ORCHESTRATE: A SYSTEM DEMONSTRATING EASY SUBSTITUTION OF BROWSING SEMANTICS ENGINES

An Undergraduate Research Scholars Thesis

by

TATE ANDERSON BANKS

Submitted to the Undergraduate Research Scholars program at
Texas A&M University
in partial fulfillment of the requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by Research Advisor: Dr. Richard Furuta

May 2020

Major: Computer Science

TABLE OF CONTENTS

Page

ABSTRACT ... 1

DEDICATION ... 2

ACKNOWLEDGMENTS ... 3

NOMENCLATURE ... 4

CHAPTER

I. INTRODUCTION ... 5

- Browsing Semantics Engines ... 5
- Orchestrate ... 6
- Experimental Engines ... 7
- Literature Review ... 7
- Preview ... 8

II. DESIGN ... 9

- Goals ... 9
- Orchestrate Version 1 ... 9
- Orchestrate Version 2 ... 10
- Improvements to Orchestrate 2.0 ... 14
- Final State ... 21

III. TESTING ... 23

- Completed Testing ... 23
- Theoretical Applications ... 25
- Storyspace Comparison ... 28
- The Future of Orchestrate ... 29

IV. CONCLUSION ... 31

REFERENCES ... 33

ABSTRACT

Orchestrate: A System Demonstrating Easy Substitution of Browsing Semantics Engines

Tate Banks
Department of Computer Science and Engineering
Texas A&M University

Research Advisor: Dr. Richard Furuta
Department of Computer Science and Engineering
Texas A&M University

The Orchestrate project explored the separation of a browsing semantics engine from the other user-visible components of a hypertext or other linking system. This allows more flexibility in the semantics of the links themselves, enabling more variety with regards to conditionals behind links, and ultimately will greatly help the implementation of new and different linking semantics. The project builds upon much previous work in hypertext and linking semantics.

The Orchestrate platform was designed and implemented to model several new and novel linking semantics and behaviors. It was built with the goal of enabling further work and development to take place from within or without the program, and will very probably be used in future studies and enable future academic papers for submission to scholarly venues.

Once initial review, improvements and testing to the initial build of the Orchestrate platform was completed, additional features and tests using various browsing semantics methods were undertaken.

DEDICATION

This thesis, as many of my other endeavors at Texas A&M, is dedicated in gratitude to Craig Brown of the Craig and Galen Brown Foundation. His immense generosity is what enabled me to attend Texas A&M, and my time at this University has been the most formative period of my entire life. Thank you once again, Mr. Brown.

ACKNOWLEDGMENTS

I would like to thank Dr. Furuta, who was very willing to take me under his wing when I was looking for a research project to get involved with.

I would also like to thank several of my professors who got me very interested in computer science after I changed majors, including Dr. Dylan Shell, Dr. Philip Ritchey, Dr. Dmitri Loguinov and Dr. Martin Carlisle. I would not have been able to enjoy this field of study nearly as much as I have without their excellent teaching and phenomenal classes.

Finally, I'd like to thank everyone who enabled my Texas A&M experience and made it such a formative period in my life. Thanks go to my parents, my mentors and colleagues in the Student Government Association, all the fine roommates I've had, and the many friends I've made who I know will always be there when I need them.

NOMENCLATURE

Orchestrate	The system being built and tested in this project
… 1.0	Initial version of the system with limited functionality that already existed
… 1.5	Improved version of Orchestrate 1.5 with more flexibility
… 2.0	Version of Orchestrate that incorporated logical engines and their substitution
… 2.0.1	Initial functional build of Orchestrate 2.0
… 2.1	Version of Orchestrate 2.0.1 with significant additional functionality added
Media	Information that is presented to a user in some form or fashion
Engine	Logical backbone that determines the behavior of a piece of media

CHAPTER I

INTRODUCTION

Systems of electronic media presentation have long been a main focus of Dr. Richard Furuta's research and courses at Texas A&M University. He has authored and co-authored many papers on the subjects of electronic media and hypermedia, including "Petri-Net-Based Hypertext: Document Structure with Browsing Semantics" and "Display-agnostic Hypermedia," which have been published in academic conferences and journals dedicated to the study of electronic media systems. These systems can take many forms and use several different mechanics, including hypertext, speech-to-text, embedded multimedia, and more. Such systems have boundless uses – from use in online academic courses or as supplements to brick-and-mortar courses to e-books to website content to some forms of electronic advertisements, and much, much more. Such systems are commonplace and often taken for granted.

However, much research has gone into the design, implementation and improvement of such systems. One has only to look at web tutorials or message boards from the early days of the Internet and compare them to those of today to realize how far the presentation of electronic media has come due to this research. This is due not only to improvements in graphics and aesthetic design but also to developments in the design of features and logical backend systems associated with the platforms.

Browsing semantics engines

These logical backbone systems for the organization of the media, referred to hereafter as browsing semantics engines, or engines, and the exploration of new and different designs for them, are the main motivation for the Orchestrate project.

A large amount of current research in media presentation, and indeed the main focus of the literature review involved in this iteration of the Orchestrate project, revolves around these browsing semantics engines. These provide logical direction for how the media consumer, hereafter referred to as the user, interacts with, progresses, observes and consumes the media presented.

Browsing semantics refer to the behavior and intended behavior designed by the programmer into the platform; the engine is the "brain" that determines the semantics. These semantics can be as simple as a "next" button at the bottom of a chunk of text that users must click on to view the next piece of text, along with a "previous" button that allows the user to view the previous text. However, these semantics can be very complex. One such example comes from the paper "Card Shark and Thespis: Exotic Tools for Hypertext Narrative" by Mark Bernstein. This paper describes a theoretical engine, Thespis, involves a giant state machine keeping track of multiple characters within a narrative and their states, which in turn determines the behavior of those characters ("Card Shark and Thespis" 44). This idea illustrates the vast space of possible semantics that drives much research in the field.

