"State Machines": A High-Throughput Distributed Game Framework

by

Brandon Ivy

ELECTRICAL ENGINEERING DEPARTMENT

California Polytechnic State University

San Luis Obispo

2014

Contents

List of Tables

List of Figures

Abstract

Online multiplayer games allow competition with other opponents using the internet. Their development is difficult as it involves many complex areas of computer science - optimization, security, big data, and thread safety. Many multiplayer game frameworks have been published, but very few are generic.

In this project, we focus on the framework design of a web service which allows users to easily create multiplayer games, and artificial intelligence bots to compete in them. It handles turn-based multiplayer communication, and allows the games and bots to ignore the concepts of multiplayer systems. Multiple game instances can run simultaneously, where each game's current state is the internal representation for a game at that point in time. This web site (named throughout as "State Machines") provides users with an interface for connecting with one of many actively running state machines.

I. Introduction

State Machine: A model of computation consisting of a set of states, a start state, an input alphabet, and a transition function that maps input symbols and current states to a next state. Computation begins in the start state with an input [1].

Game: In the context of this report, a game refers specifically to a program written to simulate the internal representation of a specific game, chess or tic-tac-toe for example, as a state machine. The internal workings of the program are unimportant, but it must follow a particular input-output specification. To summarize the full specification, the program must accept two JSON[1] objects and return two JSON objects. The two inputs are a current state and a move, and the outputs an updated state and data to send to the player who's move is next.

API: An abbreviation of Application Program Interface, an API is a set of routines, protocols, and tools for interacting with software applications. The API specifies how software components should interact. A good API makes it easier to develop a program by providing all the building blocks. A programmer then puts the blocks together [3].

This project seeks to implement a web service that will provide users with an interface for connecting with one of many actively running state machines. The service runs user-written games on its servers, and bootstraps in online multiplayer support.

In this application the main focus is on representing different turn-based games using the state machine model. Multiple instances of a game can run simultaneously, where each machine's current state is the internal representation for a game at that point in time - including position of players on a game board, move order, and any other game-specific information. The "moves" the players make in a game represent events which cause the game to make a transition from one state to another.

Relatively few frameworks are designed specifically for turn-based games. Most can be configured to handle them, but restricting the framework to turn-based only opens up opportunities for optimization. Response time is often the goal of real-time frameworks, but by definition turn-based games have no concern for this statistic [4]. This allows for a much more focused design methodology prioritizing throughput.

The representation of the game as a state machine simplifies data handling for the backend of the system [5]. Specifically it allows for the use of a distributed task queue[2], which is an obvious point of horizontal scalability. The absence of response time as a game element also allows the system to process moves sequentially, similar to a pipeline. In this case game logic, the bottleneck of the pipeline, is easily scalable by simply adding more workers to the task queue. In addition, once a move has been processed by a game, the state can be stored and the game will block until

[1] JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write, and easy for machines to parse and generate. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages [2].

[2] A system for parallel execution of discrete tasks in a non-blocking fashion. There are three main elements in the system, a broker, which holds the tasks, any number of producers, which create tasks, and any number of consumers, who perform tasks the broker is storing.

another move has been received. This greatly reduces the system resources required for a single game, allowing a single server to run many games simultaneously.

While the game logic itself runs on the backend of the web server, clients run on users' computers and communicate via the game API. As the common endpoint is an API, clients can be either artificial intelligence "bots," written by users and run on their own computers, or graphical front-ends that allow human players to play.

The project goal is to create a central service that facilitates autonomous competition across many games. Online multiplayer games are common today, but require error-prone code to handle the complexities of multiplayer connections. In most cases, this code achieves essentially the same task. The "State Machines" system handles as many tasks as is practical of a generalized multi-user state machine. This allows users to write programs that focus simply on the game logic itself, and at the same time inherit multiplayer functionality and a web endpoint anyone can use to play the game against opponents. The system scales easily, it simply requires the initialization of more servers to connect to the system and "consume" tasks. In the scope of this project, this is a manual operation; it is trivial to automate it. Ultimately it will be publicly accessible via the internet, and will allow anyone to either compete online via the game API or create games of their own, and view the results in a web browser.

