Retro Roman Zombie Apocalypse:

A Real-time Multiplayer Web Application for Saving the Colosseum from the Undead

Technical Report #TR-20140416-1

Robert Ribeiro
Department of Computer Science
Indiana University
South Bend, Indiana

Graduate Independent Study
CSCI-Y 790
Faculty Advisor: Dr. Dana Vrajitoru
May 3, 2013

I. Introduction

Retro Roman Zombie Apocalypse is a real-time multiplayer web application that has been developed with a unique, distributed game model through the use of cutting edge libraries, tools, and concepts. It has been the product of an effort to learn more about the platforms that are driving the direction of the software industry. It is no secret that the web is one of the fastest growing sectors of the software industry. However, what may be surprising is the role that game development is playing in that growth area. According to data from IBISWorld, social gaming is the fastest growing sector overall for 2013 [1,2]. While this has been developing, increased efforts in platform development have produced interesting and high powered tools for web development. One of the most popular and fastest growing of these is Node.js [3]. This, combined with HTML5 and a number of other frameworks and libraries, has allowed for the production of a game that performs well in real-time on minimal server hardware with a unique data model. This paper will explore the evolution of the project concept and the requirements established for its development. It will also explain the various tools and libraries used to make this idea into a real product. Finally, it will cover

the details of the development process and plans for future work.

II. Motivation and Theme

This project has always been intended to be a game, and Node.js was always the intended target. However, the original proposal was far different from the current incarnation. The idea began as a text-based multi-user dungeon game [4] that could be played over Twitter. After the limitations of the service prevented that from proceeding, the idea was transitioned to a turn-based game of a similar style that could be played from a web page. As the exploration of JavaScript and Node.js continued, two libraries were discovered that changed the course of the design. First, Socket.IO provided the opportunity to have real-time communication with the server and through the server to other clients. This changed the direction to allow the game to be played in real-time but still text based. EaselJS was the second design impacting library. It enabled 2D animation with reasonable effort in a familiar syntax. This opened the door for a side perspective “Beat ‘em up” melee game in the spirit of games like Double Dragon [5]. The remaining design was a combination of penchants for pixel art, ancient Rome, and the undead.

III. Requirements and Game Model

Having established a theme and gameplay style, some form of data model designs needed to be established in advance. One of the key issues for deployment was the expense of acquiring time on a server capable of running a game of this nature. Linode [6], a well-known virtual server provider charges $20 per month for their entry level server instances. Because these costs, the decision was made to utilize an offer from Amazon to have access to low-powered EC2 micro instance (currently limited to 613 MB of RAM and one CPU core) [7], which could fill the roll of a Node.js server, for free for one year. This meant that one of the requirements was to make the server as light as possible in order to run on one of these lightweight virtual servers.

This need for an ultralight server process was the leading influence on the game’s data model. Most multiplayer games have a central host or server which manages the state of the game and the communication between clients. Due to the limitations of Socket.IO, pure peer-to-peer communications are not possible in most browsers. At the time of the writing of this document, the only production stable browser with that capability, available through WebRTC instead of Socket.IO, is Google Chrome [8]. This meant that the server still needed to facilitate all communications. However, storing the game state for a multitude of games in memory was seen as a significant risk given that only 613

MB of memory would be available for the entire system. This meant the development of a new, unique model for multiplayer game state management.

One option that presented itself was the idea of having one of the players serve as the host. However, this was undesirable for two reasons. First, this is typically used when it is possible to form peer-to-peer connections. In addition, there appeared to be significant issues with trying to reliably pick a new host and transfer control of the game state upon spontaneous disconnection of the host. Instead, a distributed state model was proposed. Each player would be capable of maintaining a portion of the game model and sending it to the other players. This means that each player generates their own zombies and periodically sends a message to all of the other players updating their own position, direction, and other state as well as the position, direction and other state of all of the zombies under their ownership. Damage to enemies would then be shipped back to the owners for them to maintain their portion of the game state. What each player actually sees as they play the game is the composite of their own model and all of the partial models received from the other players. This model allows players to drop in and out without causing renegotiation of host configuration, while maintaining the other advantages of storing no game state on the central communication server.