The engine used in a media presentation system is obviously one of the biggest factors in the design of the system, and fundamentally dictates not only how the media is delivered to the user, but sometimes the form of the media itself (or, conversely, the form of the media can dictate the form of the engine). Innovation in the design of browsing semantics engines can therefore drive innovation in media presentation and consumption and reveal new, and possibly more effective, methods of communication and storytelling.

Orchestrate

The Orchestrate platform is software developed to enable easy substitution of browsing semantics engines and media. This ease of substitution translates directly into ease and speed of modelling, developing and testing new engines.

This ease of substitution arises from the separation of the browsing semantics engine from the media and the Orchestrate system, making the media display system modular in nature. Each aspect of the system, the media, engine(s), and/or controller(s), can be worked on independently, enabling not only ease of development for each individual part but also the ability to "mix-and-match" various pieces of systems to create and test new ones.

This ease of substitution will likely prove hugely beneficial to research on browsing semantics engines going forward. The design, implementation and testing of the Orchestrate system is detailed later in this paper.

Experimental Engines

In addition to Orchestrate, several experimental engines and features were developed by other researchers under Dr. Furuta. These include mood lighting and Bluetooth functionality, as well as an engine to analyze emotionally-connotated words in a piece of text.

The fact that these engines were developed simply by interested student researches further illustrates the need of an easy-to-use system to test them, such as Orchestrate.

Literature Review

Several pieces of peer-reviewed research literature, primarily describing possible designs for browsing semantics engines, were reviewed over the course of the project.

Such papers include the aforementioned "Card Shark and Thespis" by Bernstein, along with some of his papers over the Storyspace system; "Browsing Intricately Interconnected Paths"

by Pratik Dave et al; and "Display-agnostic Hypermedia" by Unmil P. Karadkar et al. Several of these papers describe theoretical or experimental browsing semantics engines, or, in the case of "Paths", ideas for the traversal of graph-based implementations thereof ("Card Shark and Thespis" 41, Dave 95, Karadkar 58).

The literature review provided valuable insight into the requirements of Orchestrate in supporting the engines, as well as stoking interest in future possible projects using the engine and possible test cases for Orchestrate itself.

Preview

The remainder of this paper will discuss the design of the Orchestrate system, results of testing and usage, and the conclusions regarding its use and the further direction of Orchestrate-related projects. Chapter 2 will discuss the design and building process of the current version of Orchestrate. Chapter 3 will discuss the testing and analysis carried out with the system. Chapter 4 will present the overall conclusions regarding the final version of the platform.

CHAPTER II

DESIGN

The Orchestrate platform was designed to allow the easy substitution of browsing semantics engines and other functional pieces of an electronic media system. Additional concerns included ease of use, generalization of systems and quality of code and operation.

Goals

Easy substitution of logical engines, as the primary motivation for this iteration of Orchestrate, was kept in mind during the designing of the system. Goals identified for the functionality of the engines with regard to Orchestrate integration included ease of switching between logical engines, consistent compatibility with the rest of the Orchestrate system, and a generalized communication structure to reduce work having to go into other segments of the system during an engine change.

Orchestrate Version 1

Orchestrate 1.0

At the beginning of the design process of the current version of Orchestrate, a previous version, hereafter referred to as Orchestrate 1.0, which supported very limited functionality, was already in existence. This version consisted of a directory of Python files that could be run locally on a device in order to interact with each other. The structure of this version consisted of a server that provided service to two types of clients: a “client” process which displayed the media, of which there could be multiple instances; and a “server control” process, of which there could only be one and with which a user could control the actions of the media-displaying client(s). The operation of the code was not very generalized; the server control had to be

initialized before the client(s), and, as mentioned, only one server control could exist, liming the number of users who could control the media to one. This version also supported timing data being sent between the various processes.

Orchestrate 1.5

It was decided to use this previous version of Orchestrate, Orchestrate 1.0, as a starting point, and to integrate all of the new functionality into it. The first thing that was decided was that the scheme of the code was to be generalized to not only allow for multiple server controls to exist, thereby allowing multiple users to control the media being displayed, but also for the sequence of process initialization to be more generalized. This would allow clients to be created before server controls, and multiple instances of either process to be created in either order.

These changes were made to the original version, keeping other functionality, such as timing, intact. In addition, functionality was added to the server controller to allows individual controllers to terminate without terminating the entire system. A debugging period followed these changes and the resulting version, referred to hereafter as Orchestrate 1.5, were saved on online version control platform GitHub.

Orchestrate Version 2

Motivation

The next step of the project was to introduce engines into the Orchestrate platform. Up until this point, all of the logic of the media was stored in the client process. This meant that it was impossible to separate the logic from the media display, so a completely new client file had to be written, and completely new media output code had to be generated, for each new logical idea that was developed. In addition, there was no way for client processes to communicate with

each other in Orchestrate 1.0 or 1.5, so logical systems couldn't pass information to other processes.

New Features

The basic design that was arrived at was to introduce an “Engine” process, of which there could only be one, into the architecture of Orchestrate. The engine represents the logical brain of the piece of media being displayed, and the integration of this aspect of the Orchestrate system and its essential modularity would be the key piece that would enable the testing of new and different logical engines in the future.

The Engine would take information from the Server, including user input commands from Server Control instances, and generate a state variable, sending that data to the Server. The Server would then send that variable to the Client, thereby dictating the behavior of the Client and the way in which the media would be displayed. The Server would then receive data from the Client and pass it to the Engine. The Engine would receive the data being displayed by the Client, essentially letting it “see” what the Client was doing, as well as further commands from the Server Controller passed through the Server to the Engine. It would use this data to calculate further appropriate states.

Aside from this new Engine process being added to the Orchestrate architecture, the Client would have to be reworked to accommodate the changes in the system. It would have to be responsive to the state value being passed into it from the server, and it would have to send its data to the server as it was printed. Therefore, a new Client file was created for the purposes of testing the new Engine architecture.