II. Background

II.1 Previous Examples

There are several previous examples of other multiplayer game frameworks. The most similar to "State Machines" is named RTF as it is a "Real-Time-Framework" for games [6]. RTF is focused on response time for the suability of real-time[1] games. Real-time games, unlike turn-based games, require low latency in order to guarantee usability and overall fairness.

Researchers at the University of Münster developed an experimental game, Rokkatan, to test a proxy-server architecture for a massively multiplayer RTS[2] (Real-Time Strategy) game [7]. Their implementation ran multiple copies of the each game simultaneously on different servers to handle inputs from many players, and communicated amongst themselves to keep the game state updated. This model is good for RTS games, which are for the most part real-time. Even though there are turn-based components, modern RTS games cannot be represented with a turn-based architecture.

II.2 Main Differences

Most multiplayer frameworks are not very generic - that is, they assume and require the game to fit certain characteristics that are not included in the simplest definition of a game. This is because generic multiplayer frameworks, in most cases, are not very useful. Frameworks of this type must be permissive - since nothing is assumed about the game itself, the system simply passes moves around. Thus permissive frameworks leave some extra work to the coder, since sanitization must be done manually.

This project instead targets turn-based games in order to simplify the codebase and improve throughput. In turn-based games, the delay between moves is not important. The removal of response time as a requirement allows for optimizations that greatly improve overall throughput. One example is the use of large distributed queues to handle game tasks, which are horizontally scalable. From the user perspective, the slower move response time does not prevent game throughput as bots can participate in any number of simultaneous games.

All communication required for competition uses a rest API, which makes the system compatible with many different languages without difficulty. This allows for the creation of bots as well as GUI (Graphical User Interface) front-ends for the game messaging system. In effect, a player can write a bot to play tic-tac-toe and have it compete against thousands of other bots in a very short time. The highly generic nature of the messaging system allows for educational applications as well, for instance the entire "Wumpus world" AI teaching environment could be implemented as a game and the system would handle any multiplayer aspects as well as score keeping and statistic generation. This would remove almost all dependency[3] requirements for students to begin coding in the environment. Students could write bots to compete against each other, and the system would automatically record scores and milestones while allowing professors to easily and interactively demonstrate different implementation strategies [8].

[1] A game where time progresses continuously according to the game clock. Players perform actions simultaneously as opposed to in sequential units or turns.

[2] Real-Time Strategy: A type of game where moves do not progress in turns. Players position and maneuver units under their control as quickly as they can send commands.

[3] The degree to which each program module relies on each one of the other modules.

III. Requirements

The website must allow users to submit programs of their own, which will then be run by the server against other similar programs. As this requires actions performed by a largely automated backend and users will be expecting real-time feedback, a sizable amount of effort is put into code testing to ensure reliability and predictability. Therefore a large percentage of the code, almost half, is dedicated to test cases. Development in this style is slow, but deliberate and implicitly documented [9]. The methodology behind this is that the time put into test cases should pay off in the form of debug time later [10].

III.1 Marketing Requirements

As a website is only successful if there are other users to compete against, it must be enjoyable to use as well as functional. The marketing requirements satisfied by this project are as follows:

1. The site must be easy to use.
2. The site must be easily maintained and improved.
3. The site must be reliable.
4. The backend API must be generic enough to handle many game types.

Table 1 lists the engineering and marketing requirements of the site. The requirements' justifications are listed in the third column. All engineering requirements are the direct result of at least one marketing requirement.

Table 1: "State Machines" Requirements and Specifications

Marketing Requirements	Engineering Requirements	Justification
3	The site will be thoroughly tested using unit tests and distributed load testing to simulate realistic use cases.	Thorough testing ensures that already written code works as expected, and decreases debug time by narrowing the scope of problems.
2, 3	The site will be written using as many frameworks and libraries as is practical.	Concise code is easier to debug and teach to new developers who may contribute in the future.
1, 2	The front-end of the site will be minimal, with emphasis on conciseness and simplicity.	A low learning curve encourages new users and increases the practicality of the system's use in college AI classes.
4	The game API will not interpret game data, only pass it from player to game and back.	Reduces the amount of information that can be inferred from the API by the server, but it also lowers overhead and increases compatibility.

