y: newPlayer.getY()});
215 
216     // send existing players to the new player
217     var i, existingPlayer;
218     for (i = 0; i < players.length; i++) {
219         existingPlayer = players[i];
220         this.emit("new player", {id: existingPlayer.id, x: existingPlayer.getX(), y: existingPlayer.getY()});
221     };
222 
223     // send existing asteroids to the new player
224     var j, nextAsteroid;
225     for (j = 0; j < asteroids.length; j++) {
226         nextAsteroid = asteroids[j];
227         this.emit("new asteroid", {x: nextAsteroid.getX(), y: nextAsteroid.getY(),
228             xSpeed: nextAsteroid.getxSpeed(), ySpeed: nextAsteroid.getySpeed(), size: nextAsteroid.getSize()});
229     };
230 
231     // add the new player to the players array
232     players.push(newPlayer);
233 };
```

3. Within this callback, the server sends a message to the other connected clients notifying them of the new player (line 214, using Socket.IO's broadcast.emit function).

The server also sends a message to the newly connected player for **each** already-connected client, to notify the new client of the existence of the other clients (line 220, using Socket.IO's emit function).

4. Finally, both the new client and the previously-connected clients execute callbacks to process the event in their own `onNewPlayer` functions:



```
// called when a new player is connected
function onNewPlayer(data) {
    console.log("New player connected: "+data.id);
    // create a new player object based on position data sent from the server.
    var newPlayer = new Player(data.x, data.y);
    newPlayer.id = data.id;
    // add the new Player object to the remotePlayers array so we can access it later.
    remotePlayers.push(newPlayer);
};
```

The Player Movement Sequence Diagram is shown next. This diagram shows the sequence of events involved in player movement, from the player pressing on the keyboard to initiate a command, to the other connected players being notified:

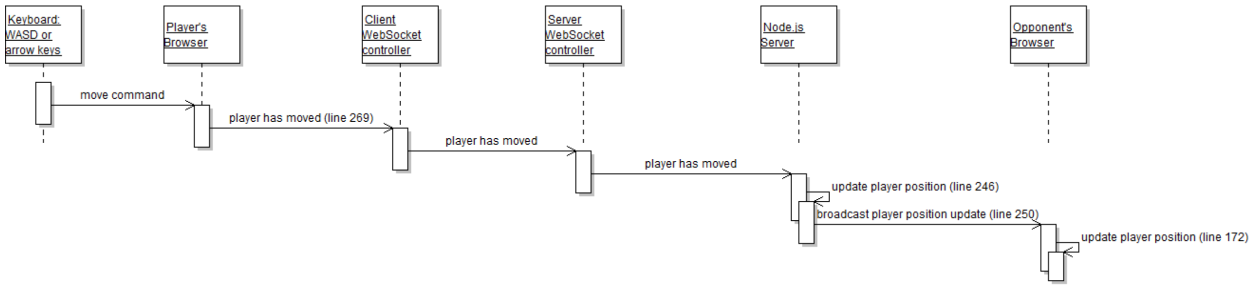
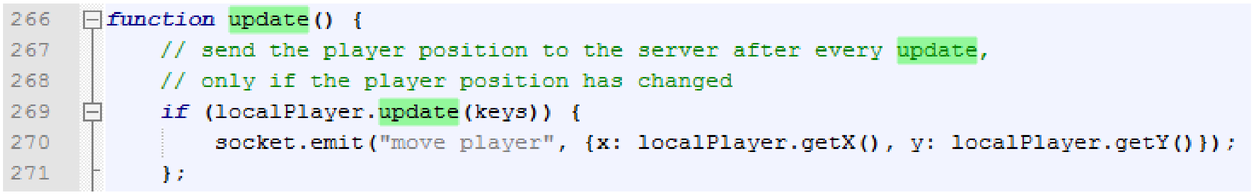

Keyboard: WASD or arrow keys
Player's Browser
Client WebSocket controller
Server WebSocket controller
Node.js Server
Opponent's Browser
move command
player has moved (line 269)
player has moved
player has moved
update player position (line 246)
broadcast player position update (line 250)
update player position (line 172)

Figure 7. Player Movement Sequence Diagram

To trace the consistency between this UML model and the implementation, we can trace the sequence of commands and the associated source code that execute them:

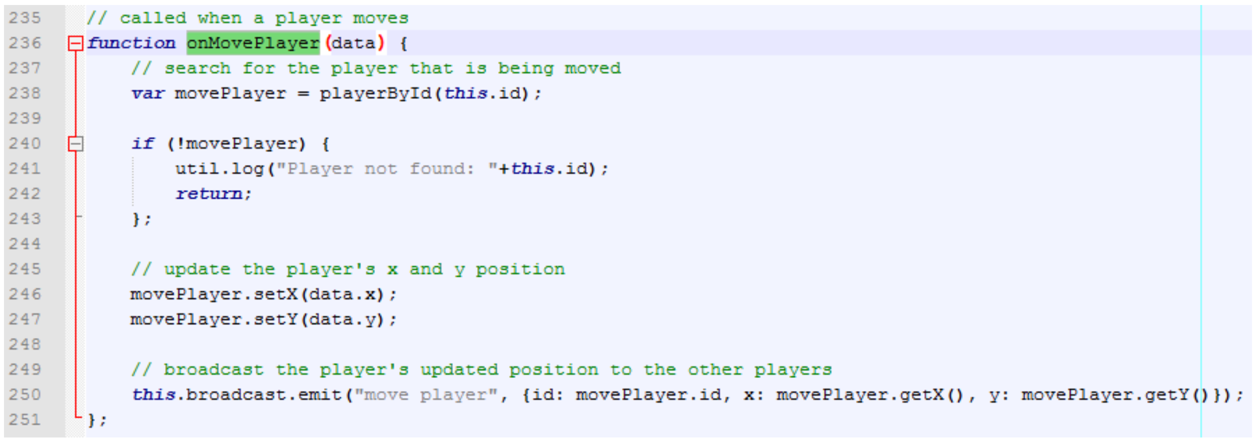
1. The player initiates a command to move his spaceship. This is done by using either the W-A-S-D keys or the arrow keys (W corresponds to up, A to left, S to right, and D to down). These keyboard presses are detected by the browser and passed on to a keyboard listener in the client that runs in a loop.

2. The client initiates an event message to the server, indicating that he has moved (line 269). The keys variable is the return of the function in the Keys class that handles keyboard input:



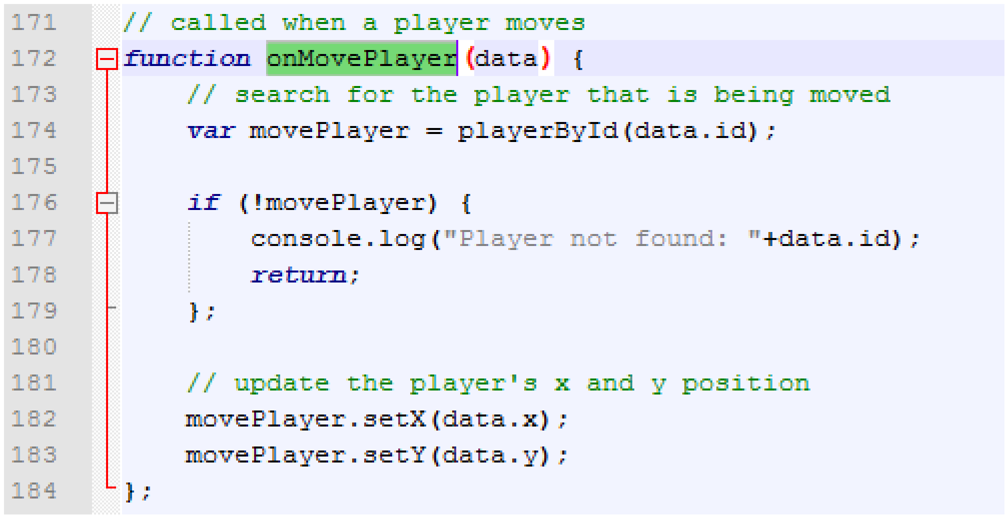
```
266 function update() {
267     // send the player position to the server after every update,
268     // only if the player position has changed
269     if (localPlayer.update(keys)) {
270         socket.emit("move player", {x: localPlayer.getX(), y: localPlayer.getY()});
271     };
```

3. After the message is routed to the server, the server's `onMovePlayer` callback is invoked. The server updates the player's location (line 246), and broadcasts the player's new coordinates to the other connected clients (line 250):



```
235  // called when a player moves
236  function onMovePlayer(data) {
237      // search for the player that is being moved
238      var movePlayer = playerById(this.id);
239
240      if (!movePlayer) {
241          util.log("Player not found: "+this.id);
242          return;
243      };
244
245      // update the player's x and y position
246      movePlayer.setX(data.x);
247      movePlayer.setY(data.y);
248
249      // broadcast the player's updated position to the other players
250      this.broadcast.emit("move player", {id: movePlayer.id, x: movePlayer.getX(), y: movePlayer.getY()});
251  };
```

4. Finally, the other connected clients execute callbacks to process the move event in their own `onMovePlayer` functions:



```
// called when a player moves
function onMovePlayer(data) {
    // search for the player that is being moved
    var movePlayer = playerById(data.id);

    if (!movePlayer) {
        console.log("Player not found: "+data.id);
        return;
    };

    // update the player's x and y position
    movePlayer.setX(data.x);
    movePlayer.setY(data.y);
};
```

The Player Firing Sequence Diagram is shown last. This diagram shows the sequence of events involved in players firing their cannons, from the player pressing on the keyboard to initiate a command, to the other connected players being notified. This sequence is very similar to the Player Movement sequence. The main difference is in the processing required by the server.

