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.

*Todo list

Some abstract 3
A discussion on how I came to a decision on the final requirements/objectives. 5
Perhaps separate design and implementation? 6
Figure: Possibly a diagram of how the systems communicate 7
Also considered? 10
Also considered? 12
Also considered? 12
Something about key structure in Redis 13
Also considered? 13
Also considered? 14
Also considered? 15
More problems - perhaps saving error when demoing? 16
Expand if possible 17
Expand if possible, refer appendices 17
Create appendices for testing 17
Expand if possible 17
Lots of stuff here 17

2 Abstract

Some abstract

Contents

1 Acknowledgements **2**

2 Abstract **3**

3 Background and Objectives **4**
3.1 Context 4
3.2 Objectives 4
3.3 Relevant Literature and Existing Systems 5

4 Development Process **5**
4.1 Process Model 5
4.2 Planning 5
4.3 Implementation Tools 5

5 Design and Implementation **6**
5.1 Overview 6
5.2 Presenter System 7
5.2.1 Modularity and Mote Types 8
5.2.2 The User Interface 9
5.2.3 Rendering Feedback 10
5.3 Distribution Server 11
5.3.1 Dealing with Scale 11
5.3.2 Distributing the Motes 12
5.3.3 Server Design 13
5.4 Client System 14
5.4.1 Why web-based? 14
5.4.2 System Design 14
5.4.3 User Interface 15

6 Problems **15**
6.1 Communication protocol 15
6.2 Dynamic resource loading 16

7 Testing **16**
7.1 Testing during development 16
7.2 Iteration Testing 17
7.3 Automated Stress Testing 17

8 Evaluation **17**

3 Background and Objectives

3.1 Context

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"[26].

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.

3.3 Relevant Literature and Existing Systems

Add relevant literature

4 Development Process

4.1 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.

4.2 Planning

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

4.3 Implementation Tools

The system was implemented in three parts, each communicating over network services. The presenter system was primarily developed in Python[4], using the Django[3] 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[6] 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[7] library.

Redis[15], 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[2].

The system was developed on a VPS provided by Linode on which I configured Ubuntu 10.04 with all the tools mentioned above. Nginx[19] was configured with Gunicorn[1] to serve the Django application, it was also used to proxy to the Node.js server. I also configured a development environment using tmux[9] (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[12] 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[21] version control system for versioning and backing up the project code and documentation. This was mirrored on a private remote hosted by GitHub[5] 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.

5 Design and Implementation

Perhaps separate design and implementation?

5.1 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"[24], 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.

Missing figure

Possibly a diagram of how the systems communicate

5.2 Presenter System

The presenter system is developed as a web application 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’[25].

Also considered: *Implementing the presenter system as a native, ‘desktop’, non-web-based application. However, it is more likely that a presenter will have access to a browser than the ability to install and configure software on systems. I also have more experience developing web based applications than I do native applications, and felt the simple database interactions I was planning would be more suited to a web-based system.*

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.

5.2.1 Modularity and Mote Types

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.

Also considered: *Implementing a more abstract way of defining and extending content types, perhaps using an EAV (Entity-attribute-value) database model. This would reduce the need for developers to write code to add new types. However, this would also mean creating a framework for automatically generating code for the client representation and feedback representation, so I felt this would complicate the project beyond its scope.*

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[10] library to store the JSON representation of the mote in Redis, making it available at the key: "mote:[mote_pk]". This uses the as_dict method and does typically not need overriding by derived classes.

push

Uses redis-py to publish a message to the Redis pub/sub channel specified by a parameter to the method. The message contains the event string and some data, in this case, the event 'adminPublishedMote' and the ID of the mote. It also sets a key 'plan:[plan_id]:latest_mote' to the ID of the latest mote, so that clients that aren't currently subscribed to the plan's channel can find the latest mote when they choose to subscribe.

JSON was used for mote serialisation as both Python and JavaScript have native support for it, it's also very lightweight, designed for data interchange between web applications where there is no need for additional context to be present in the serialised data.