The graphic used to visualize the initial concept for this new version of Orchestrate, hereafter referred to as Orchestrate 2.0, is shown in Figure 1. As development of the code continued, this early concept was open to change.

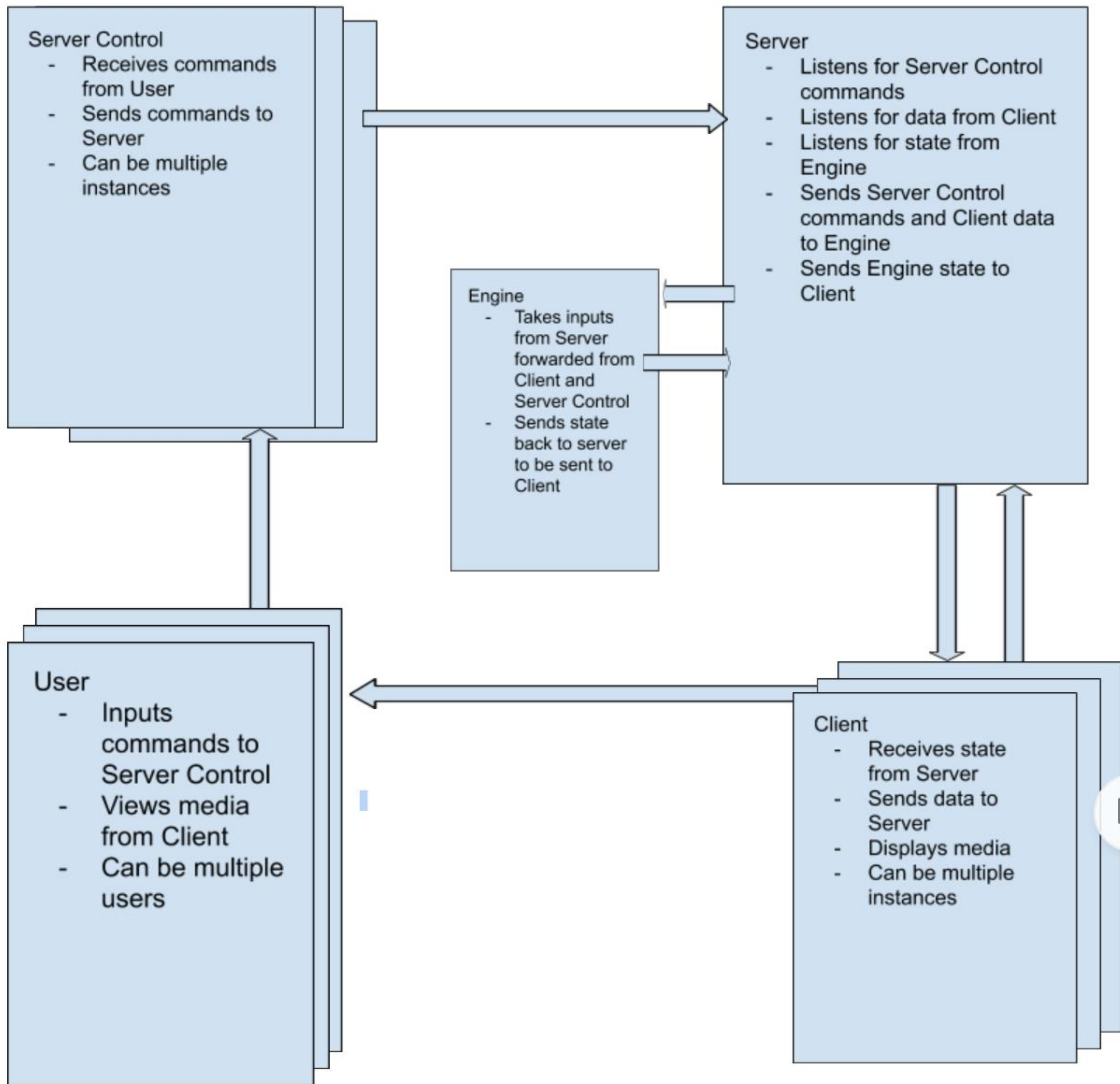

Figure 1. Initial design diagram for Orchestrate 2.0.

Implementation

As these developments represented a significant change to the Orchestrate platform, it was decided that simplicity needed to be maintained for this first integration of the Engine functionality. The timing functionality from Orchestrate 1.5, which allowed the Server to generate timing data and share it with other processes, could have caused a lot of confusion with the interprocess communication needed for the Engine functionality. Therefore, the timing algorithms were removed (but still stored in Orchestrate 1.5 via version control systems) with the intent of being added back into the system at a later point.

Once this functionality was removed, a prototype of the system was implemented. The Engine was built by essentially copying the already-existing client code. This was done because the Engine has very similar methods of interaction with the client – the same connection scheme and very similar instances of communication with the server.

In the initial prototype version of Orchestrate 2.0, an input from the server controller was checked to see if it matched one of a predefined set of inputs allowed by Orchestrate; based on the command input, a set of global server Boolean values was set; and then a separate thread within the server sent those Boolean values to the engine for it to decode and use as it would, according to the logic of the engine. At the same time, the client would continuously send the last thing it printed to the server, and that would also be sent to the engine for the purpose of calculating the state. The engine would calculate the state based on this data, and send it back to the server to be passed on to the client. The client, in turn, would print media based on whatever state value it received from the server.

After experiencing many issues regarding the way in which commands from the server controller were handled, the mechanism for passing the command to the engine was reworked.

