Final Major Project - Internet Engineering (H622)

Developing a device-agnostic, real-time ARS/PRS system with support for broadcasting rich interactive content

Final Report

Author: Thomas Usher *Supervisor:* Dave Price

Date: —- *Version:* 0.1

1 Acknowledgements

With thanks to Dave Price, for feedback, advice and suggestions to myself and the rest of the project group during the development of this project.

Many thanks to Denise Williams-Morris and Alex Kruzewski, for testing, feedback and moral support.

2 Abstract

Some abstract

Contents

1 Acknowledgements **2**

2 Abstract **3**

3 Context **3**
3.1 Introduction . . . 3
3.2 Objectives . . . 4

4 Relevant Literature and Existing Systems **5**

5 Methods and Planning **5**
5.1 Process Model . . . 5
5.2 Planning . . . 5
5.3 Implementation Tools . . . 5

6 Design and Implementation **6**
6.1 Overview . . . 6
6.2 Presenter System . . . 7
6.3 Client System . . . 8
6.4 Real-Time Communication . . . 9
6.5 Pushing Content to the Client . . . 10

7 Problems **11**

8 Testing Strategy **11**

9 Critical Evaluation **11**

3 Context

3.1 Introduction

Audience Response Systems (ARS), more commonly referred to as 'clickers' have been used with varying degrees of success in educational establishments around the world for numerous years. Their use as a means to improve interactivity between students and instructors in an educational environment has been widely documented, with research indicating notable effects on student responsiveness, involvement and success. While this project does not involve additional research in to the value of PRS systems, it does attempt to highlight and enhance upon "The give-and-take atmosphere encouraged by use of clickers which... makes the students more responsive in general"[17].

From using these clickers from a students' perspective, I have noted two primary issues with the current implementation of these ARS systems:

- The devices are limited in what can be displayed on an individual users' device - while the majority have no displays at all, those that do typically only show which number/letter the user has selected. This limitation restricts interaction to simple question and answer style communication, as well as limiting potential for individual or group specific broadcasts.
- Specific clicker hardware is required. While they can be relatively cheap, costs do increase with complexity and they are single-purposes devices. In most cases, specialist receiver hardware is also required and in larger scale cases, technicians are required to maintain these systems.

This project attempts to solve the above issues by creating a 'next-generation' ARS system for use on any system with a relatively modern web-browser and the ability to access the local network. While the system is platform and device agnostic, the increasing popularity of Apple's iPad and other tablet computers to enhance textbooks in education provides an ideal device around which the system was built and tested within the time constraints (although it does not use any tablet or device specific implementations).

The system is built around an extensible framework; allowing additional types of content to be created for broadcast, and to allow the system to be used in other use cases, such as school classrooms and business meetings.

3.2 Objectives

Clickers exist to improve interactivity during lectures, helping to engage students in the subject, improve attention-span and providing additional ways for students to feel personally involved in the subject. Secondarily, facilitating real-time feedback allows lecturers to tailor their approach based on the class response - for example, the lecturer might broadcast a set of questions about the current topic, if a large percentage of students answer incorrectly, the lecturer might wish to attempt another explanation. This project attempts to maintain all of these benefits while building on the types of content which can be broadcast, as well as adding additional forms of interactivity where possible.

The system was designed to:

- Be very simple to use for both the presenter and attendee.
- Be configureable once per site, rather than requiring setup for every meeting/lecture.
- Usable on a wide array of mobile and personal computing systems, including mobiles, tablets and laptops.
- Facilitate near-instant, actionable feedback to the presenter.
- Support a variation of 'types' of material, with the facility to extend and add more if required.

4 Relevant Literature and Existing Systems

5 Methods and Planning

5.1 Process Model

For this project, I wanted to enjoy the development of the system as much as possible, so adopting a process model that best suited the way I write software was important to me. As my typical development methodology is 'hack at it until it works', I wanted to find a process that injected a degree of organisation in to the process, giving me a way to plan and view my progress while maintaining focus on the code rather than the documentation.

