Web Server

Intermediate Report

Scott Dunham, Michael Lim, Eli Mallon,
Kimberlee Redman-Garner, Logan Jones

I. **Abstract**

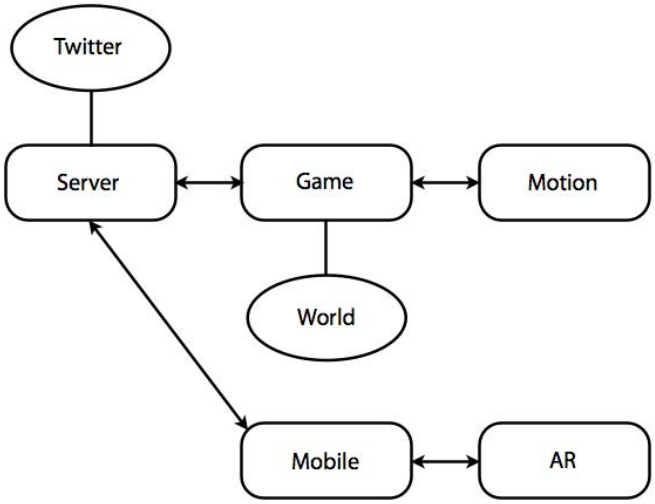
The Web Server's primary task is relaying game state information between mobile devices and game server. Additionally, the web server hosts a web portal that allows many types of user interactions outside of the actual game play such as leader boards, play-by-play and even a type of marketing for the game that will explain the purpose and plot of the game and serve as a public, web-based "face" for Vi-Char. One of the more difficult features to implement available on the web interface is the twitter module.

Currently, each of the modules in our product function as separate node.js servers. Implementation of the servers will be described in detail in this document. The messenger module will be doing the bulk of the work in the client/server relationship between the Game Server and the Mobile Devices to achieve a networked, fast and interactive game. This include the need for a structured database, and a consistent data format that can be read and written between all clients. Game state data is sent to the server via a GET request. This will return a JSON object containing relevant game state data. In order to change the game state, a POST request can be sent to update aspects of the game state.

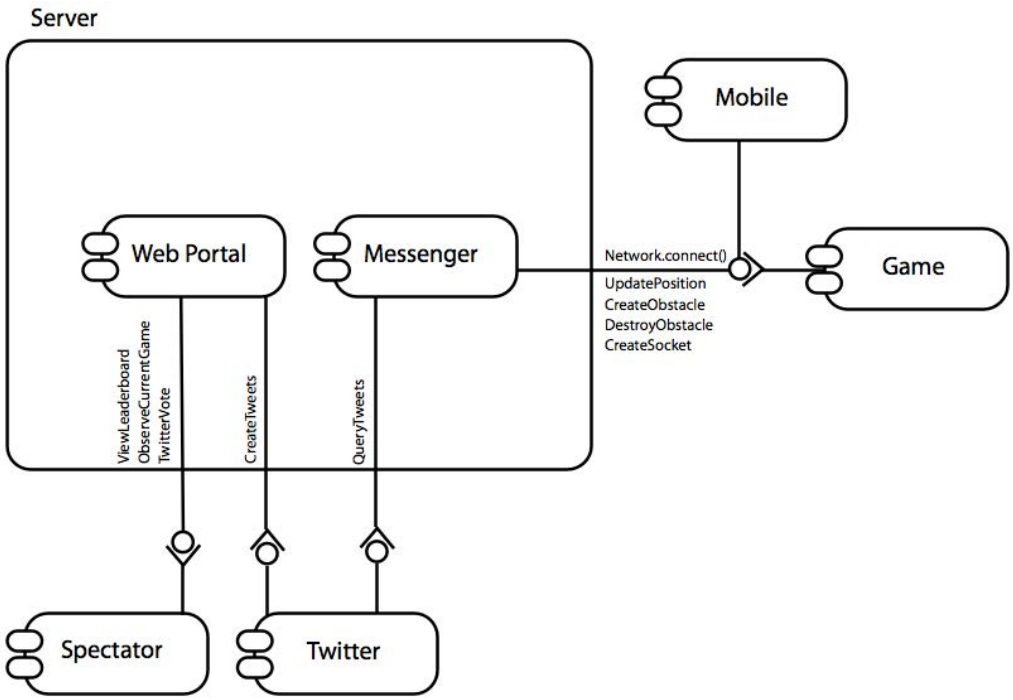
The web portal is the aforementioned "face" of the Web Server and will be a place where game information will be aggregated, organized and presented to be viewed by the public. Twitter is utilized to integrate larger numbers of players into the game. We have utilized sockets to retrieve tweets containing a certain hashtag, and display them on the main splash page. The major point of the Web Portal is to complete the gameplay experience in a manner that will include all users interested in the game.