Such a model does have a penalty for the server in the form of network transmission complexity for the central server. On a regular basis, n clients will send a message to the server which is then rebroadcast to n-1 clients. This gives the situation where the number of messages sent/received by the server for a single round of message distributions has complexity $O(n^2)$. Over the course of the game, however, m messages are sent by each player, which gives an overall complexity of $O(n^2m)$. Because of the potential for complexity to explode and the need to keep performance reasonable for each client, the number of players was set to four, capping the player count contribution to complexity on a per game basis. Given n players playing the game, the number of messages sent/received by the server over the course of the game drops to 4nm, which brings the complexity down to O(nm). With enough players or enough messages, even this could pose a threat to a low powered virtual server. Seeing that it would not be desirable to have to cap the total number of players in all games, a requirement was set to attempt to minimize the number of messages sent without compromising game performance and smoothness.

Since JavaScript is the language that powers both both Node.js and most client-side browser scripting, it should not come as a surprise that the decision was made to use only JavaScript for all of the programming.

This presented an additional level of difficulty as the developer has no prior experience in JavaScript. On the topic of JavaScript, one of the principle libraries that powers many websites is one called jQuery. It allows developers to manipulate the Document Object Model (DOM), which the W3C, a governing body for web standards, defines as "The Document Object Model is a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents. The document can be further processed and the results of that processing can be incorporated back into the presented page" [9]. In addition it provides methods for the simplification of the process of making asynchronous requests for data from a server. As an added challenge, it was decided that jQuery should not be used directly by this project, but rather, other methods of interacting with the DOM and server should be found and used.

Finally, as was mentioned earlier in this section, the number of players for the game was capped at four. However, all interesting and scalable games have some means of allowing for more than one game at a time to allow for more players. For this reason, a final requirement that the game support multiple, simultaneous game rooms was established. This was decided to mean that not only would it need to be possible to run concurrent games, but also users would have to be able to enter and exit rooms on their own. Thus, the basic requirements were laid out: the game would use a distributed state model to facilitate running the main server on an Amazon EC2 micro instance while avoiding jQuery and allowing for multi-room support.

IV. Libraries and Tools

A large part of this undertaking was a matter of finding the right tools for producing such a product. This, of course, includes the libraries and frameworks supporting the application, but it also involves development environments, project management platforms, source control, and user analytics. This section is meant to cover what value each tool provided in the development process and why it was chosen.

Node.js [10] is a network application server platform built on Google's V8 JavaScript engine. It specializes in non-blocking I/O and scalability, making is very useful for real-time applications. At the same time, it sits on top of a language that is accessible to a wide audience, given JavaScript's long history on the client side of the web. Node.js has been growing by leaps and bounds since its inception in 2009. NPM, the package manager for Node, has over 29,000 packages and almost 10 million package downloads per week at the time of the writing of this document [11]. It was realized that Node had unseated Ruby as the

en vogue web application platform by following the Hacker News feed [12] prior to the beginning of this project. With Node having reached this important of a status and offering such efficiency for server side processing, it was the natural choice for a game like this.

While Node.js provided a great platform to build on, it provides no structure for doing so, which is why Express was also selected for this project. Express is an MVC framework for building web applications on Node.js [13]. This makes it analogous to the purpose that Rails serves for Ruby programmers. MVC is a pattern for organizing modern applications. MVC stands for Model-View-Controller [14]. It allows for the separation of the concerns of the presentation to the user (the view) from the backend representation of the data (the model). In the middle, the controller acts coordinates taking in user input, processing data changes with the model, and reflecting those changes back to the view. It helps keep code cleaner, and is a foundational pattern for modern software and web development. Express made it possible to easily define directories for housing script files and images. In addition, it bundles jade, a powerful view engine, and LESS, an engine for more flexible and sensible CSS. It also offers many other features, such as the ability to define routes to individual controllers. This means that a developer can easily dictate what controller is triggered when a user visits, MyWebStore.com/products/ vs. MyWebStore/myAccount/. With the need to separate the viewable web page creation from the complex logic for a game server, Express was another must-have.