In the firing case, the server must create new Projectile objects when the player fires, and must create event messages to notify **both** the firing player and the other connected clients of them. Here is the Firing Sequence Diagram:

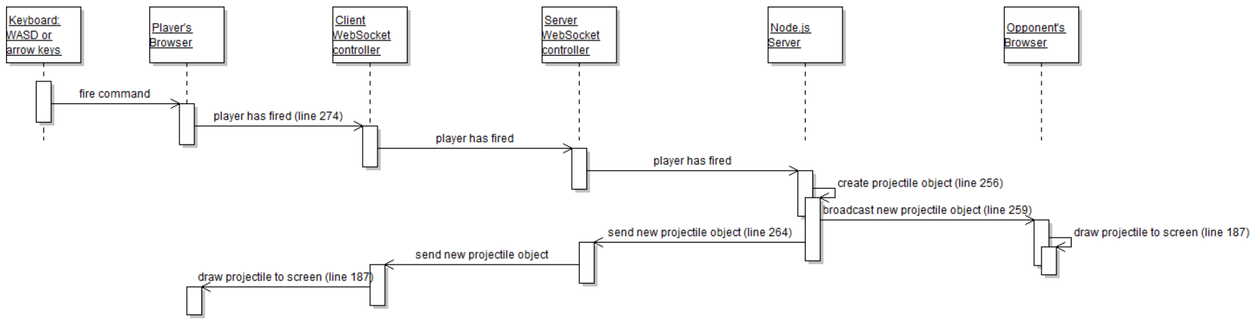
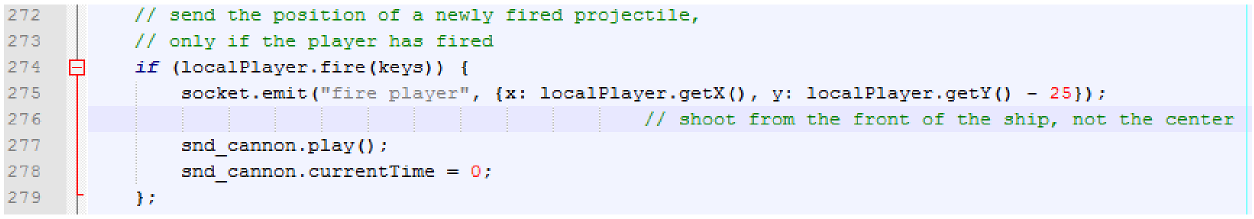

Keyboard: WASD or arrow keys
Player's Browser
Client WebSocket controller
Server WebSocket controller
Node.js Server
Opponent's Browser
fire command
player has fired (line 274)
player has fired
player has fired
create projectile object (line 256)
broadcast new projectile object (line 259)
draw projectile to screen (line 187)
send new projectile object (line 264)
send new projectile object
draw projectile to screen (line 187)

Figure 8. Player Firing Sequence Diagram

To trace the consistency between this UML model and the implementation, we can trace the sequence of commands and the associated source code that execute them:

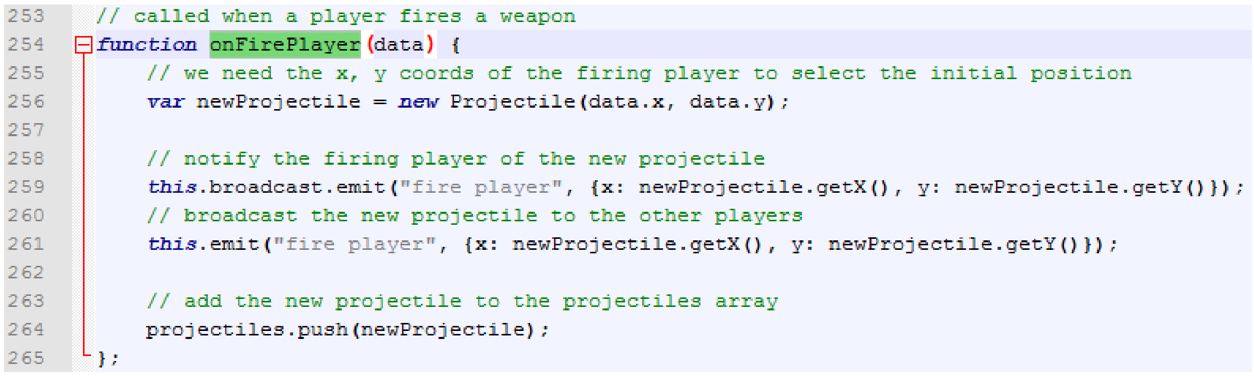
1. The player initiates a command to fire his spaceship's cannon. This is done by using the space bar. These keyboard presses are detected by the browser and passed on to a keyboard listener in the client that runs in a loop.