IV. Design

IV.1 Methodology

The use of the python web framework *Django*[1] simplifies the design of the entire system. Notably the separation of models guarantees that extra time spent in one place will not need to be repeated. The features in one *Django* model are easily accessible by other models and reduce their development time [11]. The specific schema used by the *Django* models for communication is specified in Appendix B.

IV.2 Front-end

Figure 1, an overview of the system's interconnections, shows that the front-end (website) is minimal. There is no fancy web development or graphic design - emphasis is put on the backend of the site for reliability and features. The front-end allows bots to be registered, and games created, uploaded, and configured. It also displays results, as well as statistics (defined per game) on the bots, as well as their relative ranking. This data can be used to compare overall bot performance. A more detailed website map is provided in Appendix A.

IV.3 Backend

Bots first connect to the API to initiate a game, and they are be placed in a queue. The message the bot receives from the API contains the url the bot should use to receive the first state. The "matchmaker" - a *Django* model on the web server - pulls bots off of the queue and organizes them into "lobbies" by game type and ranking. This portion of the system is particularly open to expansion. Specifically, it would be convenient if bots could request specific opponents. Currently the matchmaker uses a greedy best-first algorithm to match opponents, but the framework model makes it very easy to re-write the logic used to pull bots off the queue.

When a lobby is created the first game update request is passed over to a *celery*[2] queue, which manages work load distribution. There are multiple queues, one per game type, as the queues are lightweight [12]. This further eases the complexities of load distribution by allowing for identical workers. Every worker is capable of running any game, but workers can be instructed to listen to specific queues if it is necessary to focus more processing power towards specific games. This situation will likely become common once the site goes public, as different games inevitably have different popularity levels, and therefore varying resource demands.

Upon receiving a move, a worker will check which game type the request is destined for. Once found, it will start an instance of that game, if it is not already running. In the case where a worker is running a specific game type for the first time, it will automatically download the binary for that game from the static data server and store it locally for next time. Games are designed to run indefinitely, they simply process moves passed to them until otherwise terminated.

When a lobby is created it is passed over to a "referee," which all of the players will connect to directly. This referee is currently implemented as a *Django* model as this gives it implicit access

[1] A free and open source web application framework, written in Python, which follows the model/view/controller architectural pattern.

[2] An open source asynchronous task queue/job queue based on distributed message passing. It is focused on real-time operation, but supports scheduling as well.