Instead of using an input command to set the values of a set of global Boolean values, the Server instead simply stores the input command in a global string variable, which is in turn concatenated with the Client data and passed as one string to the Engine. This solved many issues with concurrency, interprocess communication barriers and race conditions, as well as simplifying the code and being more efficient on runtime. After more debugging, Orchestrate 2.0 was found to be operational. This first fully operational build of Orchestrate 2.0 is hereafter known as Orchestrate 2.0.1. The final architectural design for Orchestrate 2.0.1 is illustrated in Figure 2.

After this working prototype of Orchestrate 2.0 was completed, several example engines and an example client were created in order to test the system. The tests are described in more detail in subsequent sections of this paper; in short, the engines caused the client to behave in different ways depending on which engine was being used with the Orchestrate system at any given time. These successful tests proved three things, all of which were critical to the goals of Orchestrate. First, they proved that the platform functioned as intended; second, that the engines were easy to substitute with this version of Orchestrate; and third, that the engines successfully caused different and correct behavior when paired with the same client.

Improvements to Orchestrate 2.0

While Orchestrate 2.0 was fully-functional, several improvements and consolidations still needed to be made with regards to the project. One important objective involved assuring that the Orchestrate system still started and exited appropriately. During testing of the initial functional version of Orchestrate 2.0.1, user-prompted termination of individual Server Control instances, or even the entire Orchestrate session, was barely functional. In addition, Orchestrate 2.0 still

needed to be consolidated with Orchestrate 1.5 so as to have the timing functionality in the final design.

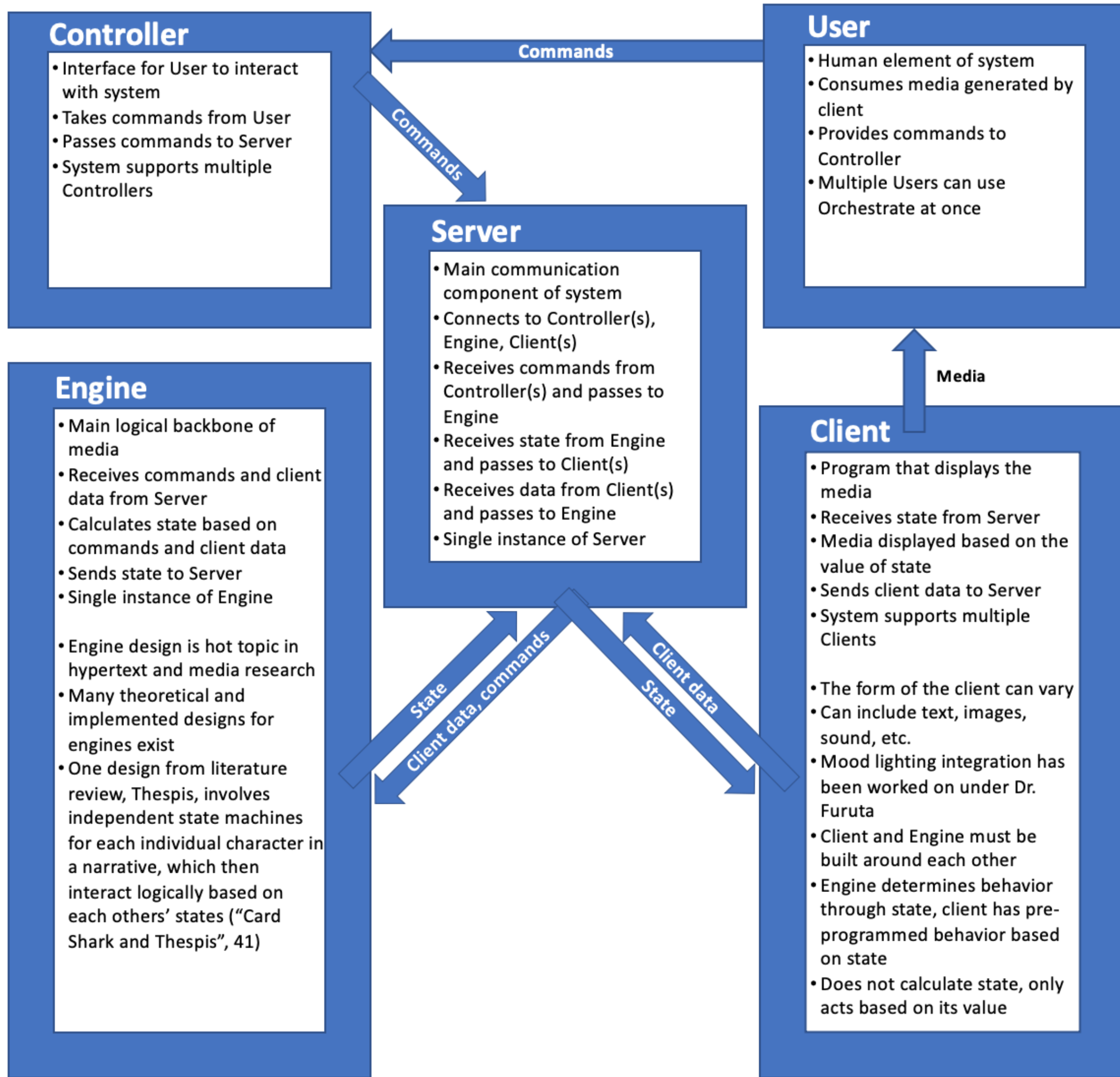

Figure 2. Final design for Orchestrate 2.0.1.

In addition, more generalization work needed to be performed regarding how Orchestrate 2.0 interacted with users, and how information was passed between processes. In Orchestrate 2.0.1, the engine passed a single value, state, to the server, which was in turn passed to the client.

This needed to be generalized to a 2-value system of state and count instead of just state, which would greatly increase the feasibility of the system for use with any large piece of text-based media.