These three modules make up the higher-level design of the Web Server architecture that will complete the game experience as a networked, multiplayer game and also promote the entire Vi-Char project within the class and beyond. This document will provide relevant implementation, design, and procedural detail in order to replicate and understand the current product.

II. **Design Summary**



On a global level, the web server will primarily interact with the game engine and the mobile devices. The server provides a messaging system that allows game information to be updated by the game and mobile devices as events occur within the game. When game state data is received by the server, it will be sent to the appropriate place.



The architecture of the web server is comprised of three main modules. The messenger will provide an interface that allows game state information to be received from the game engine and mobile devices, and sent to relevant devices. For example, when the player moves, data will be sent from the game to the server. The messenger will then update the stored position coordinates for the mobile devices to retrieve. The web portal provides interfaces for users. Users will be able to view leader-boards and check the score of the current game. Users will also be able access twitter via the web portal and tweet certain phrases to influences the game. For this reason, twitter is providing an interface to create tweets to the web portal. Twitter is also providing an interface to the messenger to allow tweets to be searched. When relevant twitter information is received, this will also be send from the server to the game and mobile devices.

III. Install Documentation

Prerequisites: Installation of node.js via apt-get or yum.
The following walkthrough will allow a developer to install and run our product.

- Clone Git repository via https://github.com/UPS-CS240-F12/WebServerGroup.git
- Install Forever module globally ("npm install forever -g")
- Start the various Node.js servers by executing this command for each item below: *[node|nodemon|forever start]* +
    - json_server.js (messenger)
    - turret_test.js (messenger for turret coordinates)
    - twitter.js (web/twitter modules)
        - twitter.js runs on port 80, which is only allowed if it's run as root (e.g. "sudo forever start twitter.js"). Running it as root is a *temporary* workaround, which will eventually be replaced by forwarding port 80 traffic to the port we move it to.