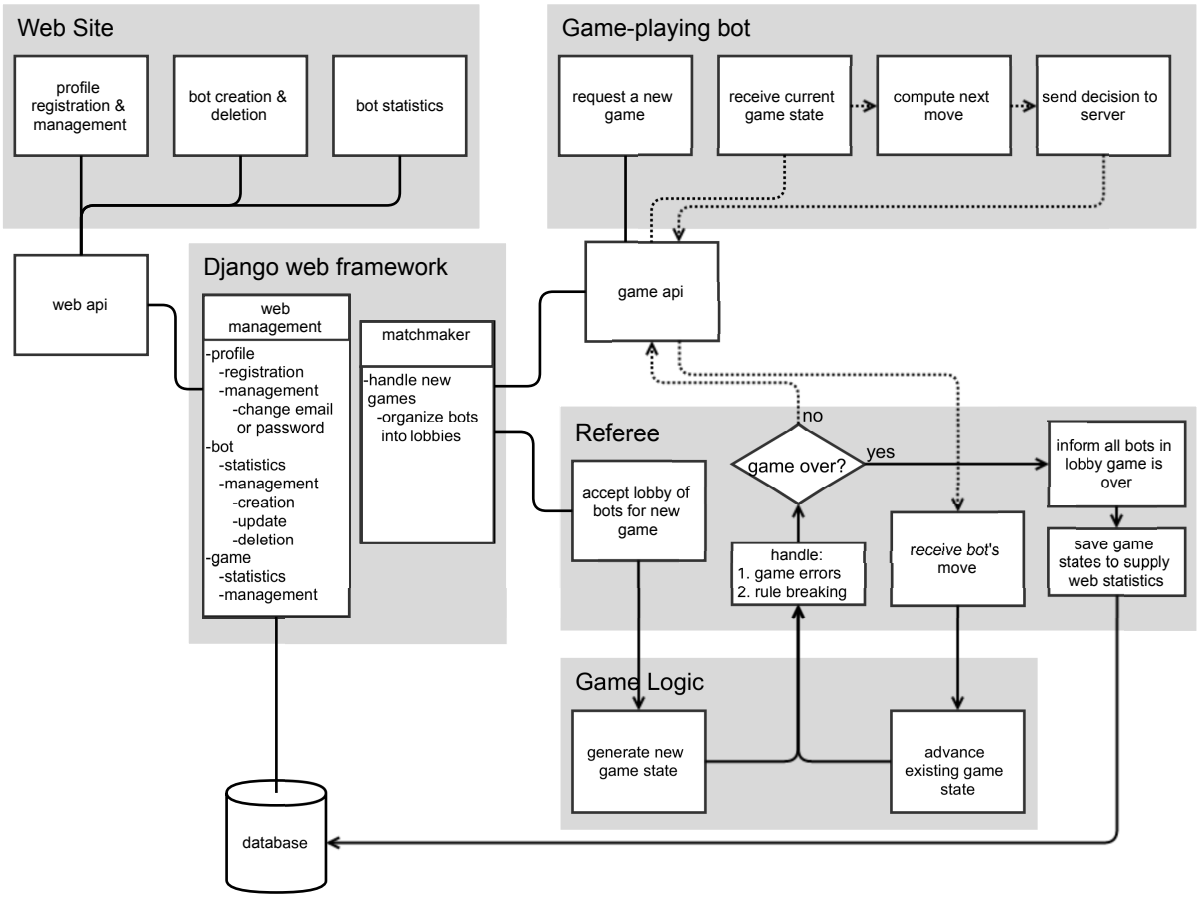

Figure 1: System Interconnections and Responsibilities

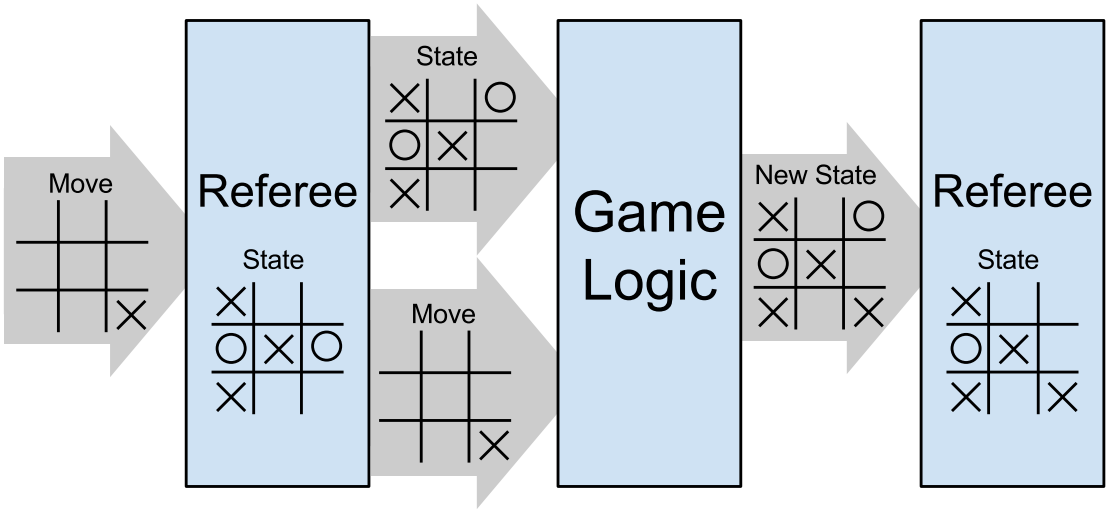

Figure 2: Illustration of a Move's Effect on a Game State

to other functionality on the site, and minimizes the layers of separation between it and the end user. The referee will be in charge of accepting and authorizing moves from players. To prevent cheating, it will also store the current state of the game. On each move, the referee will pass the move and state to a worker which will then produce a new state as well as an output. The output is then passed to the next player, who makes a move and the cycle continues.