User input generalization

Perhaps the most obvious improvement, at first, was that the system of communication between the Server Control and the server and engine in Orchestrate 2.0.1 badly needed to be generalized. As clearly shown in Figure 3, which showcases the overly-restrictive input-checking code of Orchestrate 2.0.1, the original Server input checked for a series of specific hardcoded input strings; any other input would be deemed invalid. As the logical center of the media systems is supposed to be located in the Engine, these input checks belonged in that part of the Orchestrate system. This would also greatly increase the modularity and flexibility of Orchestrate, which were both key overall goals of the development process, as Engine designers would have full control over what inputs were accepted, rather than relying upon rigid, hardcoded values in the Server.

Part of the reason for this initial rigid design is that Orchestrate 1.0 and 1.5 used these commands in its timing algorithms, and this setup hadn't been fixed in the transition to Orchestrate 2.0. However, if this input-checking code was kept in the Server, all media systems would have to be able to operate with only these hardcoded values. It was therefore decided to move the input-checking code from the Server to the Engine, and to eventually re-integrate timing functionality back into Orchestrate 2.0 using a different, as-of-yet undetermined algorithm.

The Server Control input checking was therefore changed to a system where the server only checked for two hardcoded values, End, signifying the end of the entire Orchestrate session,

```python
print("From Server Control:")
print (data)
if data == 'END':   #Exits all server controls and ends the server
    serverEnd = True
    controlData = "End "
    print("Setting command to End")
elif data == 'Exit':  #Exits just this server control
    print("Server control at address ", add, "on port", portNum, "self-terminated")
    break

elif data == 'PAUSE':
    controlData = "Pause "
    print("Setting command to Pause")
elif data == 'START':
    controlData = "Restart "
    print("Setting command to Restart")
elif data == 'RESTART':
    controlData = "Restart "
    print("Setting command to Restart")
elif data == 'RESUME':
    controlData = "Resume "
    print("Setting command to Resume")
elif data == 'BACK':
    controlData = "Back "
    print("Setting command to Back")
elif data == 'FORWARD':
    controlData = "Forward "
    print("Setting command to Forward")

else:
    print("Invalid command; try again")
    continue
```

Figure 3. Original server code for processing server control commands.

and Exit, signaling the client to close the link to one specific iteration of the server control. Any other command was sent directly to the engine to be dealt with. This allows any command to be integrated into the system based on the design of the engine, furthering the goals of decentralizing the design of the media and placing more control into the design of the engine. Only the Exit and End commands would thereafter need to be held standard for all Orchestrate-compatible systems. The version of Orchestrate encompassing these and other changes to Orchestrate 2.0.1 is known as Orchestrate 2.1.

Engine and Client Communication

The transition from a purely state-based model of engine outputs to a state-client model was relatively simple, and came with several added benefits to the system. In Orchestrate 2.0.1, the server took in the state as an int from the engine and passed it to the clients; this limited the method of communication to integer-type data. In Orchestrate 2.1, this was changed to a system where the server receives a string from the engine and passes it to the clients. This allows for complete flexibility on the part of the engine/client designers. Because most things, including integers, can be encoded as strings, virtually anything can be passed from the engine to the clients under the new approach. This means that the designers of engines and their respective clients can set up a system where anything can be passed through the server, so long as it is encoded as a string during transit.

Specifically for the purposes of short-term testing, the experimental engines and clients use this string passing to carry two integers, separated by a space and encoded as a string, through the server. The first of these integers represents the state, 1 for an active, or printing, state and 0 for a paused state; the second represents the count, and determines which content the client will actually print. The change from state to state-client resulted in much simpler logic in the engines and the clients, and much more comprehensible design for the testing of Orchestrate. In version 2.0.1, at least two states had to be designated for each individual line of content, resulting in a giant list of specific states that scaled very poorly with the number of lines added to the displayed media. In 2.1, only one count value must be generated along with each line of content, and the program could easily be configured to do away with even this 1-to-1 requirement. Therefore, the clients and engines are much more concise and easy to understand.

Server Control Exiting and Session Termination

During the design of Orchestrate 2.0, the Engine aspect of the design was added. The prior version, Orchestrate 1.5, had functional systems for cleanly exiting individual Server Control instances, or terminating an entire Orchestrate session; however, as the focus for the Orchestrate 2.0.1 build was achieving functionality of the Engine and Client communication, that functionality hadn't been carried over to the new build. In fact, during many Orchestrate 2.0.1 tests, a termination interrupt signal (ctrl + C) was used in the shell running the various Orchestrate processes whenever it became necessary to terminate a session. Clean, user-prompted termination functionality was re-added during the translation from Orchestrate 2.0.1 to Orchestrate 2.1. In fact, a new feature, dictating that the Orchestrate session itself will terminate if no Server Control instances are currently active, was added to the system in the 2.1 version, as well.

One tool used to accomplish this was the design of a user-defined exception. This would be raised when a user on any Server Control input the "End" command or the "Exit" command if there was only one Server Control instance, thus signaling the end of the current Orchestrate session. When raised, this exception interrupts other processes and begins shutting them down, enabling a very clean and responsive end to the Orchestrate session.

Timing

The other major piece of functionality that was left out during the development from Orchestrate 1.5 to Orchestrate 2.0.1 was timing functionality. Originally, the server kept track of timing data, and communicated this data to the Server Controls and the Clients. This data was also connected to the Forward, Back, Pause and Restart commands that were originally hardcoded into the system, as shown in Figure 3.

As the major focus of Orchestrate 2.0.1 development was enabling the engine and client interplay, this functionality was left off for simplicity's sake. However, one goal in the development of Orchestrate 2.1 was to bring back all functionality from Orchestrate 1.5, including this timing functionality. This timing functionality, being the least important part of the system as it pertained to a functional user-engine-client interaction pattern, was the last piece of functionality to be added to the final code.

In Orchestrate 1.0 and 1.5, timing was tied to the server command inputs. The inputs Forward, Back, Pause, Restart, and Resume directly incremented, decremented or paused the timer; therefore, it was less a scientific timer than it was an instruction count. When these inputs were repurposed for media control during the translation to Orchestrate 2.0.1, and completely removed from the server program in Orchestrate 2.1, their functionality for the timer also ceased. A completely new scheme for timing operation would need to be found for the timing functionality to be added into Orchestrate 2.1.