Also considered: *Alternative forms of serialisation, including XML and 'pickling' in Python. XML was too heavyweight and contained a lot of context data which I didn't consider necessary - it was also more difficult to parse. Language-dependent serialisation methods such as pickling were hard to decode cross-language.*

Also considered: *Caching serialised motes directly in memory on the distribution server, rather than using Redis. However, I felt that if the system were serving thousands of clients at a time, using a key-value store designed for this type of storage would be more suitable.*

Also considered: *My original design had the presenter system send a post HTTP request directly to the distribution server when a mote was pushed. A version of Redis was released during development which implemented its own Pub/Sub protocol so I switched to this to simplify and speed up communication.*

5.2.2 The User Interface

The user interface for managing plans and motes was designed to be simple and easy, this was achieved by reducing all operations down to two modes of interaction:

Lists

Lists are used to represent available plans and motes. Lists are always present in the first two columns of the interface. Three actions are possible on lists and list items:

Selecting

In the case of plans, this displays the motes within that plan, for motes, this broadcasts the mote.

Editing/Deleting
These options are presented modally on hovering over the list item. Editing takes the user to the second type of interaction, 'forms', while deleting prompts the user to permanently remove the item.

Adding
At the bottom of a list, there is usually a way to add a new item to that list. The item created is always the same as the type of item in the list - the add button at the bottom of a plan list will always create a new plan.

Forms
Forms modify attributes on items - while they can contain different fields, they all take the same structure whether the user is adding or editing an item. Editing items displays the same form as adding an item, only with fields pre-populated. Forms are only ever shown in the last column of the interface.

5.2.3 Rendering Feedback

The final part of the presenter system is how feedback from clients is displayed to the presenter. When a mote is pushed, the last column updates with some representation of received data - how this data is displayed is specified by the individual mote applications, and can therefore vary between mote types.

Feedback display is implemented in JavaScript by creating a JavaScript file in the motes' app containing a class derived from Renderer. A set of events are bound to the active Renderer, which call methods on the class when they are triggered. Each method[1] needs to be overwritten by the derived, mote type-specific class:

render
Builds the relevant representation of the results within the container with the 'renderer' ID. This is called when the renderer is first initialised.

redraw
Updates the representation with any new data received.

updateData
Updates the renderers' set of responses with any new responses passed to the method, if the 'append' parameter is true, responses are added to the existing set. If it is false, responses are cleared before new ones are added.

clientDisconnected
When the client disconnect event is fired, the renderer may choose to remove that clients' response from the display, this method is used to achieve this.

An example implementation of a renderer is that used for the *Question and Answer* mote type. For this, the render method creates a column chart using the HighCharts[18] library. the updateData method directly modifies data in the object created by HighCharts, rather than drawing a new chart every time - this created a smooth 'sliding' effect on the chart as new data arrived.

[1]Note that methods in JavaScript typically use camel case for method names, whereas in Python, PEP8 dictates that methods should be all lowercase, with underscores separating words

Also considered?

5.3 Distribution Server

As this system is designed to be used in large organisations, such as universities, it is likely that there will be thousands of clients attempting to request motes simultaneously. Combined with the need to maintain a persistent connection to each client for broadcasting content and provide updates within a short time, I felt that a traditional server would not be suitable, so the distribution server was introduced.

5.3.1 Dealing with Scale

My concern with broadcasting content in real-time to hundreds of devices at a time using a typical web server was that it is likely to struggle with the number of requests. Techniques used to emulate real-time communication either require a permanent connection open to the server, or hundreds of requests to the server per minute. Traditional web servers such as Apache use a process-based model, where each additional request requires a new thread meaning many hundreds of connections being opened and closed per minute not only causing high memory usage, but incurring quite a large processing overhead, increasing the time it takes to serve a request as the volume increases.

I chose an event-based server as the solution to this issue, which use a single thread and an event loop, removing the requirement for context switching. While this does mean significantly lower memory usage and overhead, it doesn't prevent operations in code from blocking, particularly those that involve I/O. For example; a request which queries the database will temporarily hold up the loop until the database responds, as we require a very quick response from the server, this may prove unacceptable.

To solve this, I chose to implement a distribution server in an event driven programming language, which work 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 did not be attempt to implement a server in Erlang or Scala, but instead looked at existing event-driven networking frameworks as they did not require learning a new language and seemed 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 needed. Node.js is a relatively new framework which is rapidly gaining popularity, primarily because of its speed and simplicity - it interested 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. It is also designed to be exceptionally easy to write a simple server using.