When the game is finished the referee will send statistics (and optionally a replay) back to the central server which integrates the statistics into the database.

In order to facilitate distribution of computation, reduce latency, and make response time predictable, each game process will be completely stateless. The referees will be in charge of receiving new moves from bots and remembering each game's state (positions of players on the board, etc). This means that every new move sent to a referee will be passed to the game with the stored state. The game will produce a new state and an output for the next bot, as in Figure 2. The new state is stored by the referee, and the output is passed on. This allows one game process to handle moves from different referees, and therefore different game instances, interchangeably.

IV.3.1 Usage

To communicate with the system, users write bots which connect via the API to play games. The specifics of using the game API are described in section VI.2.2. The following is an example bot in Python, which plays a game called "Guess Coin Flip." The rules of the "Guess Coin Flip" game are very simple, the player simply guesses heads or tails. If he guesses correctly, he gets a point. Incorrectly, he loses a point. It is a two player game, so the first player to three points wins.

The language chosen is not important, Python is used here simply for clarity. Bots and games can be written in any modern language, the only requirement is that it support JSON[3] and HTTP[4] communication.

---

[3]JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write, and easy for machines to parse and generate. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages [2].

[4]The Hypertext Transfer Protocol (HTTP) is an application protocol for distributed, collaborative, hypermedia information systems. HTTP is the foundation of data communication for the World Wide Web [13].

```python
import json
import requests
from time import sleep

# unique API key generated by the server, provided for each user
api_key = "67POtKhv/T/Z9fDviYUC+rWHcTJddL6w"

# url specifies joining game id 2, as bot id 1
join_url = "http://statemachines.site/game/2/join/1"

post_headers = {'content-type': 'application/json'}

# join a game by going to the join_url and providing an API key
server_response = requests.get(join_url, auth=('apikey', api_key)).json()

while True:
    try:
        # try to get the URL to send the move to
        move_url = server_response["move_url"]
    except:
        # exit if there is a bad response
        Exception("Bad response from server")

    # here is where AI decisions would be made, if the game has any strategy
    # always guess heads
    payload = {"guess": "heads"}
    # send our move
    server_response = requests.post(data=json.dumps(payload),
                                    move_url,
                                    auth=('apikey', api_key),
                                    headers=post_headers).json()
    sleep(1)
```

Since the "State Machines" system keeps track of statistics, a simple while loop is enough to allow this bot to play games indefinitely, until it is manually terminated. When each game is over the "move_url" key in the server's response will direct to the location a move should be sent to start a new game. Keeping track of whether or not the game is over, or who wins, is not necessary for a bot; this is taken care of by the server. By viewing the website, a user can compare his bot's performance to others, or view any errors the site detected relevant to his bot.

IV.4 Dependencies

The system will use as many external libraries as is practical to simplify the codebase and reduce bug occurrences by leaving more work to tested libraries. This also improves overall security compared to the alternative, which would involve reproducing the following libraries' functionality using new and unproven code. A list of the libraries and packages used by "State Machines" is shown in Table 2.

Table 2: Dependency List and Justifications

Package	Version	Justification
Django	1.6.1	Web framework
anyjson	0.3.3	JSON message support
billiard	3.3.0.17	
bson	0.3.3	Binary message support
celery	3.1.11	Distributed task queue
django-bower	4.8.1	Automated package management
djangorestframework	2.3.12	Rest API
gevent	1.0	Multi-threaded connection support
pytz	2014.2	
redis	2.9.1	Distributed Key-Value store

V. Test Plans

V.1 Unit Testing

Unit testing is used to verify the functionality of simple tasks throughout the code base. Most, but not all, sections are tested individually to reduce test development time. Some elements of the system are tested at a higher level, to ensure multiple sections work at once. This achieves most of the benefits of full test coverage at a fraction of the development time cost [14].

Testing is done through the *Django* testing framework, which is based on the Python-builtin *unittest* [15]. It is based on the *JUnit*[1] testing methodology and allows easy testing of numerous tasks requiring setup and teardown of environments. It minimizes code-reuse by allowing for nested preparation and cleanup sections.