It was decided to implement a scientific timer using Python's time library. This timer would function independently of any server commands, and would simply keep time from the start of the Orchestrate session. Since the different programs that make up Orchestrate can be started independently, the timer was set to start at the first point where the Server, the Engine, at least one Client and at least one Server were all active, as these four components are necessary for the operation of Orchestrate.

A thread, time_sender, is created with each Server Controller instance; these treads send the global timer variable to their respective Server Controllers. This implementation allows both the Server and Server Controller processes have time data.

One added benefit of rewriting the timing infrastructure is that it allowed a sizeable piece of code from Orchestrate 1.0 and 1.5 to be deleted. This piece of code, the thread time_counter, was located in the Server.py file. Deleting this not only deleted the number of threads the Server process operates with, thereby making runtime much smoother, but also made the code itself less complicated and easier to read.

After the re-implementation of timing into the Orchestrate system, the design of Orchestrate 2.1 was complete.

Bugfixes

During all stages of development, bugs were present in many parts of the Orchestrate system. The solutions to many of these bugs turned out to be great improvements to the Orchestrate system, and in some cases actually simplified the code itself. Other bugfixes required more convoluted workarounds and fixes that made the code more complex.

Final State

Orchestrate 2.1 represents the current version of Orchestrate, and the final state of the platform for this iteration of the project. Testing took place throughout the design process, and the design was tailored around the results of this testing and around desired behavior and features. Not only was testing carried out after each individual feature was added and each major version of Orchestrate was completed, but additional desired features were determined based on behavior in some of these tests. In addition, features added during the design process came at the suggestion of other undergraduate researchers and from Dr. Furuta himself, as well as an intuitive understanding of the goals of Orchestrate.

Orchestrate 2.1 is certainly not a perfect piece of code, and it will doubtless be improved upon in years and studies to come. However, it is a functional system that allows media systems

to be built and tested around pre-existing software and code templates. Therefore, the goals for the design of the system have been met.

CHAPTER III

TESTING

Completed Testing

Testing of Orchestrate with regards to its goals of modularity and Engine-Client-Controller communication began in earnest after Orchestrate 2.0 was first completed. To this end, different media clients and engines were used to test the functionality of Orchestrate and see if it had met the design criteria.

The first test of Orchestrate 2.0 was a relatively simple proof-of-concept type verification of base functionality. When Orchestrate 2.0 was completed, three different logical engines were created, hereafter referred to as Engine 1, Engine 2 and Engine 3, as well as one media client. The media in the client consisted of a simple 4-line conversation, with each line consisting of three words. The conversation would be printed by the client with behavior dictated by the Engine being used.

All three engines incorporated an initial paused state into the behavior of the client. The media would start being displayed upon a "Start" command from the user through an instance of the Server Controller. Engine 1 caused the conversation to print one line at a time, pausing in between lines and waiting on a "Forward" input from the user through the Server Controller. Engine 2 caused the same behavior, except that instead of pausing in between lines, the line would repeat itself until a "Forward" command was entered. Engine 3 simply caused all of the lines of the conversation to run after each other, with no pause or repetition state in between. Each engine also supported commands such as "Back" to return to a previous line; "Pause" to

pause the media; “Resume” to exit a paused state; and “Restart” to start the media over again from the first line.

Testing of these engines revealed the behavior to be exactly as expected. Moreover, the engines themselves were easy to switch out; the user merely had to run a different engine file and it was seamlessly incorporated with the Orchestrate system. This validated the development work that went into Orchestrate 2.0, as the goals of modularity and communication were evidently working.

As Orchestrate 2.0.1 was updated to Orchestrate 2.1, so were the Engines and Clients associated with these initial tests. As expected, the engines and clients continued to function as intended after the updates were made. In addition, as explained previously, these Engines and Clients benefitted from the simplicity of the state-client communication model, and the associated Server Controller instances now also displayed the accurate server time. Given the accurate behavior on the part of the Engines, Orchestrate 2.1, the final version of Orchestrate, showed the desired functionality and interchangeability of engines that motivated the entire project.

Figure 4 shows an Orchestrate 2.1 session using these testing engines and clients, with all four parts run locally on the same machine. The bottom right window shows some of the outputs of the Server process, illustrating data being passed back and forth between the various processes. The top right window shows the Client, and the short conversation can clearly be seen printed out following some “Invalid length” statements that indicate that the media hasn’t been started yet, and thus doesn’t have correct input data from the Server. The top left shows the Server Control, where the user input commands that were then sent to the Server and eventually