2. The client initiates an event message to the server, indicating that he has fired (line 274). The keys variable is the return of the function in the Keys class that handles keyboard input:



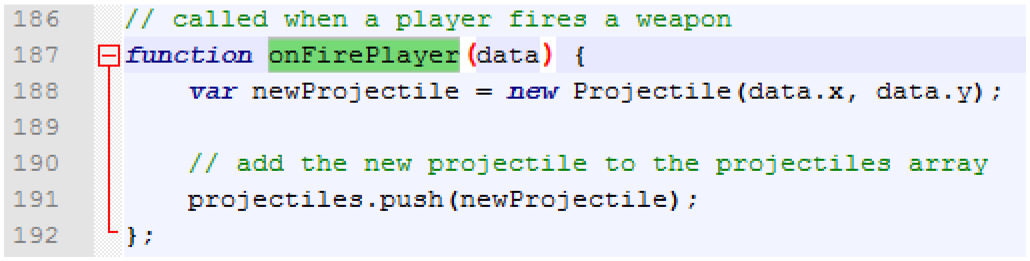
```
272    // send the position of a newly fired projectile,
273    // only if the player has fired
274    if (localPlayer.fire(keys)) {
275        socket.emit("fire player", {x: localPlayer.getX(), y: localPlayer.getY() - 25});
276                                                // shoot from the front of the ship, not the center
277        snd_cannon.play();
278        snd_cannon.currentTime = 0;
279    };
```

3. After the message is routed to the server, the server's `onFirePlayer` callback is invoked. The server creates a projectile object (line 256), broadcasts the projectile's coordinates to the other connected clients (line 259), **and** notifies the client that initiated the command of the projectile's coordinates (line 264):



```
253  // called when a player fires a weapon
254  function onFirePlayer(data) {
255      // we need the x, y coords of the firing player to select the initial position
256      var newProjectile = new Projectile(data.x, data.y);
257
258      // notify the firing player of the new projectile
259      this.broadcast.emit("fire player", {x: newProjectile.getX(), y: newProjectile.getY()});
260      // broadcast the new projectile to the other players
261      this.emit("fire player", {x: newProjectile.getX(), y: newProjectile.getY()});
262
263      // add the new projectile to the projectiles array
264      projectiles.push(newProjectile);
265  };
```

4. Finally, all connected clients execute callbacks to process the fire event in their own `onFirePlayer` functions:



```
// called when a player fires a weapon
function onFirePlayer(data) {
    var newProjectile = new Projectile(data.x, data.y);

    // add the new projectile to the projectiles array
    projectiles.push(newProjectile);
};
```

This same basic communication pattern is used to process all of the events in the system. We have shown here the player-initiated events that are triggered by user input with the keyboard: the other primary source of events in the system is a simple 2D collision detection (bounding circle) system that triggers events based on projectiles destroying asteroids, or players colliding with asteroids or friendly fire. All of these events follow the same basic communication pattern; the primary difference in the overall sequence is what actions the server must take to respond to the event appropriately.

The important aspect of the system captured by the UML sequence models is therefore
3. An *event-based* architectural style.

The UML Use Case Diagram is largely unchanged from the initial model.

Refer to Parts 1 + 2 of the assignment for a discussion of the Use Case Diagram and how it shows all of the system functionality at a glance.

ACME model: The updated ACME models are shown below. There are two ACME views: the high-level view, which shows the client(s) and server components and the connection between them, and a more detailed view of the server component.

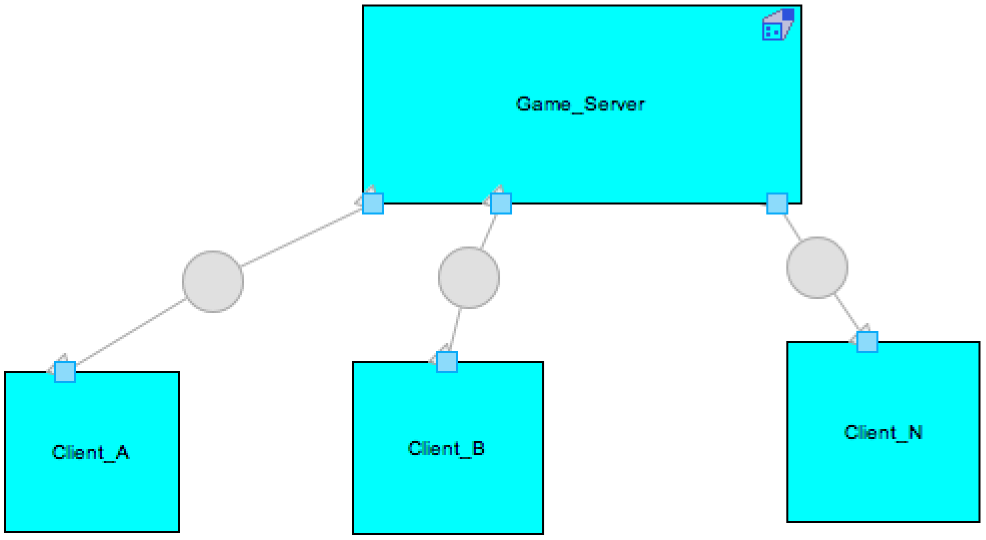


Figure 9. Updated view of Client-Server connection view Modeled in ACME

The ACME client-server connection view is shown above. This view shows the connection between the server and a variable number of clients – this is consistent with our updated xADL model and comes from our decision to change the game and have players join a squadron and work together to destroy asteroids. Note the single connector between them (the WebSocket connection) and its consistency with the other architectural views.