IV. Implementation Details

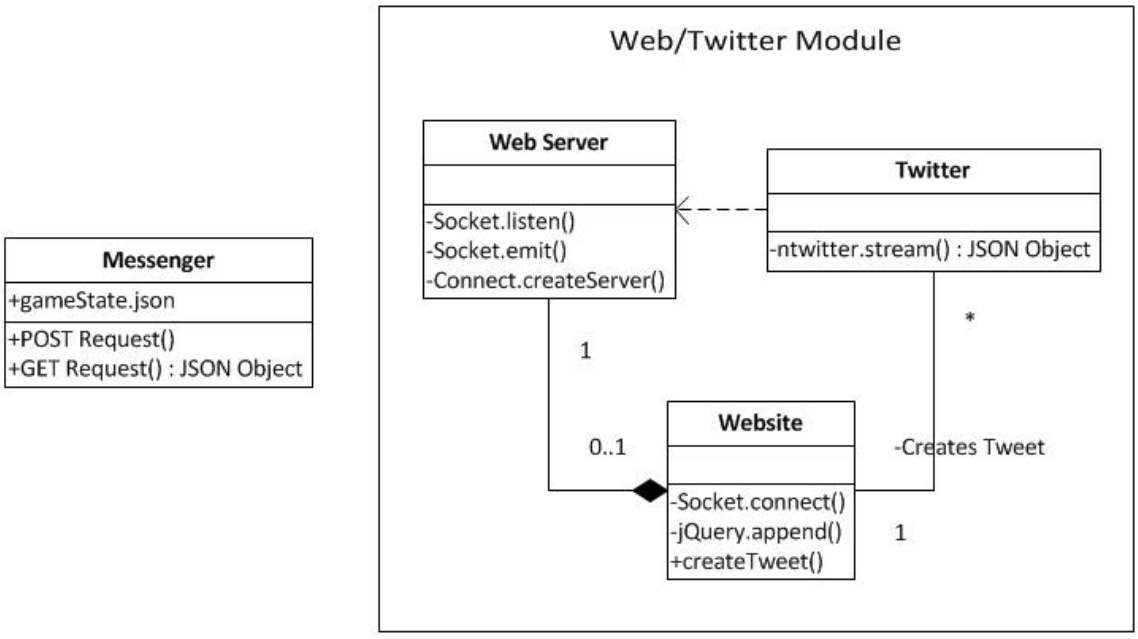


Figure 1: This figure shows a basic UML Class Diagram for the Web Server Group's work thus far. The technologies we are using don't lend themselves to this sort of diagram, but this diagram attempts to approximate the structure of our system. The Web and Twitter modules have been grouped together, since they are contained within the same Node.js file currently (see below). The Web Server owns the website, and the two are connected via a WebSocket so that Twitter data can be passed from the server to pages on the website. Users are also able to create tweets via the website, which get fed to the server via the Twitter Streaming API. The Messenger owns the gameState JSON file, which can be requested by the other groups using a GET request. Elements of gameState.json can be added/changed using a POST request.

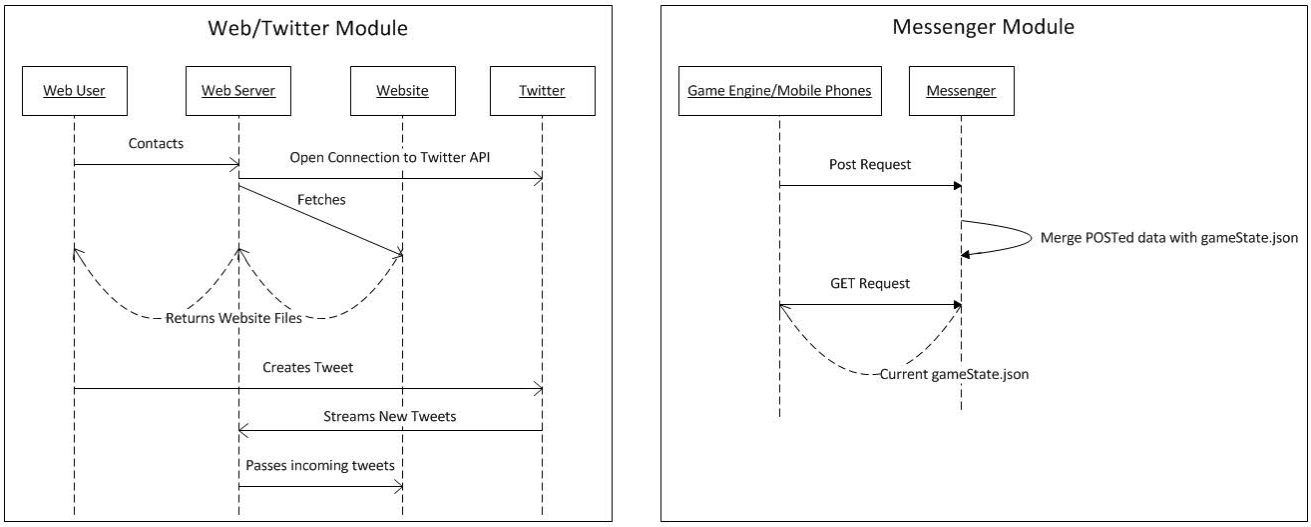


Figure 2: This figure shows a UML System Sequence Diagram of our system as it exists today. The Web/Twitter Module diagram shows the sequence of events when a web user connects to our website. When a user visits vi-