For this reason, I chose to use an iterative development approach, as I felt it best suits how I build software, allowing me blocks of time in which I can focus on writing code, while adding a sense of organisation and planning to the process. While the waterfall model might also have been suitable, I considered the requirement to complete the design before moving on to development too restrictive as I believe any software project needs room to evolve during its implementation.

5.2 Planning

A discussion on how I came to a decision on the final requirements/objectives.

5.3 Implementation Tools

The system was implemented in three parts, each communicating over network services. The presenter system was primarily developed in Python[6], using the Django[5] web framework, while the client system was written in JavaScript and served from the third component, the distribution server, written in JavaScript using the Node.js[8] framework.

Both the client system and the presenter system used a frontend written in HTML5, CSS3 and JavaScript. All JavaScript frontend features were developed using the jQuery[9] library.

Redis[12], a key-value store, was used to great effect in the project - its primary purpose was to store a cached representation of the content to be broadcast, allowing fast retrieval when required. Its built-in publish/subscribe server mechanism was also used as the communication protocol between the presenter system and the broadcast system. It was also used for caching user session data and data mappings which required quick access.

The presenter system uses a relational database to maintain a persistent store of content and plans. As Django's ORM abstracts the database away and supports many systems, the choice of RDBMS is not as important as it would be if I were not using a framework, therefore I used the RDBMS with which I am most familiar, MySQL[3].

The system was developed on a VPS provided by Linode on which I configured Ubuntu 10.04 with all the tools mentioned above. Nginx[13] was configured with Gunicorn[2] to serve the Django application, it was also used to proxy to the Node.js server. I also configured a development environment using tmux[10] (a terminal multiplexer, similar to GNU Screen) so that I could maintain a

development session across multiple connections, as well as allowing me to quickly switch between terminals.

Vim[11] was used as the primary editor for development of the system, as I felt it was the best option for developing in the terminal on a remote server and configuring it with plugins such as Command-T and Pyflakes vastly improved productivity.

With this configured, I developed by SSHing to the server from the computer I was working on and resuming the tmux session, this allowed me to change from a University computer to my home computer and be able to immediately resume where I left off.

I used the Git[1] version control system for versioning and backing up the project code and documentation. This was mirrored on a private remote hosted by GitHub[7] on a free Student/Education package. I periodically cloned the repository to my local computer, a separate server and a USB drive for backup, this meant that the code was available to me in five places, should any failures occur.

6 Design and Implementation

6.1 Overview

Diffuse is the final product of this project. Named after the concept of 'diffusion' - "the spread of particles through random motion from regions of higher concentration to regions of lower concentration"[15], drawing comparisons between the diffusion of particles and the diffusion of knowledge/information.

While the system consists of three systems, they appear to the user as two, and are therefore named as such:

Diffuse Also used as the name for the main presenter system.

Flux The distribution and 'client' end of the system.

Content consists of two primary types of entity:

Motes
: An individual item of content to be broadcast to attendees. Motes can be of a certain 'type' which define the fields of the mote, and how the mote is displayed to the presenter and the attendee. Can be considered analogous to a 'slide'.

Plans
: A plan is a set of motes; a presenter will likely want to group motes together for a certain meeting, presentation or lecture. Clients enter a plan-specific code on the client in order to begin receiving any motes pushed as part of this plan. Continuing the 'slide' analogy, a plan would be the equivalent of a slideshow.

Both motes and plans are created by a logged in presenter on the presenter system (Diffuse). These can be accessed only by the user who created them. During a presentation, the user uses the presenter system to 'push' motes out to clients accessing the relevant plan.

A client using Flux enters the plan code provided by their presenter, which will 'subscribe' them to receive motes pushed from that plan.

When a mote is pushed, it is first passed to the distribution server and cached. The distribution server then attempts to push the mote forward on to subscribed clients.

The three system parts operate independently, communicating over HTTP, WebSockets and the Redis PubSub protocol.