One thing that Express does not offer, however, is real-time communication. In order to have a real-time multiplayer game, there has to be a way to transmit data between clients on a very regular basis. This is where Socket.IO fits into the picture. Socket.IO is a JavaScript library that allows for easy to use real-time communications that works on a variety of browsers [15]. It was mainly designed around the WebSocket, an HTML5 specification allowing full-duplex communications using a TCP connection [16]. Socket.IO is usable on both the client and server, allowing for JavaScript objects to be sent back and forth with the same syntax on both sides and without any serialization efforts. Another advantage is that for less compliant browsers it has the ability to fall back Flash Socket, AJAX, and other legacy methods of communication. These capabilities made it essential for inclusion in this project.

The final piece that actually cemented the decision to use Express and Socket.IO was the discovery of Express.io. Express.io is the marriage of Express and Socket.IO into a single framework. It provides a simpler setup on the server side, which was helpful due to

the large number of options and minimal documentation presented by Express. In addition, it provides a single syntax for declaring regular routes and real-time communication routes. This made it possible to write more understandable server server code. While Express.io was not essential to the equation, its ability to lower the barrier to entry was a welcome find.

Having covered the major parts driving the server components, it is worth exploring the components of the user interface next. The first of two important libraries here is EaselJS. There are 2 methods of dealing with HTML5 graphics, and they are analogous to the old debate between raster and vector graphics. SVG allows for vector graphics in the browser, while the HTML5 canvas element allows for raster style web animation. EaselJS interacts with the canvas element with an API similar to the ActionScript API for Flash by Adobe [18], which is both easier to use and allows for more complex functionality, such as bitmap animations, texture atlases, and depth manipulation. These features are what enabled the project to go from text based to graphics based without incurring the overhead of building a graphics engine from scratch, making it another essential component of the game.

The other front end component is one that was added later in the project. Knockout is a library built on the MVVM pattern that allows for automatic DOM manipulations based on a binding model [19]. MVVM, has some similarities to MVC. The first two letters still mean the same thing: Model and View. However, in place of the controller, the VM stands for “view model”. The view model acts as an intermediate between the view and the model in a more event oriented style. Knockout creates a binding between the view and the view model. Then, when the view model is changed, the changes are automatically reflected in the user’s view. Similarly, user interactions with the view trigger events on the view model to interact with other pieces of the application’s back end. This was an important compliment to EaselJS, because EaselJS was only concerned with the canvas element. Knockout allows for the updating of text fields, such as score and health, as well as switching in and out view elements. This is what allows the game to run without ever reloading the page. Data received from game logic or from the server is pushed into the view model, causing the display to be updated without any need to refresh. This kind of automaticity and flexibility to manipulate the DOM, combined with Socket.IO’s communication capabilities, allowed for the fulfillment of the “no jQuery” requirement and with less complexity.

Another late entry to the list of libraries is Lo-Dash. As the game developed, an inheritance hierarchy was established,

with both Player and Zombie being children of Character. However, it was useful to store all of the players and zombies in a single array of characters. However, it was often necessary to get a single player or zombie out of this array, knowing only the value of an id or other property of the target object. Lo-Dash provides a list of functions for querying and manipulating arrays with much higher efficiency than even certain native JavaScript functions [20]. It actually has its origins as a drop in replacement for another library, underscore.js. It sacrifices compatibility with older browsers in favor of slimness and performance. Given that HTML5 was a requirement for this project to have access to the canvas element for graphics, there was no need to require much backwards compatibility. Thus, Lo-Dash was chosen to speed up the development of sections where array manipulations were important.

The final library is a small one, but it served an important function in guaranteeing data correctness across clients and within the server. Node-uuid is a library with one purpose: creating universally unique ids (UUIDs) that statistically have a near zero chance of producing collisions. This was important to ensure that no two zombies had the same id, particularly since they would be simultaneously generated by multiple clients. In addition, this library was used server side to solve the problem of how to identify individual game rooms and prevent collisions with those ids. It was not necessary for player ids, as those ended up being the same as the actual players' Socket.IO socket ids.