More detail about the nature of the server side of the application can be seen in the ACME server view below. This view has been updated significantly; before, we had modeled four ports, each of which can be used to connect a single WebSocket connection with a player, a Game_Engine component, to maintain the game state and rules / logic, and a clock to synchronize the system. The ports on the server component correspond to client 1, client 2, and … client n, since the system no longer has a limit of four players. The Game_Engine component we initially modeled has been replaced by the actual components on the server side: the game_js component, which does most of the game logic and environment simulation, the Keys_js component used to process user input, and the Asteroid_js, Player_js, and Projectile_js components, which encapsulate the logic for each of the major on-screen classes. The Clock component has been removed too, as we realized the entire system is event-based and asynchronous – the use of a lightweight transport protocol is used to make the game as near-real-time as possible, but there is no actual synchronization component to our system.

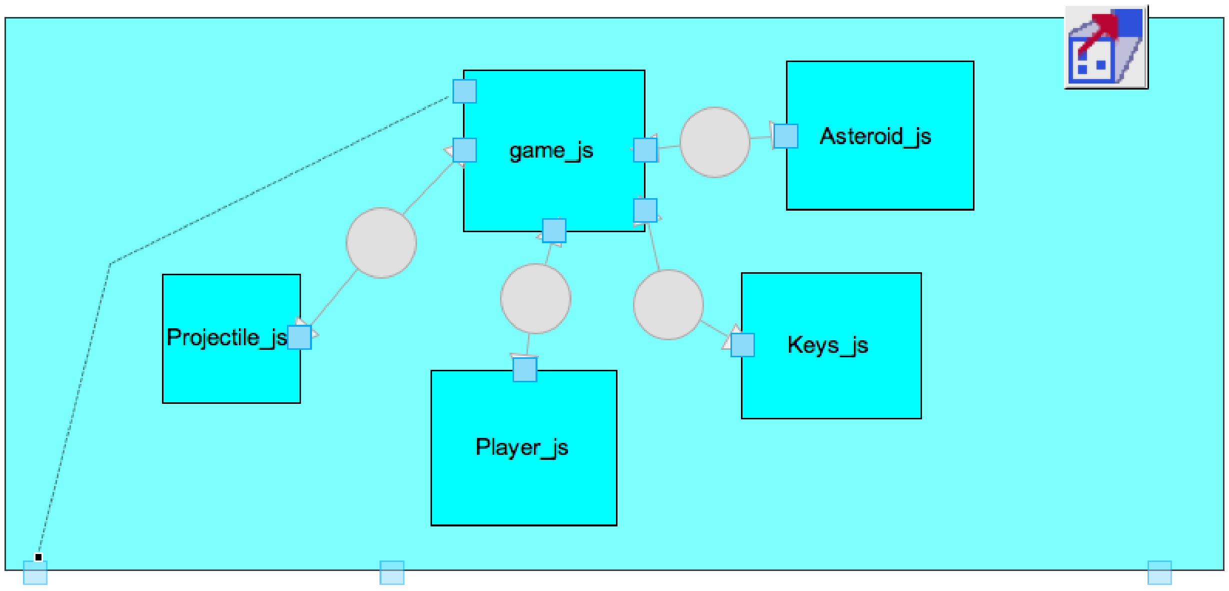


Figure 10. Updated Client-side view Modeled in ACME

[redacted] Assessment of the Implementation Experience:

Even after having chosen to use HTML5 and WebSockets, which we had found a great deal of information and thought we understood pretty well, once we set to work on actually *using* them in code of our own, there was a definite learning curve involved. One of the biggest obstacles at the start of the implementation project was figuring out how to install **Node.js** on a host to act as the server. We were confused by all of the tutorials we found online, and the number of Platform-as-a-Service (PaaS) providers who offer to provide Node instances on their own "clouds", but want to charge you for that privilege. Eventually, we figured out that our own machines would work just find as Node hosts for a small number of players, and we were able to move on to focusing on producing code and keeping it consistent with our architecture models.