the Client. The bottom left shows the outputs of the Engine process, and illustrates how the Engine calculated its data based on inputs from the Server.

```
initial-clients-code — -bash — 80×24
Tates-MacBook-Pro-2:initial-clients-code tatebanks$ python3 serverController.py
/Library/Frameworks/Python.framework/Versions/3.7/Resources/Python.app/Contents/
MacOS/Python: can't open file 'serverController.py': [Errno 2] No such file or d
irectory
Tates-MacBook-Pro-2:initial-clients-code tatebanks$ python3 serverControl.py
Server Control. sends commands to the Server
Commands: "Start", "Restart", "Resume", "Pause", "Forward", "Back", "Exit", and
"End"
[server time: 0] [# of clients: 0] Enter Server command:
Start
Sending command: Start to Server
[server time: 6.162194013595581] [# of clients: 1] Enter Server command:
End
Sending command: End to Server
[server time: 18.283445835113525] [# of clients: 1] Enter Server command:
Server terminated
Server Control Ended
Tates-MacBook-Pro-2:initial-clients-code tatebanks$
```

```
initial-clients-code — -bash — 80×22
Tates-MacBook-Pro-2:initial-clients-code tatebanks$ python3 client1.py
Connecting to Server at port 22001 127.0.0.1
Invalid length
Invalid length
Invalid length
Invalid length
Invalid length
Hello,
How're
You?
Fine,
Thanks!
Yourself?
Very
Well,
Thanks.
So
Long,
Then.
Media finished
```

```
initial-clients-code — -bash — 80×24
['Default', 'default123']
['Start', 'default123']
New server info: 1 1
['Default', 'default123']
['Default', 'default123default123default123']
['Default', 'Hello,']
['Default', "How're"]
['Default', 'You?']
New server info: 1 2
['Default', 'Fine,']
['Default', 'Thanks!']
['Default', 'Yourself?']
New server info: 1 3
['Default', 'Very']
['Default', 'Well,']
['Default', 'Thanks.']
New server info: 1 4
['Default', 'So']
['Default', 'Long,']
['Default', 'Then.']
New server info: 1 5
['Default', 'default123']
['End', 'default123']
Engine Ended
```

```
server-code — -bash — 80×24
Sending -> Default Well, <- to engine
Received 1 3 from engine
Received Thanks. from client
Sending -> Default Thanks. <- to engine
Received 1 4 from engine
Sending -> 1 4 <- to client
Time: 15.231199979782104
Received So from client
Sending -> Default So <- to engine
Received 1 4 from engine
Received Long, from client
Sending -> Default Long, <- to engine
Received 1 4 from engine
Received Then. from client
Sending -> Default Then. <- to engine
Received 1 5 from engine
Sending -> 1 5 <- to client
Time: 18.270904064178467
Received default123 from client
Sending -> Default default123 <- to engine
Received 1 5 from engine
From Server Control:
End
Command: End
```

Figure 4. An Orchestrate instance run locally on four Terminal windows on the same MacBook Pro laptop.

Theoretical Applications

However, once this stage was reached, the COVID-19 epidemic started ramping up in intensity in the United States. This interrupted plans to work with other individuals who had previously had association with the project and test some of their code with Orchestrate, in addition to doing focus testing. Communication between parties seemed to break down somewhat after the pandemic hit, and some other undergraduate students who may have been more willing to help during normal times seemed to not be as interested given the unusual virus situation.

Given this, much of the potential of Orchestrate must be analyzed by the theoretical, rather than by looking at the rigorously-tested. Thankfully, weekly meetings with Dr. Furuta and significant literature review can provide valuable insights into cases where the Orchestrate platform can help advance development of future logical engines and electronic media presentation systems.

Ideas from Work with Dr. Furuta

Dr. Furuta, the primary user of Orchestrate in the near future, proposed several different possible systems that he believes Orchestrate could be used to model. Analyzing these ideas and considering how they would be implemented with the current version of Orchestrate can provide justification for the design choices made in the development of the system, and provides good thought experiments for proving the functionality of Orchestrate. These various ideas model a diverse set of media presentation systems and illustrate the flexibility the platform will contribute to Dr. Furuta's research.

Dr. Furuta proposed the idea of a client with multiple videos, which would have the ability to play independently; the associated engine would control the order in which the videos play, and would even be able to stop videos prematurely and start others, or run them simultaneously. The engine could decide when to start or stop the videos, which would play in the client, based on Orchestrate's timing functionality, data from the client, or user input. This would allow easy control of the display of multiple videos of many different formats, and could be used with multiple Client instances, as well.

Another idea Dr. Furuta thought was interesting was the easy design and implementation of card games possible using Orchestrate. Given one client with the ability to display the cards of a standard 52-card deck, various engines could be created to model various card games using the

same client. This would allow the designers of these card game systems to work only on the Engine portion of the card game; they wouldn't have to work on the Server Control or Client parts, as the Server Control is generalized and the single card-displaying Client would be sufficient to display most card games. This would therefore help facilitate and expedite the modeling and implementation of various card games and the creation of new ones, which is exactly the sort of thing Orchestrate was designed to do.

One media system that was worked on, but that experienced complications due to coronavirus, would have been an emotion-analyzing engine. Various pieces of code were worked on by other undergraduate researchers working with Dr. Furuta; however, these other researchers didn't continue to work under Dr. Furuta for more than one semester, and due to COVID-19, it was hard to collaborate with them once Orchestrate 2.1 was finished. These pieces of code could, if modified to fit the specifications of an Orchestrate Engine, analyze the emotional content of the words in a piece of written media, and change mood lighting or other cosmetic effects to emphasize the emotional states derived from the words. This interesting case shows another possible browsing semantics engine that can definitely be modeled using Orchestrate.

Ideas from Literature

The literature review associated with this iteration of the Orchestrate project consisted mainly of papers describing theoretical logical engines. These papers represent the exact type of situation that Orchestrate was designed to assist with, and therefore, analyzing the feasibility of some of these engines' usage with Orchestrate is a good way to determine the functionality of the system.

One very interesting example from the literature reviewed involves a giant state machine keeping track of multiple virtual characters and their states. These states determine the behavior

of the associated characters. This idea is a theoretical engine known as Thespis, and comes from Bernstein's "Card Shark and Thespis: Exotic Tools for Hypertext Narrative."

This Thespis idea could be implemented using Orchestrate in several ways. It could consist mainly of one huge engine, with the clients serving simply as displays for the characters' actions. In this idea, the Engine itself would take a huge amount of work, but the associated Orchestrate Clients would be very simple. Orchestrate could also easily be modified to support more decentralized Client programs; in this case, all of the characters' potential actions could be stored in individual Clients, and the Engine would simply keep track of the characters' states, with each Client instance actually deciding the actions of its associated character based on the states from the Engine.

In addition, the system could be set up such that a user could elect to "play" any of the characters, and that character's actions would be decided by user input rather than Engine computation. This wouldn't require any modification of the Orchestrate platform, but rather would have to be designed into the Engine program by its designer. Either way, this seems to be a system that Orchestrate could definitely support.