Beyond these libraries, a few other modern tools were employed to manage and develop the project in line with modern development standards. Trello [23] was chosen to assist with project management. Many modern agile projects focus on proper division of requirements into manageable bite-sized tasks. Having an online board to track these little pieces makes it possible to drop in a new idea that may be another week down the road with ease. In addition, it leaves a history of what has already been done, so getting back up to speed on what progress has been made is easy.

As for development environments, WebStorm by JetBrains [24] was the environment of choice. The developer had previous experience with JetBrains' products which have the advantage of autocompletion, code analysis, and refactoring tools, such as scope-sensitive variable renaming. In addition, their most recent releases have excellent support for Node.js, Express, jade, and LESS, all of which are already part of this project.

Finally, the last, but still critical part of any modern development effort is source control. While many people will debate their favorite source control platform, the last few years have shown that git has come ahead as a clear winner in the popularity contest.

Launched in 2008, Github had overtaken both Google Code and SourceForge by 2011 [25]. Thus, any project that is up to current standards would be expected to use both git and Github for source control, which is exactly what was done for this project.

V. Development

With the tools explained, it is now worth understanding the development process and a few of the challenges. As the development process started, there were two approaches to be considered: vertical slicing and horizontal slicing. Vertical slicing is the practice of implementing small bits of functionality across all impacted layers before moving on to the next piece of the project. On the other hand, horizontal slicing involves building complete layers and then building the interfaces between them. Given the fact that horizontal slicing is generally frowned upon and the fact that the distributed game model was so central to the game, a distinct effort was made to develop the project in vertical slices.

During the planning process, three major stages were established. The first was to create a working multiplayer prototype. Second was to create an actual working single multiplayer game. Finally, the third stage was to wire in the ability to have multiple game rooms running simultaneously. The multiplayer prototype started with just figuring out how to display a graphic on the canvas element, then how to move it. That led to moving it on another client's screen, which then turned into each client having individual control of separate entities on the screen. This, combined with better timing provided the basis for sending movement between clients and keeping their games in sync. While these steps were taking place, much of the basics of the server were laid out. During this first stage, it became apparent that a good motion model would be necessary for the game to run smoothly across a network without constant updates. This led players sending each other not only position information but also direction vectors, so that the character could continue to be moved in the same direction at a slightly slowed pace. This allowed the character sprite to already be about 90% of the way to the next update position most of the time, reducing the perception of lag and jitter. This also gave way to the idea of "heartbeat" updates. If no change in direction vector is made, sufficient smoothness could actually be achieved by only sending an update every 500 milliseconds. This contributed to the goal of reducing the number of messages sent significantly. In addition, restrictions were made so that all data to be sent to other players generated during one tick of the game clock were sent as a single message through the server to the other clients. This section of the project code was largely dependent Express, Socket.IO, and EaselJS

to get the basics running. Node-uuid was also used as a player id before the socket id was discovered to be available and useful.

With a working multiplayer prototype formed, the next stage, creating a single multiplayer game, started with the movement to a more object-oriented model. Previously, player data was just an anonymous JavaScript object that was interpreted on both ends by convention and character class concerns leaked into the main game logic. With the character specific logic pulled out, player and enemy animations were drawn and packed into a single texture atlas, which was then animated with EaselJS. During this stage, numerous issues and bugs were found and overcome. At a number of points, methods were implemented to ensure that animations ran smoothly and as expected, instead of terminating without completion. An algorithm for sorting the characters by depth on the stage was devised as well. Enemies started as a single sprite added to the canvas, and were then expanded to be able to move on their own, target the closest player, attack, and die. In addition, the original multiplayer prototype had been focused on sending information out to other players. With the addition of attacks and damage, information now had to be shipped back to the player that “owned” the zombie to maintain synchronicity of the distributed game state. Also, in order to manage the issue of concurrency among updates being sent between players, dead characters were tracked upon death to avoid being accidentally “revived” to the stage. It was about this time that the concept of player health and points came into play and the first Knockout view model code was added. In addition, Lo-Dash played a number of important roles during this phase for finding various elements and sub-collections in the character array.