Having some previous experience with JavaScript programming, I was comfortable with using it alongside **socket.io**, which was new to me. One of the nice things about event-driven JavaScript is that even though it follows a particular pattern, the freedom to create your own custom events and deliver just about any data payload alongside them through JSON means that the pattern is both highly customizable and highly structured. Having an understanding of the working of our code, tracing the messages in the UML sequence diagram to the source code was very easy to understand.

The consistency argument, then, seemed straightforward when considering our three primary architectural decisions enumerated above. The biggest updates and reworking that I realized we needed to do wasn't even a result of anything we wrote – it was incorporating the **socket.io** code into our models and keeping them consistent that way. When we chose to use WebSockets, we were looking at the raw API, and thinking that we could just use it directly by writing our own JavaScript. As we read more about what other people who are using WebSockets have done (and particularly people who were using them to make games), we realized that **socket.io** was very popular, fit with our initial architectural decisions, and simplified some of the code we had to write. In retrospect, I think it was a very good decision to include it in our project. But it does also serve as a reminder that in any project, you can gain knowledge and insight by surveying *everything* else that is out there and available, but at some point, you have to just start.

I feel that our arguments for consistency between our code and our models are actually pretty strong, and I would be confident in defending them. I think focusing on the connectors, and not just the components, when beginning to model helped in this regard, and is different than what I have tended to do in the past on coding projects. When we actually set about to *do* the implementation, I didn't feel that our models were very much help in directly assisting us in writing code. Some of this is probably attributable to the fact that we chose to keep our models at a fairly abstract level. But I also felt that the *process* of producing those models for Parts 1 + 2 of the assignment **was** very helpful in guiding implementation, because doing that work helped me understand much better what we were trying to accomplish and at least *how* the software components that we had never used before operated, before we tried to write our own code.

If I had to do this all over again, I actually think I might use xADL again. Just thinking initially about abstract components and connectors really helped me understand our project pretty well. Often what happens when trying to model something that has not been implemented in UML is that I or we (if I'm working with a group of developers) focus almost exclusively on the components (it's hard to break out of the Object-Oriented mindset when using UML), and the connectors go largely unexamined (and usually misunderstood). One of our goals in choosing to make a Web application was to become more familiar with the tools, and I wouldn't hesitate to try and build something for the browser again – modern web browsers and the new HTML standard provide a **lot** of powerful functionality to programmers, and not having to re-compile **everything** you write is so nice. On the whole, I did really enjoy this assignment and I felt like it was effective in trying to apply the ideas about software architecture that were discussed in class to a small-enough-to-be-manageable but large-enough-to-be-interesting project.

[redacted] Assessment of the Implementation Experience:

The overall experience was delightful and fun. I had a fantastic time working with HTML5 and WebSockets to make our game consistent with the architecture. The only difference was that some features weren't implemented from our original proposal, such as booster packs to offer a variety of weapons and health packs. Overall, the best part was making the game as close to real-time as possible and minimizing the lag during gameplay. Thanks to WebSockets and Node.js, the API offered superior scalability and synchronicity for many players and rapid screen updates of moving objects.

One of the influential factors that made us alter our game was a user interface issue. We already knew the arrow buttons would be used to move spaceships around the canvas, and the space bar would be used to fire weapons. Then we had the problem of how to let users rotate their space ships (because they were only firing upwards to the top of the screen). As a result, we altered the game mechanics: instead of players fighting against each other, they work together to survive through an asteroid field and avoid shooting each other.

The other delightful experience was working with Node.js and HTML5, something on my bucket list for some time. I was also in charge with setting up a server for us to test our game, and this was a huge challenge. I tried setting up Node.js on servers at UCI and UC Berkeley, but got stuck because of permissions. Then I signed up for a free-tier account with Amazon's EC2 service, but was overwhelmed with the options. In the end, we used one of our machines as a local host and other devices would connect to it.

The consistency argument is fairly easy to make since our game uses the classic client-server architecture where clients connect to a server, and the game play is controlled from the server. Our implementation abides closely with the model made in all of our software architecture models from UML, xADL, and ACME. I cannot say I am 100% sure that the code is consistent with our architecture, but I can say I am 90-95% confident. We are using the right connectors and components, all are linked appropriately, and more. The only changes made to the model were stripping away certain components that we didn't have time to implement. For instance, our original proposal had booster packs to help players, but we removed that due to time constraints. The model did help us a lot since it guided our implementation throughout the process. When it came to debugging, we were able to quickly identify the source and fix it. If we had to do this all over again, I think we would keep things the same as it is, but add more game physics to make the game more challenging and fun!

Appendix – Stats and Screenshots

Altogether, the system's code consists of:

- `index.html`: the webpage that hosts the game.