V.1.1 Front-end

The website uses a web API to fetch data from the database through *Django*, providing input sanitization[2]. All APIs use a Representational State Transfer[3] (REST) interface, and test cases use REST queries to communicate directly with each model. In this way all models are tested independent of the rest of the system.

V.1.2 Backend

Similar to the web API, test cases for the game API use REST queries[4] as well. Thorough testing throughout the process paid off greatly in the form of bug prevention - the simple unit tests implemented earned back the time required to write them many times over.

[1] A unit testing framework for the Java programming language. JUnit has been important in the development of test-driven development, and is one of a family of unit testing frameworks which is collectively known as xUnit. It defines how to structure your test cases and provides the tools to run them [16].

[2] Input sanitization is a coding technique which scrubs user input to prevent it from "jumping the fence" and exploiting security holes. Three of the top five most common website attacks - SQL injection, cross-site scripting (XSS), and remote file inclusion (RFI), share a root cause in common: input sanitization. Or to be more accurate, a lack thereof [17].

[3] A software architectural style consisting of a coordinated set of architectural constraints applied to components, connectors, and data elements, within a distributed hypermedia system [18].

[4] A query, in the context of APIs, refers simply to the act of requesting a response using the API.

VI. Specifications

VI.1 Web API

The web API is used mainly by the web pages and javascript loaded by users to simplify interactions to and from the database as well as enforce a separation between front-end (the interface a user sees) and backend (the complicated logic behind the scenes) functionality.

This API can also be used by the bots written by users. While this API has no connection to the game logic itself, it may be useful to read game metadata or stats from a bot. The details of the web API can be found in Appendix C.

VI.2 Game API

The game API is used by bots to connect to games and make moves. In order to make the API language-independent, all data is encoded in JSON unless otherwise stated. The game API is currently very broadly defined, in order to eliminate the possibility of inadvertently designing an interface incompatible with game types that have not yet been written. As the games themselves are arbitrary binaries, one option is to let each game designer implement their own game-specific communication protocol and have all of the bots of that game adopt the same protocol. This makes statistic gathering difficult, however, as each game would require custom rules to infer the statistics from the game state. The API is therefore slightly more specific than the most generic option, to make logic in the statistic and referee models simple.

VI.2.1 Bots

As of now, game "moves" generated by bots are only required to contain:

apikey the API key of the owner of the bot, given to a user upon account creation

data the actual move data used by the game, stored in a string

For security reasons, the actual URL each move should be sent to is unique. This prevents impersonation as other users. Bots request the first URL by querying the API's "join" endpoint using an API key to authenticate. Subsequent URLs are provided by the server in each response. Responses from the server contain:

move_url the URL the next move should be sent to, it contains a randomly generated string created by the server

game_data the player-specific data sent back from the game, which is used to make strategic decisions

VI.2.2 Games

The games themselves, on the other hand, see the other side of the game API. Since miscellaneous data in each message is used by the "State Machines" system itself, the formatting and content of the messages seen by the games are quite different than those sent by bots.

A message received by a game process is formatted as follows:

state the current state of the game, empty when a game is just starting

moves a list of moves made by bots since the last state change, indexed by player id. The list is empty at the start of each game.

The messages sent back from the game to the rest of the system must contain a minimum of:

state the new state of the game, after moves have been applied. This is stored by the system to be processed once the next move has been received.

player_data a list containing data to send back to players, indexed by player id. The system determines who is allowed to make the next move based on which players receive data.

score a list of scores, indexed by player id. This is used by the stats model to rank individual bot performance.

At this point the stats *Django* model, the section of the system dedicated to keeping track of aggregated scores for all games, uses the "score" field returned from the game after each move. The score data is stored in the database, and used later when users browse the site to view results. In the future the game API can be expanded to ask more stats from a game in order to generate more granular statistics. The score value will be left however to ensure backwards compatibility with games written before any API changes.

VII. Conclusion

The project's code is completely written, with lingering bugs. The API specifically has deadlock issues when several players are submitting moves in succession, which is necessary functionality for almost all turn-based games.