The final stage was to expand this single game into multiple game rooms. At this point the view itself got much more complex as more Knockout was used to build the room construct. In addition, Lo-Dash became an important server side element, as working with an array of rooms, each containing an array of players belonging to the room, became crucial to the multi-room model. During this phase, the most critical problem to be solved was player disconnection. Unlike all of the other real-time routes used in this project, the disconnect real-time event does not take a data message that can be used to identify the sender. In order to solve this issue, the socket used to start a game had to be trapped in a JavaScript closure [26] for reference later during the disconnect event. Other parts of this stage included sending the room list and being able to refresh it, switching some of the major JavaScript libraries to download from Content Delivery Networks (CDN) instead of adding traffic to the game server itself. Finally, the site

structure was completed, including CSS and layout of the various elements, as well as citing all of the frameworks and libraries on the web page. With all of the work completed, the game was deployed to an Amazon EC2 micro instance running Ubuntu, and setup to receive traffic from our privately registered domain name. This completed the development process for this project.

VI. Future Improvements

While we are very satisfied with the design and performance of the final product, it is our belief that no software is ever truly finished. With that in mind, there are a number of future proposals that allow this to continue to be a work in progress beyond the life of this graduate independent study. Currently, the game supports a fairly flat difficulty model. It would be preferable to allow the difficulty to scale up over time, increasing the frequency of enemy spawns, the frequency and range of their attacks, and the total health with which they spawn. While the lag compensation model is currently sufficient, a more variable model, one based on the actual differences in the timing of messages received and the distance of the jump in sprite position caused by the updates, would likely smooth out some of the remaining jitters in the gameplay. Of course, in order to make this ready for a full public release, an additional level of polish would be preferred, including a title screen, proper loading protocols, and actual death animations. Finally, it would be desirable to back the site with a database to allow not only a blog and other site features, but also the ability to allow for player accounts, high score tracking, and other multiplayer features.

VII. Closing

Retro Roman Zombie Apocalypse was a unique experience among the typical coursework of a graduate level Computer Science program. It permitted the full development of an idea into a fully deployable product. In addition, it promoted the exploration of new libraries and tools, as well as honing the skills necessary to produce quality software. Most importantly it fostered the ability to make architectural and algorithmic design choices not normally available to a student developer.

Web References

[1] http://www.forbes.com/sites/caroltice/2013/02/07/what-business-should-you-start-fast-growing-sectors-for-2013/
[2] http://www.ibisworld.com/media/2013/04/16/top-10-fastest-growing-industries/
[3] http://mashable.com/2011/03/09/node-js/
[4] http://en.wikipedia.org/wiki/MUD
[5] http://en.wikipedia.org/wiki/Double_Dragon
[6] http://www.linode.com/
[7] http://aws.amazon.com/ec2/
[8] http://www.webrtc.org/
[9] http://www.w3.org/DOM/
[10] http://nodejs.org/
[11] https://npmjs.org/
[12] https://news.ycombinator.com/
[13] http://expressjs.com/
[14] http://www.codinghorror.com/blog/2008/05/understanding-model-view-controller.html
[15] http://socket.io/
[16] http://www.websocket.org/
[17] https://github.com/techpines/express.io
[18] http://www.createjs.com/#!/EaselJS
[19] http://knockoutjs.com/index.html
[20] http://lodash.com/
[21] https://github.com/broofa/node-uuid
[22] http://en.wikipedia.org/wiki/Universally_unique_identifier
[23] https://trello.com/
[24] http://www.jetbrains.com/webstorm/
[25] http://readwrite.com/2011/06/02/github-has-passed-sourceforge
[26] https://developer.mozilla.org/en-US/docs/JavaScript/Guide/Closures