- Four server-side JavaScript files:
  - `game.js, Asteroid.js, Player.js, and Projectile.js`
- The Socket.IO node module code.
- Six client-side JavaScript files:
  - `game.js, Asteroid.js, Player.js, Projectile.js, Keys.js and requestAnimationFrame.js`
- Three CSS files.

The working system contains 8,520 lines of code (LOC):

- 1332 lines of code written by us
- 7188 lines of code in the socket.io node module package

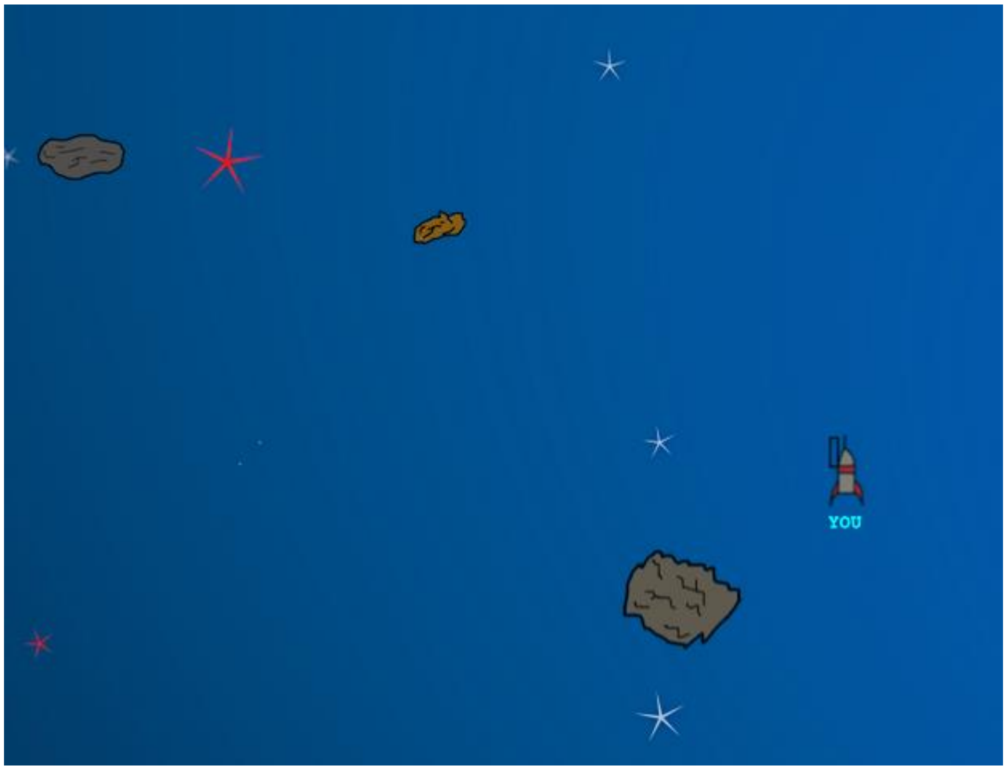


Figure 11. Close-up view of Spaceship and Asteroids