Using the event based constructs in Node.js, I built a server tailored to the type of communication I needed, thus tackling the blocking I/O problem on two fronts; reducing the amount of I/O time required through caching, and attempting to prevent I/O from blocking. To cache, I had the presenter system push serialised versions of motes to Redis, and used the distribution system to quickly retrieve and push these out to clients.

Also considered?

5.3.2 Distributing the Motes

As this application required content to appear on all client devices within a reasonable period of time after they are broadcast, it needed to use some form of real-time communication method. As both the client and presenter are web-based, choices for doing so are quite limited due to the HTTP protocol being entirely request-response oriented. If I chose to use code which was not being executed in the browser, I could use networking technologies such as sockets to allow the client to 'listen' for communication from the broadcaster.

However, there are various ways to support a communication channel in web applications:

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.

Unfortunately, the only one which reliably works on a large subset of browsers is long polling. I could have implemented 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).

To save writing implementations of each communication method, I used a library available for Node.js, socket.io[14], which allows the client and server to determine which communication technique to use depending on what they each support. It provides a simple API to abstract these techniques away and allowed me to simply request some data be transmitted.

Also considered?

5.3.3 Server Design

The implementation of the server was largely assisted by the use of the Connect middleware framework for Node.js. Connect created a basic HTTP server, with middleware for serving static files (used to serve the client application) and for session management.

On top of the basic HTTP server, an event-based cross-system communication micro-protocol was developed. A typical communication would be a JSON string containing two top-level objects: 'event' and 'data'. The former contains a single camel-cased string describing the event which triggered the communication. The latter contains all the relevant data pertaining to the operation.

Operations were triggered by events from the Pub/Sub channel and from the connected sockets. For example, when the event with the event string 'adminPublishedMote' was published on the Pub/Sub channel, a function is called which parses the 'data' object and forwards the parsed mote to clients connected to the server and subscribed to the relevant plan.

Along with supporting communication through the socket one way (pushing motes out to clients), the server also needed to deal with responses coming back from clients. To do this, it would listen for the 'clientRespondedToMote' event on the socket and store the answer in a plan/mote specific hash in Redis.

Whenever the distribution server needs data, it is requested from Redis using the node_redis[13] library. Redis, being very efficient at retrieving data by key from memory (a time complexity of O(1)) required very little time to fetch the required data (although even then, due to the event-loop nature of the server, it continued processing requests).

Redis was used to great effect to reduce the amount of I/O needed to store and retrieve data on the server. Rather than requesting data from the presenter server when it was needed (triggering a request to the RDBMS or filesystem), the presenter was designed to cache any relevant data in Redis before the distribution server needed it. This included serialised motes, mappings between plan 'access codes' and plan IDs as well as the current status of plans.

As the system does not request a user be logged in on the client to respond to motes, it is necessary to make some assurances that a single device is a single user. To do this, Connect's session management middleware combined with the Connect Redis[11] session backend to store user sessions in Redis. Session keys are used throughout the system as a unique user identifier, so when a user refreshes Flux, rather than creating a new response, their old response is overwritten.

Using sessions also allows a 'clientDisconnected' event to be fired when the session expires (when the user no longer has Flux open). The server then removes that users' response and notifies the presenter, which will in turn call the 'client_disconnected' method on the currently active renderer, removing their answer in realtime.

Something about key structure in Redis

Also considered?

5.4 Client System

The client system is a critical component for meeting the requirements of the project. Referred to as Flux, this is the system which attendees connect to in order to receive broadcasted content. While the resources for this component are served both from the distribution server and the presenter system, it is considered a separate entity as the majority of its operations are on the client-side, in JavaScript.

5.4.1 Why web-based?

In my research I noted that every platform I studied had a proficient browser, most being based on the WebKit[23] 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 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 is required, although styles for different screen sizes are necessary. It is also 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 therefore chose a browser-based solution for the client as I felt it was the most appropriate option for a truly multi-platform system. While I may have lost some of the flexibility of device-native applications, I considered this 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.

Also considered?