These are only a few examples of theoretical media systems that Orchestrate could support. The system was designed with versatility in mind, so the system can support an infinite number of media systems, designs, and architectures, ideally with little to no modification to the Orchestrate server itself.

Storyspace Comparison

Two pieces of literature reviewed for the project were "Storyspace 1" and "Storyspace 3" by Mark Bernstein. These papers discuss the usage and development of the platform Storyspace, which is used to write hypertext literature ("Storyspace 1" 173, "Storyspace 3" 41). Pieces of

hypertext literature are comprised of written media with embedded links, which can link to graphics, additional information, or even other parts of the literature. There is a huge variety of possible architectures that a piece of hypertext literature can take, and this variety has lent itself to both academic research and artistic endeavors.

Storyspace is a leading platform for the development of these pieces of hypertext literature, and is a widely-used tool to help authors create these hypertexts (“Storyspace 1” 173). According to Bernstein, “It has been widely used for teaching hypertext writing, for crafting hypertexts, and for studying published hypertexts; at times, Storyspace has seemed almost synonymous with literary hypertext.” (“Storyspace 1” 172)

Orchestrate could function in a similar role for semantic engine development as Storyspace fills for hypertext literature. They are both tools that enable easy development of media systems, and both developed primarily for non-commercial use. Perhaps, as Orchestrate gains more and more use, the platform can be expanded and refined, eventually leading to mass use in the vein of Storyspace.

The Future of Orchestrate

Orchestrate will surely be a helpful tool for Dr. Furuta’s future research, and help him and his researchers develop logical engines and systems of electronic media presentation. Several engines developed by his researchers, including the aforementioned engine which analyzes the emotional content of a text, would have been able to be rigorously tested and used had Orchestrate 2.1 already been completed while the researcher who designed it was still with Dr. Furuta.

Just as Orchestrate 2.1 and the current functionality associated therewith was build from an existing system, further iterations of Orchestrate could be developed in the future. These

iterations could improve on the functionality currently present, or could introduce even more new features and functionality into the system. Orchestrate exists to serve the needs of those developing systems of media presentations, and should more needs arise that Orchestrate 2.1 doesn't completely cover, Orchestrate could absolutely be further evolved to enable that further functionality.

In addition, Orchestrate could develop in a way that enables it to integrate with other systems and display methods. Dr. Furuta himself is no stranger to this sort of adaptation. In a paper by Dr. Furuta, Unmil Karadkar, and others, entitled "Display-agnostic Hypermedia," researchers present ways to adapt the same piece of hypermedia to different display formats without having to actually modify the hypermedia itself (Karadkar 58). This technology, and the developments on the concept since, could be used with the display of Orchestrate to ensure its compatibility with many other systems and display parameters, further increasing the platform's usefulness.

CHAPTER IV

CONCLUSION

The final version of Orchestrate consists of a system of files, or a format for writing new files, that can be run together to create modular systems of electronic media. A working system run on Orchestrate has four parts: The Server, which ideally stays constant for all systems; The Engine, which contains the logic of the media system; Server Controllers, which allow users to input commands and data into the system; and Clients, which display the media to the user. This system should allow developers of electronic media systems to have an easy-to-use platform to develop and test their systems, and will likely be very helpful for Dr. Furuta and his future research.

The final version of the Orchestrate platform fulfills all of the desired functions set out in the design of platform's latest iteration; it successfully facilitates communication between users, Clients and the Engine while maintaining the modularity key to the easy substitution of engines. This functionality should be greatly helpful in the development of future engines and electronic media systems.

Initial testing of the Orchestrate system yielded encouraging results. It showed that appropriately-designed media systems work well with the Orchestrate system, and that these media systems are not hard to design around Orchestrate's requirements. Testing also confirmed the modularity of the system, and demonstrated how easy it is to change the different parts of a media system using Orchestrate.

Even more excitingly, many untested theoretical engines are, at least intuitively, supported by the design of the Orchestrate system's infrastructure. Many of these are elaborated

upon in the literature reviewed for this project, and upon an evaluation of how these various theoretical engines would need to work, Orchestrate seems to be able to support many of them. This suggests that Orchestrate has the potential to be a powerful tool in the development and testing of new engines and media systems.

REFERENCES

Stotts, David P. and Richard Furuta. “Petri-Net-Based Hypertext: Document Structure with Browsing Semantics.” ACM Transactions on Information Systems, Vol. 7, No. 1, pp. 3–29, January 1989. Print.

Karadkar, Unmil, Richard Furuta, Selen Ustun, YoungJoo Park, Jin-Cheon Na, Vivek Gupta, Tolga Ciftci, and Yungah Park. “Display-agnostic Hypermedia.” Hypertext 2004: Proceedings of the Fifteenth ACM Conference on Hypertext and Hypermedia, August 2004. pp. 58–67. Print.

Bernstein, Mark. “Card Shark and Thespis: Exotic Tools for Hypertext Narrative.” Hypertext ’01: Proceedings of the 12th ACM conference on Hypertext and Hypermedia, September 2001, pp. 41-50. Print.

Dave, Pratik, Unmil P.Karadkar, ,Luis Francisco-Revilla, Frank Shipman, Suvendu Dash, and Zubin Dalal. “Browsing Intricately Interconnected Paths.” Hypertext ‘03: The Fourteenth ACM Conference on Hypertext and Hypermedia, ACM Press, 2003. pp. 95–103. Print.

Bernstein, Mark. “Storyspace 1.” Hypertext ’02: Proceedings of the thirteenth ACM conference on Hypertext and Hypermedia. June 2002, pp. 172-181. Print.

Bernstein, Mark. “Storyspace 3.” HT '16: Proceedings of the 27th ACM Conference on Hypertext and Social Media, July 2016, pp. 201-206. Print.