6.2 Presenter System

The presenter system is developed using the Django framework and therefore uses an interpretation of the MVC design pattern, referred to by the designers of Django as the 'MTV' pattern (Model, Template, View). This pattern defines 'model' the same way as in MVC, as the definition and structure of system data, 'view' as 'which data is presented' and 'template' as 'how the data is presented'[16].

A base 'Mote' model serves as the central model around which the whole system is built. Mote types derive from this model, adding their own fields where required. This polymorphic design, combined with Django's concept of 'apps' both aid with the requirement to create a modular, extensible system.

The Django 'app' concept allows sections of the system to be separated in to individual modules, with their own models, views, resources and templates. To support modularity in Diffuse, new mote types are created in separate Django apps containing a model which extends the 'Mote' model, implements a serialisation method on the new model, and defines how the mote should be represented on the client and the presenter systems. Developers wishing to extend the system need only add a new app to the presenter system and the entire system will now take advantage of it - this is the only place in the system where content types need to be defined - the distribution and client system do not need to be updated.

Diffuse makes available five content types by default to demonstrate the flexibility of the system:

Question and Answer - mote_qa
: Defines a 'Question' model derived from Mote, as well as an 'Answer' model with a many-to-one relation between answers and questions. Presented on Flux as a question with a list of answers. Selected answers are immediately recorded and shown as a dynamically updating graph on the presenter interface.

Slide - mote_slide
: A basic 'slide' style mote type, defining a single 'Slide' model, allowing the presenter to push non-interactive HTML content to users.

Associate - mote_associate
: Defines an 'AssociationGroup' model derived from Mote, along with an 'Association' model with a many-to-one relation between associations and association groups. An association is defined as two strings, the left side, and the right side. When presented to a client, both the left side and right side are randomised, prompting the user to 'join up' the correct association pairs.

Heatmap - mote_heatmap
: A simple 'Heatmap' model, extending Mote, contains an attribute for an image. This image

is shown to the client, and any clicks on the image by the client are shown as a ''heatmap' to the administrator.

Feedback - mote_feedback
The'Feedback' model is another simple model which simply defines a 'description' attribute. When pushed to a client, the description attribute is displayed along with a text entry form. When this entry form receives input, the presenter interface is updated. This is designed to facilitate instant feedback from a class/meeting.

The Mote model has three significant methods, all of which are used for communication with the distribution server.

as_dict
Returns a Python dictionary of model instance attributes and metadata. Derived classes override this method in order to add their own data to the dictionary. This is used to build a JSON representation of the model instance to be sent to the distribution server.

cache
Cache uses the redis-py[**?**] library to store the JSON representation of the mote in Redis, making it available at the key: mote: [mote_pk].

6.3 Client System

Design of the Client system Browsers are available on the vast majority of user agents in variatons incarnations - in my research I noted that every platform I studied had a proficient browser, most being based on the WebKit[14] open source project (providing HTML rendering and JavaScript implementations) which is also used for Google's Chrome and Apple's Safari browsers. From past experience, I know how web applications can be used for excellent cross-platform support, backed up by research in to existing multi-platform ARS' systems, which all appear to be web-based.

A web-based client solution would be accessible from any device with a browser that supported full HTML, JavaScript and some form of real-time communication protocol (even if emulated), features which all of the devices I researched have. No device-specific code would be required, although styles for different screen sizes would be necessary. It would also be possible to use technologies only present in more modern browsers (such as HTML5 features) by making use of graceful degradation on devices that do not support it.

I will be using a browser-based solution for the client as I feel it is the most appropriate option for a truly multi-platform system. While I may lose some of the flexibility of device-native applications, this is an appropriate compromise. Past experience with web applications using JavaScript and an understanding of how to implement a real-time application also contribute to this decision.

The web application will be a thick client, with the majority of operations performed on the client, this means heavy use of JavaScript and HTML5 features where applicable. The client will connect to a server from which it can retreive content.