5.4.2 System Design

On loading, Flux attempts to connect to the distribution server using the client-side version of the socket.io library. This establishes a realtime method through which to communicate for the rest of the session. The same event-based micro-protocol was implemented here in order to deal with events from the distribution server. For example, receiving a communication with the event 'serverPushedMote' triggers a function which updates the display. These two features combine to create a simple web page which updates in realtime, on request.

In order to render a pushed mote, Flux performs a variety of operations triggered by its update_mote function. It begins by updating the list of past motes available in the history panel. It then attempts to load the scripts and styles required by that mote type. This is done by using yepnope.js[16] to check whether the scripts are already loaded, and if not, request them from the distribution server.

The distribution server proxies to the motes' resources the developer created for the presenter system and returns them Flux.

Once these resources are loaded, the returned JavaScript resource will contain a subclass of ClientMoteRenderer (named after the object identifier) which is instantiated and has its 'render' method called. This method fetches the mote types' template (again, created by the mote type's developer) which will be written using the mustache[22] template language and rendered using mustache.js[8]. The ClientMoteRenderer subclass will likely implement additional functionality which will eventually trigger Flux's 'respond' function, returning the users' response to the distribution server.

For example, QaRenderer, the implementation of ClientMoteRenderer which is loaded when a mote of type Question and Answer is pushed calls the respond function with the answer the user has selected and updates the display to show some feedback to the user.

Also considered?

5.4.3 User Interface

It was important that this part of the system was very easy to use, so I attempted to limit the amount of choices a user had at any one point. On initially loading the system, they user is prompted to enter the access code for the plan they wish to join. This is the only option they have available at this point. Entering the code will trigger a communication with the distribution server to check whether the access code matches an existing plan. If it does, the page will be updated with the most recently pushed mote to that plan.

The renderers implemented for the built-in mote types are designed to respond immediately on interaction - there is never a need to select a 'send' button so that the user never needs to understand the concept of communication between client and server.

There are two other interface elements available to the user, neither of which are critical to its operation. The first, visible to the top left, is the currently active plan. Selecting this will allow the user to change to another plan. The second, available to the top right, is a history panel. Selecting this will drop down a list of all past motes, allowing the user to navigate back to a previous mote if desired.

6 Problems

6.1 Communication protocol

Once I had the distribution server communicating in both directions with the client, I had the problem of making communications trigger operations on each system. I struggled with figuring out a way to establish a light RPC protocol, until I came across the nodechat.js[17] which used an event based model, including an event string as part of the communication between systems.

Implementing this was simple, and it made my system much more flexible, however, I did not design the triggered events before implementing them, so I found myself getting confused by what was triggering which events, when to respond to them, and how to 'bubble' them through all three systems when required. To resolve this, I established a naming convention for events; starting with the system from which they originated; ‘admin’, ‘server’, ‘client’, followed by the operation which triggered the event (rather than the operation the event should trigger) in the past tense.

For example, the ‘clientRespondedToMote’ event is triggered by the client when the user responds to a mote. This event is returned to the admin by triggering the ‘serverPushedResponse’ event.

The other issue I had with this protocol was maintaing context between communications. For example, if the client requested the existence of a plan from the distribution server, it was necessary for the distribution server to know which client requested it, and the client needed to know when it received a response.

As there would be no concept of a ‘response’ to a certain communication in this protocol, I had trouble figuring out how to know when the distribution server was responding. My eventual solution to this was to include just enough context in the sent message so that the response could be identified and to remove the need for context from events. For example, the client identifier would be sent with the request for the existence of a plan so that the distribution server could trigger a ‘serverSetPlan’ event only on the requesting client.

6.2 Dynamic resource loading

One of the more complicated issues I came across when developing Flux was the issue of dynamically loading mote-type specific resources on request.

The first issue I encountered with this was a race condition where the system would attempt to instantiate the loaded object before it had successfully loaded. I attempted to fix this by instantiating the object in a callback triggered by the resource loader, but the race condition still existed when the resource had loaded but the browser had not fully interpreted the new script. I ended up fixing this by using yepnope.js, which had built in callback functions which worked perfectly.

The second issue I had with this was with maintaining mote type modularity. I originally had all mote-specific resources for Flux stored in a separate directory from the main mote type definition, this allowed me to use the distribution server to serve these resources. However, I wanted all mote-specific items to be in a single folder, regardless of whether they were used for Diffuse or Flux. To achieve this, I moved these resources in to the folder which would typically be served by Django and updated Flux to find them on the Diffuse domain.