The deadlock occurs when waiting on an update from the move key-value store, which is waiting on the response from the game. To resolve this issue more testing must be done to determine why the response from the game does not correctly unblock the second user's connection.

Despite the bugs, throughput testing on the functional aspects of the API show that the system is currently very capable of high throughput without further optimizations. In fact, local testing on a laptop was not enough to adequately determine the throughput because the system was bottlenecking on the network interface bandwidth, not CPU, meaning that distributed testing is necessary to determine its full throughput capability. The code will need to be fully functional before this kind of testing is possible.

The web front-end is completely functional, but very barebones. No styling has been done for the site, so it is not production-ready as an externally facing website. The front-end is in place mainly for debugging purposes - finishing the appearance of the front-end of the site is trivial. This will be done once the API is complete and will take no more than 20 man hours.

As it is, the system is a functional reference implementation of a multi-threaded and horizontally-scalable multiplayer game framework which abstracts over client communication and overhead for turn-based games. The API's deadlock bugs prevent the system from being fully operational, but the logic is unit tested and the majority of the system is reusable as-is for other tasks.

The code is available in its current state at `http://users.csc.calpoly.edu/~bivy/sm.zip`.

VIII. References

[1] A. Kent and J. Williams, *Encyclopedia of Computer Science and Technology: Volume 25 - Supplement 10: Applications of Artificial Intelligence to Agriculture and Natural Resource Management to Transaction Machine Architectures*, ser. Encyclopedia of Computer Science Series. Taylor & Francis, 1991. [Online]. Available: https://encrypted.google.com/books?id=W2YLBIdeLIEC

[2] ECMA, *ECMA-404: The JSON Data Interchange Format.* Geneva, Switzerland: ECMA (European Association for Standardizing Information and Communication Systems), October 2013, 2013. [Online]. Available: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

[3] Authorize.Net, *Merchant Web Services SOAP API Documentation*, ver. 2.0 ed., Customer Information Manager (CIM), February 2014.

[4] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, pp. 299–319, Dec. 1990. [Online]. Available: http://doi.acm.org/10.1145/98163.98167

[5] A. Striegel and G. Manimaran, "Dynamic class-based queue management for scalable media servers," *Journal of Systems and Software*, vol. 66, no. 2, pp. 119 – 128, 2003. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121202000699

[6] F. Glinka, A. Ploß, J. Müller-lden, and S. Gorlatch, "Rtf: a real-time framework for developing scalable multiplayer online games," in *Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games.* ACM, 2007, pp. 81–86.

[7] J. Müller and S. GORLATCH, "Rokkatan: Scaling an rts game design to the massively multiplayer realm," *Comput. Entertain.*, vol. 4, no. 3, Jul. 2006. [Online]. Available: http://doi.acm.org/10.1145/1146816.1146833

[8] J. Laird and M. VanLent, "Human-level ai's killer application: Interactive computer games," *AI magazine*, vol. 22, no. 2, p. 15, 2001.

[9] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *Software Engineering, IEEE Transactions on*, vol. 28, no. 10, pp. 970–983, Oct 2002.

[10] N. D. Singpurwalla, "Determining an optimal time interval for testing and debugging software," *Software Engineering, IEEE Transactions on*, vol. 17, no. 4, pp. 313–319, Apr 1991.

[11] M. F. Sanner *et al.*, "Python: a programming language for software integration and development," *J Mol Graph Model*, vol. 17, no. 1, pp. 57–61, 1999.

[12] R. Belkin and V. Ramachandran, "Mechanism for implementing multiple thread pools in a computer system to optimize system performance," Apr. 1 2003, uS Patent 6,542,920.

[13] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Rfc 2616, hypertext transfer protocol – http/1.1," 1999. [Online]. Available: http://www.rfc.net/rfc2616.html

[14] Y. Cheon and G. T. Leavens, "A simple and practical approach to unit testing: The jml and junit way," in *ECOOP 2002—Object-Oriented Programming*. Springer, 2002, pp. 231–255.