char.pugetsound.edu. the web server simultaneously fetches the website files and opens a persistent connection to the Twitter API. The user is able to browse the website and create tweets. These tweets are then fed back to the website via our server's API connection, and inserted into the page dynamically. The Messenger Module shows how incoming POST requests from the Game Engine and/or Mobile phone groups are merged with the current game state, which can then be returned to these groups via a GET request.

All of the web group's modules are individual node.js servers.

Messenger Module:

The Messenger maintains an HTTP server on port 1730.

Internally maintains a gameState just as a node object. Incoming GET requests return a JSON representation of this object. When POSTed a JSON object, it iterates over the incoming object and makes all the changes to the existing object. Data can be any of the supported JSON data types. The server will parse them to ensure the JSON is valid, then merge it into the gameState object.

Node.js objects are hash-based, so with an existing gameState of size m and an incoming JSON object with n data points, average case is O(m) and absolute worst case is O(n+m). The object is kept in memory and not frequently written to disk, so if we discount network latency the bottleneck will only be in parsing the incoming game object. It's crazy fast. Initial testing has confirmed it can handle 10,000 rapid-fire GET requests made from ten different clients in less than 2ms per request.

From a *technical* standpoint, the Messenger is very simple. From an *API* standpoint, the Messenger is more complex.

There are a few potential problems with node's async nature. There is no guarantee that the data you POST to the server won't already have been modified by the time your request is processed. For this reason, we are creating a document entitled "gameState etiquette." It can be viewed on our wiki here, but the fundamental tenets are as follows:

- Only modify the gameState within your own namespace–unless you really, really know what you're doing. Basically, for 99% of use cases, you will only modify parts of the game tree that you "own" as documented by the evolving API.
- The game server owns the formal state of the game. This means that when a player wants to take some action–say, placing a turret–they will create a request that the game server will then parse and instantiate the turret into the world.
- JSON Objects are the only data structure you need. JSON supports arrays, but asynchronous modification makes these problematic. If you want to delete entry 2 from an array, but another client has appended something to the top of the array, you'll delete the wrong entry.

Strictly speaking, all of these components are optional–if you really know what you're doing, there's nothing in the API preventing you from modifying whichever parts of the gameState that you please. That said, adherence to this etiquette can be considered a best practice, best allowing for iterative development and future compatibility. As with real life, if you breach etiquette, make sure you have a very good excuse for doing so.

Web and Twitter Modules:

The Web module consists of a simple Node.js server (twitter.js) which serves out a directory of HTML, CSS, and JavaScript files on port 80 (the standard HTTP port) using a module called Connect. The web interface currently consists of two pages- the home page (index.html) and the current game page (current.html). Both of these pages would be relatively static if it weren't for the inclusion of content being fed in via the Twitter API. At this time, the Twitter module is completely integrated with the Web module as the website is the primary interface through which users will interact with it. These modules will need to be split out in the near future, however, as the Mobile group has expressed an interest in allowing users to vote via Twitter using their interface as well. Additionally, greater fault tolerance measures will need to be developed. While there isn't much we can do if our server crashes, we have implemented all of our Node.js server files using the forever module, which automatically restarts any node server which crashes. Beyond this we haven't implemented much in the way of fault tolerance, which is potentially an issue in the case of our integration with the Twitter API. In the future, we'll need to be able to gracefully handle any downtime Twitter might experience.