However, because the client was being served from the Flux domain, and JavaScript resources were attempting to load from another domain, the same origin policy implemented in most browsers prevented them from loading. I tried to work around this by requesting the JavaScript resource using JSONP, but this added extra unnecessary complexity. My final solution was to use Nginx to proxy requests for the resources on the Flux domain to the Diffuse resource folder.

More problems - perhaps saving error when demoing?

7 Testing

I adopted a number of testing strategies during the development of the project, each serving a different purpose.

7.1 Testing during development

An informal testing strategy was used during development cycles. This usually consisted of

- Creating a 'test' entries in the presenter system database and ensuring they worked as I built the system.
- Adding debug code (logging, prints, debug output) to systems.

The use of Django's fixtures system made it easy to load a set of test data in to the database during development. Combined with the use of South[20] to create data migrations when models were changed, persistent test data was a valuable way to ensure the system was working.

This approach was used primarily to ensure I could clearly see the workings of the system as it was built. It was not intended to ensure the system met specifications.

Expand if possible

7.2 Iteration Testing

At the end of every iteration, I went through a testing plan to ensure that what I had built met the specifications determined at the start of the iteration. Feedback from this testing then went in to planning the next iteration.

Expand if possible, refer appendices

Create appendices for testing

7.3 Automated Stress Testing

As this system is built around many clients connected simultaneously, I needed to test how the system would behave under normal use. As I only had access to a few client systems at a time, I wrote a set of scripts to emulate the behaviour of clients in order to ensure that the distribution system and presenter system renderers could cope with normal (and abnormal) numbers of responses.

Expand if possible

8 Evaluation

Lots of stuff here

References

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

A Python WSGI server used to serve the Django application.

[2] 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.

[3] 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.

[4] 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.

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

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

[6] 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.

[7] 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/

[8] Jan Lehnardt. Mustache.js. `https://github.com/janl/mustache.js/`.

Mustache.js is the JavaScript implementation of Mustache, used to parse and render motes' templates in Flux.

[9] 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.

[10] Andy McCurdy. Redis-py - GitHub. `https://github.com/andymccurdy/redis-py`.

The Redis-py library was used as to interface with Redis from the presenter system. This is the official repository for the library.

[11] Vision Media. Connect Redis on GitHub. `http://github.com/visionmedia/connect-redis`.

Connect Redis was used as the session backend for Connect's session middleware.

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

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

[13] Matthew Ranney. node_redis on GitHub. `http://github.com/mranney/node_redis`.

node_redis was the library used to interface with Redis from the Node.js distribution server. Initially, the redis-node-client library was used but the developers stopped supporting it and node_redis replace it.

[14] Guillermo Rauch. Socket.IO. `http://www.socket.io`.

Socket.io was the library used on both the client and server to implement realtime web-based communication methods.

[15] 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.

[16] Alex Sexton and Ralph Holzmann. yepnope.js - A conditional loader for your polyfills. `http://yepnopejs.com/`.

yepnope.js was used as a loader for mote type-specific scripts. While it also supports conditional resource loading, this feature was not used for the project.

[17] Justin Slattery. NodeChat. `https://github.com/jslatts/nodechat`.

Used as inspiration for the event-based communication model used in Flux.

[18] Highslide Software. Highcharts - Interactive JavaScript charts for your web projects. `http://www.highcharts.com/`.

HighCharts was used to render results for some content types. I found the library to be extremely flexible and well documented, and it worked exceptionally well with the live updating required by the system. HighCharts was available under Creative Commons Attribution-NonCommercial 3.0 for non-commercial projects.

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

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

[20] Torchbox. South. `http://south.aeracode.org/`.

South is a data migration system for Django. It was used to maintain test data while adding and changing models during development.

[21] Linus Torvalds. Git. `http://www.git-scm.com/`.

The project was versioned using the Git version control system.

[22] Chris Wanstrath. Mustache. `http://mustache.github.com/`.

Mustache was the templating language used for defining the structure and layout of motes rendered on the client.

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

Provides details on support for HTML5 and JavaScript technologies in WebKit

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

Used for inspiration for project name.

[25] 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.

[26] 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