[15] P. S. Foundation. (2014, May) Python unittest. [Online]. Available: https://docs.python.org/3/library/unittest.html

[16] junit team, *JUnit wiki*, 4th ed., 2014. [Online]. Available: https://github.com/junit-team/junit/wiki

[17] A. Weiss. (2012, October) Prevent web attacks using input sanitization. [Online]. Available: http://www.esecurityplanet.com/browser-security/prevent-web-attacks-using-input-sanitization.html

[18] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, 2000.

A. Site Map

This is a general overview of the site's structure.

- Site
  - Index - Description of the site, some game statistics
  - Tutorial
  - Stats
  - Contact us
- Users
  - Login/Registration with verification, OAuth
  - User Profile - Change nickname, emails, password, settings
  - Contact us
- Game
  - List - Sort on removable columns - similar to amazon product filter
  - Create/View/Edit - Game descriptions expand
  - Documentation Wiki - Integrated with real API
  - Stats
- Bot
  - List
  - Create/View/Edit
  - Stats - Queue positions
- Server/daemon management
  - Start/Stop/Restart
  - Loaded game binaries
  - Resource usage
  - Stats

B. Database Schema

```
CREATE TABLE "game_game" (
    "id" integer NOT NULL PRIMARY KEY,
    "name" varchar(24) NOT NULL,
    "slug" varchar(24) NOT NULL,
    "description" text NOT NULL,
    "players" integer NOT NULL,
    "owner_id" integer NOT NULL REFERENCES "auth_user" ("id"),
    "binary" varchar(100)
);
CREATE TABLE "game_gamestat" (
    "id" integer NOT NULL PRIMARY KEY,
    "game_id" integer NOT NULL REFERENCES "game_game" ("id"),
    "stat_id" integer NOT NULL REFERENCES "stats_stat" ("id"),
    "name" varchar(24) NOT NULL
);
CREATE TABLE "game_lobby_bots" (
    "id" integer NOT NULL PRIMARY KEY,
    "lobby_id" integer NOT NULL,
    "bot_id" integer NOT NULL REFERENCES "bot_bot" ("id"),
    UNIQUE ("lobby_id", "bot_id")
);
CREATE TABLE "game_lobby" (
    "id" integer NOT NULL PRIMARY KEY,
    "game_id" integer NOT NULL REFERENCES "game_game" ("id"),
    "total_slots" integer NOT NULL,
    "filled_slots" integer NOT NULL
);
CREATE TABLE "game_move" (
    "id" integer NOT NULL PRIMARY KEY,
    "token" varchar(32) NOT NULL,
    "lobby_id" integer NOT NULL REFERENCES "game_lobby" ("id"),
    "bot_id" integer NOT NULL REFERENCES "bot_bot" ("id"),
    "player_id" smallint unsigned NOT NULL
);
CREATE TABLE "bot_bot" (
    "id" integer NOT NULL PRIMARY KEY,
    "name" varchar(24) NOT NULL,
    "description" text NOT NULL,
    "owner_id" integer NOT NULL REFERENCES "auth_user" ("id"),
    "game_id" integer NOT NULL REFERENCES "game_game" ("id"),
    "created" datetime NOT NULL
);
CREATE TABLE "bot_botstat" (
    "id" integer NOT NULL PRIMARY KEY,
    "bot_id" integer NOT NULL REFERENCES "bot_bot" ("id"),
    "stat_id" integer NOT NULL REFERENCES "stats_stat" ("id"),
    "name" varchar(24) NOT NULL
```

```
);
CREATE TABLE "stats_stat" (
    "id" integer NOT NULL PRIMARY KEY,
    "duration" integer NOT NULL,
    "current_slot" integer NOT NULL,
    "num_slots" integer NOT NULL
);
```

C. Web API v 1.0 Endpoints

These are all of the endpoints currently available to the web API. The front-end site uses them to communicate with the database. The game API is a subset of the web API.

- \bot
  - \create
  - \delete
  - \< *id* > (PUT)
  - \< *id* > (GET)
- \task
  - \create
  - \< *id* > (PUT)
  - \< *id* > (GET)
- \game
  - \< *id* > (GET)
    - \join
      - \< *botid* > (GET)
- \move
  - \< *token* > (PUT)
- \matchmaker
  - \connect
- \stats
  - \< *key* > (GET)
- \user
  - \login
  - \logout
  - \< *id* > (PUT)