Tweets containing the hashtag "#vichargame" are fed to the website using a combination of two Node.js modules- ntwitter and Socket.IO. The ntwitter module is designed to utilize the Twitter API in a number of ways. Our implementation takes advantage of the Twitter Streaming API to open a connection between our server and Twitter through which any tweets containing the hashtag "#vichargame" are passed in real time in the form of a JSON object.

Once our server receives new data from twitter, it is passed along to both pages using the Socket.IO module. Socket.IO is a Node.JS implementation of WebSockets- an API intended to allow clients and servers to "push messages to each other at any given time" (http://davidwalsh.name/websocket). On the server side, we designated a port (1337) on which Socket.IO would "listen" for connections initiated from the client side. When a user connects to our website, a connection is established with the server on the specified port in a simple JavaScript file. Using this established connection, the server "emits" any incoming Twitter data, which is then received and utilized on the client side. On the homepage, any tweets received via this connection are inserted directly into the page using jQuery methods. The current game page also inserts these tweets, but separates them based on how we have envisioned the Twitter voting system working. Tweets which also contain the hashtag "#robot" are put in one section of the page as a vote for the robot team, and tweets with "#eye" are placed in another as a vote for the mobile phones.

To facilitate the twitter voting system mentioned above, the current game page has two buttons which allow users to tweet the appropriate hashtags by clicking on one of two

buttons. Clicking on these buttons utilizes Twitter's Web Intent tool to provide a simple interface through which the user can write and submit a tweet with the appropriate hashtags to vote for their chosen team. There is currently no connection between our Twitter and Messenger modules, though this will be relatively simple to implement in the future. One potential way to implement this is to hide these buttons, and reveal them when a vote is called. After a predetermined amount of time, the number of tweets for each team can be counted and data indicating the winning team can be sent to the Messenger module using a simple POST request.

V. Procedures Followed

Development Responsibility:

We have been working on the following modules, with certain people taking the lead.
-Web/Twitter modules (currently integrated-- see above): Scott
-Messenger: Mostly maintained by Eli--Scott modified for the demo.

Tools used:

-Source Control: Like the rest of the divisions, we have been using Git for our source control. This has been pushed to the shared remote repository on Github.
-Documentation: Mainly through comments in code. A more formal API is being codified using the Github wiki.
-Organization/Reports: We have used Google Docs/Gmail to coordinate meetings and assignments, and to collaborate on our written reports

Testing & Verification:

Given the size of our code base, large scale testing has not been an issue yet. The individual working on a given piece of code usually is able to test it completely before finishing. As of yet, we have not had to deal with too much integration, but the use of our public API means that other groups have been able to interface with our servers with little to no tweaking.

Coding Style Guide:

Again, given the amount of code we have had to work with, there hasn't yet been a need for a formal style guide. In addition, much of our current work has been with proof-of-concept pieces and prototypes. As we move on to more permanent solutions, we will begin implementing stricter guidelines. For now, here are some basic rules we have tried to adhere to:
-Naming:
Variables should use lower camel case, constants should be all in uppercase. Keep variable and function names descriptive. Avoid using single letters or ambiguous abbreviations
-Comments:
Functions should have a preceding comment describing the overall purpose of the function, as well as pre and post conditions if necessary. Have comments for any line that's purpose would not be immediately obvious to someone from a different division.
-Miscellaneous:

Try to keep open braces on the same line as the statement it is a part of. Declare one variable per line for readability sake. Try to use triple equality (===), to avoid implicit type-casting. For non-trivial boolean conditions that may be reused, assign the value to a variable, rather than doing the evaluation inside an if statement. Avoid unnecessary nested if statements by returning as soon as possible.