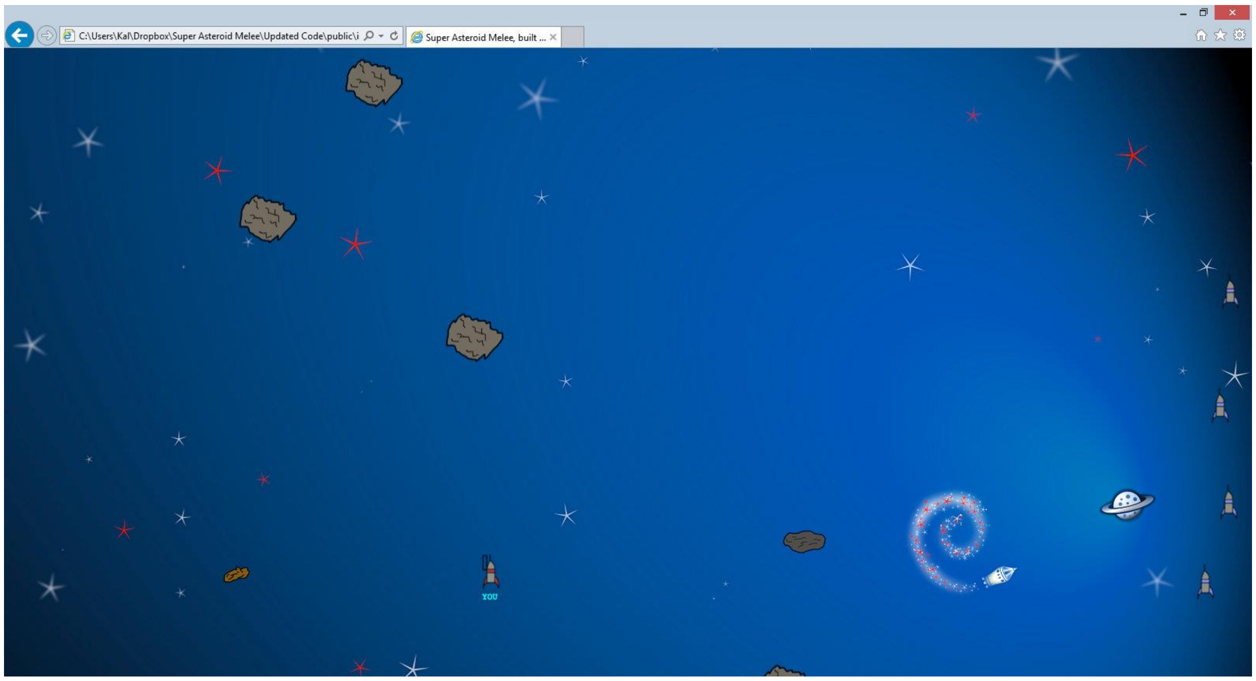
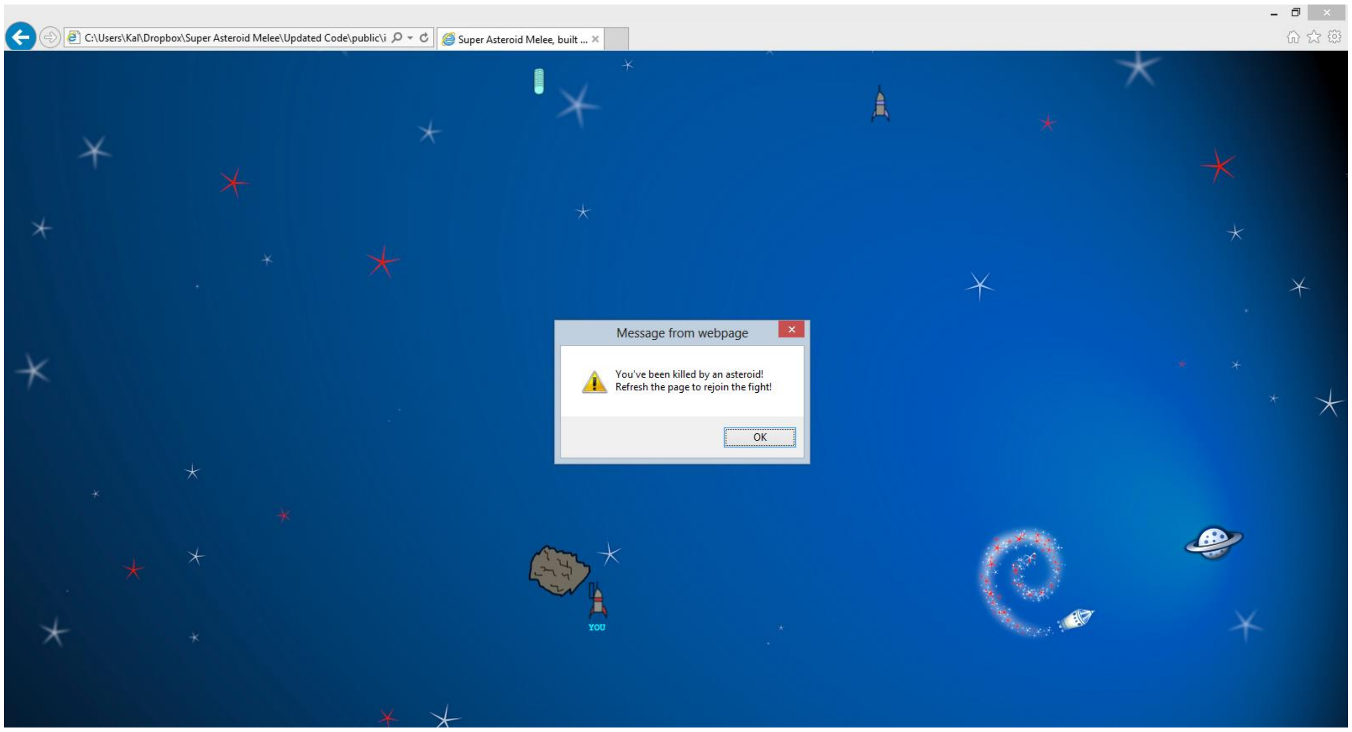

C:\Users\Kal\Dropbox\Super Asteroid Melee\Updated Code\public\i
Super Asteroid Melee, built ...
YOU

Figure 12. Multiple players connected (the other spaceships are hiding to the right of the screen)


C:\Users\Kal\Dropbox\Super Asteroid Melee\Updated Code\public\i
Super Asteroid Melee, built ...
Message from webpage
OK
YOU

```
You've been killed by an asteroid!
Refresh the page to rejoin the fight!
```

Figure 13. Our Noble Hero is Killed by an Asteroid