6.4 Real-Time Communication

Discussion of communication between client and presenter Being able to handle many concurrent connections through the use of an event based server solves part of the problem, but there are still

issues if the code being written is blocking (the main execution loop of the program must wait for an event to complete before it can return, holding up other processing). Event driven programming works in the same way as an event driven server - using a main event loop to support callbacks from functions so that operations can continue while I/O is being performed, operating on the results only when they return.

This type of programming paradigm is implementable in the majority of programming languages, although it is more prevalent and easier to implement in functional languages such as Erlang and Scala. There are also a number of frameworks for other languages available which support event-driven networking, such as Twisted (for Python), EventMachine (for Ruby) and Node.js (for JavaScript).

As I am not familiar with functional languages, I will not be attempting to implement a server in Erlang or Scala, but the existing event-driven networking frameworks interest me as they do not require learning a new language and seem relatively popular. Twisted and EventMachine are mature frameworks, both with a large community and many extensions, but they both seem too bloated for what I need. Node.js is a relatively new framework which is rapidly gaining popularity, primarily because of its speed and simplicity - it interests me because it uses JavaScript which is also used by my client system (an opportunity for more code reuse and resource sharing) and is a language with which I'm familiar. There is also a library available for it which abstracts a lot of the client-side communication issues as I will detail in the next section.

Web-based applications aren't usually known for taking advantage of real-time communication, it wasn't until the increased popularity of Ajax that we saw anything beyond standard request-response style websites. Even now, the most common type of interactions require the web browser to make a request to the server before it updates anything. However, we are now beginning to see 'comet'[4] style web sites, where the client can 'listen' to a server, updating whenever something changes in applications such as chatting. This is made possible using various techniques which either create or emulate 'socket' style connections:

WebSockets
: An implementation of a communication channel over TCP for use implementation in web browsers as part of the W3C HTML5 specification. Currently only implemented in the latest version of modern browsers, excluding Internet Explorer.

Flash Sockets
: TCP sockets can also be implemented using the ActionScript 3.0 Socket class, requiring the browser to load a Flash file and therefore requiring the Flash browser plugin. While this is the next best thing to using WebSockets, many mobile browsers do not fully support Flash.

Ajax long polling
: A technique which is most commonly used for real-time web applications as it only requires a browser to support the XMLHttpRequest object, commonly used for Ajax interactions. This technique makes an asynchronous request to the server which the server keeps open until it has new data to send. As soon as new data has been received, the browser starts a new long polling request. While this technique is available on more devices, it is also slower and causes the browser to constantly be in a 'loading' state.

Multipart streaming
: The XMLHttpRequest object can also be used to react to a server using a multipart content type; suggesting to the browser that the response will be delivered in multiple parts. The

browser keeps the connection open and calls a callback function whenever a new part of the response is delivered.

There are other techniques, but the above are most commonly used. Unfortunately, the only one which reliably works on a large subset of browsers is long polling. I could implement the system using just this techniques, but it would then be unable to take advantage of the faster protocols such as WebSockets, available on the latest mobile browsers (such as Safari on iOS 4.2.1).

A library is available for Node.js, socket.io which allows both the client and server to determine which technique to be used and fall back to it as required, with an API to abstract these details away. This would allow me to support many different browsers with whatever technique they require without having to re-implement all the methods and is therefore what I will be using for this part of the system.

6.5 Pushing Content to the Client

To have content pushed from the presenter system to the client, I will need a way for one system to tell the other what content to broadcast and when.

To begin with, it is necessary to figure out in what format I should be serialising data for transfer between all the systems involved. As both Python and JavaScript have native support for serialising to/from JSON (JavaScript Object Notation), a lightweight format designed for data transmitted between a server and web application, this seems like a natural fit for this project.

Sending the data from presenter to the node.js server could be done by sending it a POST request with the JSON as the value, which could then parse the object and determine which clients to send it to. The node.js server will then need to cache this object so that all future request for it from the client will not cause an additional request back to the presenter, leading back to the original bottlenecking problem.