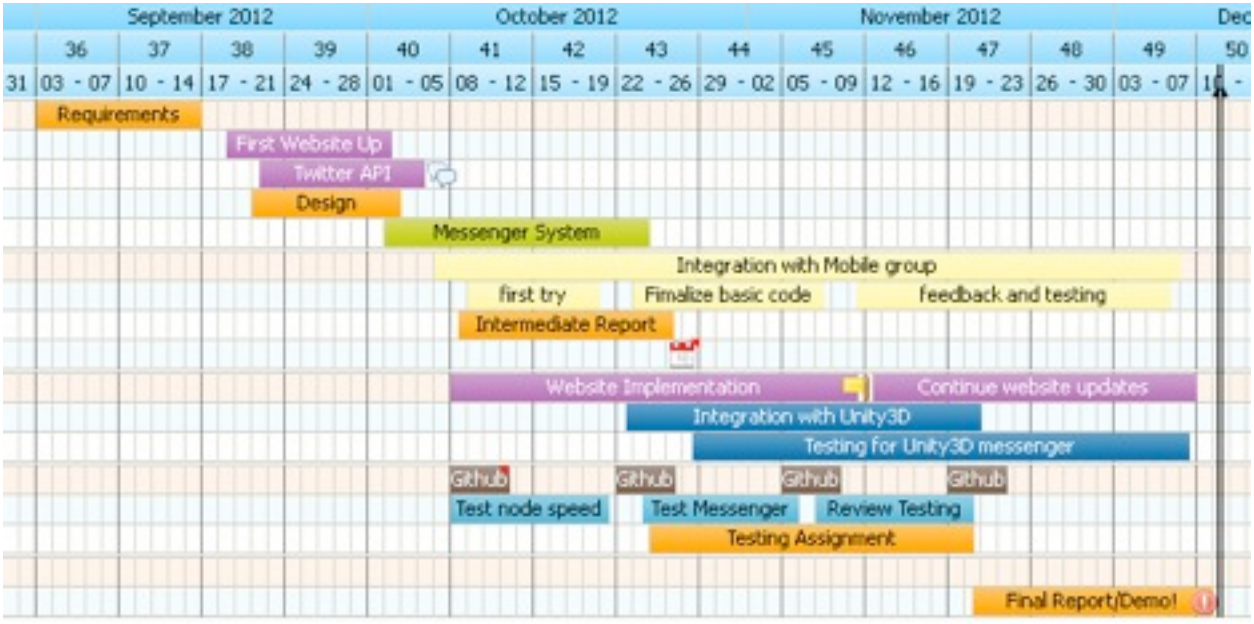
VI. **Reflection**

Overall, integration between groups has been the hardest part to see through. The challenge is definitely something we are capable of although it was not an immediate start to integrate. Looking back, starting this earlier would have been better but it is hard to see how to have done that exactly as we were getting to know our project still.

One major challenge has been an easy way to implement Node.js with Unity3D. Although that hasn't proven to be un-doable so far, it is not built-in to Unity3D like many other modules of Unity3D are. The networking set-up for Unity3D is well supported for JSON files, though, so we have devised our own method for this by supporting the Game Design team's API with a simple HTTP server solution to the Messenger module we use to host an ever-changing JSON file on our server for the game engine and phone users to request and change as needed.

VII. Timeline

In our timeline below, there have been updates to every aspect of our work. The largest one being the level of detail. Previously, it was difficult to provide detail without knowing implementation strategy.



VIII. Glossary and References

JavaScript Object Notation (JSON) - A plain text notation for data storage and exchange. Structured similarly to a Map, with key/value pairs.
HTTP Server - a computer program that delivers (serves) content, such as web pages or data, using the Hypertext Transfer Protocol (HTTP).

Connect Node.js Module - http://www.senchalabs.org/connect/
Twitter Streaming API - https://dev.twitter.com/docs/streaming-apis
Twitter Web Intents - https://dev.twitter.com/docs/intents
ntwitter Node.js Module - https://github.com/AvianFlu/ntwitter
Socket.IO Node.js Module - http://socket.io/
WebSockets - http://davidwalsh.name/websocket
jQuery - http://jquery.com/