It would be possible to simply have the distribution server store the JSON in memory, but if this system were serving tens of sets of audiences at a time, this caching implementation would best be suited to a tool designed for this type of use. Key-value databases are particularly useful at this type of task, allowing very quick retrieval of content by key from memory. Memcached is a commonly used implementation, but I have experience with Redis, a VMWare product which performs a similar task, has a very simple, clean interface and has libraries for both Django and Node.js.

Redis can also be used as a Publish/Subscribe server - allowing me to do away with the POST request by having Django publish the latest object ID to be broadcast directly to a pub/sub channel which the distribution server is subscribed to. On receiving an update from the channel, the node server can request the JSON object from the presenter server, cache it in Redis and then distribute it out to clients from the cached copy.

7 Problems

Any issues encountered during development

8 Testing Strategy

How I tested - use iteration documents as appendices

9 Critical Evaluation

Lots of stuff about how good it was

References

[1] Scott Chacon. Git. `http://www.git-scm.com/`.

The project was versioned using the Git version control system.

[2] Benoit Chesneau. Gunicorn. `http://gunicorn.org/`.

A Python WSGI server used to serve the Django application.

[3] Oracle Corporation. MySQL. `http://www.mysql.com/`.

MySQL was chosen as the RDBMS used for storing data on the presenter system. I did not use the site for documentation, as using the Django ORM reduced the need for direct interaction with the database.

[4] Dave Crane and Phil McCarthy. *Comet and Reverse Ajax: The Next Generation Ajax 2.0*. Apress, 2008.

Read the first three chapters which outline what Comet applications are, how they work and how they can be implemented.

[5] Django Software Foundation. Django Project. `http://www.djangoproject.com/`.

The Django web framework was used for the presenter system to abstract away a lot of the functionality which was not considered a defining point of the project. The Django documentation on this site is well-written and complete, it was often referenced and proved very useful during development.

[6] Python Software Foundation. Python. `http://www.python.org/`.

Python was the language chosen for the implementation of the presenter system. Documentation provided on this site is complete and detailed.

[7] GitHub Inc. GitHub. `http://www.github.com/`.

A Git remote was set up as a private repository on GitHub.

[8] Inc. Joyent. Node.js. `http://www.nodejs.org/`.

Node.js was used to write an event-driven distribution and forwarding server for content. Documentation on Node was relatively poor at the time of development, but the site still served as reference material for Node.js functionality.

[9] The jQuery Project. jQuery. `http://www.jquery.com/`.

jQuery was the library of choice for frontend JavaScript, this site provided excellent documentation for referencing jQuery functionality/

[10] Nicholas Marriott. tmux. `http://tmux.sourceforge.net/`.

tmux, a terminal multiplexer similar to GNU Screen was used to enable me to maintain a development session using multiple terminals.

[11] Bram Moolenaar. Vim. `http://www.vim.org/`.

The Vim editor was used as the primary editor for coding and development of the system.

[12] Salvatore Sanfilippo. Redis. `http://www.redis.io/`.

Redis was used for multiple purposes in the project, including caching and distribution. The Redis site has exceptional documentation on all commands useable and proved very useful during development.

[13] Igor Sysoev. nginx. `http://nginx.org/`.

Nginx, a lightweight HTTP server, was used to proxy to all services required by the system.

[14] WebKit. The WebKit Open Source Project. `http://www.webkit.org/`.

Provides details on support for HTML5 and JavaScript technologies in WebKit

[15] Wikipedia. Diffusion. `http://en.wikipedia.org/wiki/Diffusion/` [From version created at: 5:45 18 April].

Used for inspiration for project name.

[16] Wikipedia. Django FAQ - Django appears to be a MVC framework, but you call the Controller the view, and the View the template. How come you dont use the standard names? `http://docs.djangoproject.com/en/dev/faq/general/`.

Referred to for information on the Django designers' interpretation of the MVC pattern, MTV, and their description of it.

[17] William B. Wood. Clickers: A Teaching Gimmick that Works. *Developmental Cell*, 7:796–798, 2004.

A report on the use of clickers in teaching - used to research the validity of the project