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: 1 May 2011 *Version:* 1.7

Declaration of Originality

In signing below, I confirm that:

This submission is my own work, except where clearly indicated.

I understand that there are severe penalties for plagiarism and other unfair practice, which can lead to loss of marks or even the withholding of a degree.

I have read the sections on unfair practice in the Students Examinations Handbook and the relevant sections of the current Student Handbook of the Department of Computer Science.

I understand and agree to abide by the Universitys regulations governing these issues.

Surname .....

First Names .....

Number of Pages .....

Student Reference Number .....

Signature .....

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, Lily Usher, Alex Kruzewski and my parents, for testing, feedback, proof-reading and moral support.

Abstract

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, however, they suffer from a few important shortcomings. Using existing mobile devices communicating over a standard network, an alternative ARS has been developed which extends the concept of an ARS while attempting to solve some of the problems with existing systems. The rationale and design for the system is presented, along with a conclusion that the system is largely successful in its goal and a few points on potential future improvements and developments for the system.

Definition of Terms

Presenter
: The user running a presentation. A lecturer, teacher or meeting co-ordinator.

Attendee
: User attending a presentation. A student or one attending a meeting.

Presentation
: A setting in which a single Presenter (occasionally multiple) is interacting with multiple Attendees, typically in order to present information. A lecture, class, talk or meeting.

Contents

1 Background and Objectives

1.1 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.

From using these clickers from a student's perspective, three primary problem areas were noted with the current implementation of these ARS systems:

Accessibility
: 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.

Interaction
: The devices are limited in what can be displayed on an individual user's 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.

Flexibility
: ARS devices are usually single-purpose and rarely used for anything beyond prompting an audience for answers to a question.

While this project does not involve additional research in to the value of ARS 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"[82] while attempting to alleviate the above issues in a new, 'evolved' ARS implementation.

1.2 Objectives

ARSs 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 presentation. 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 creating a 'next-generation' ARS system by focusing on improving the three problem areas: accessibility, interaction and flexibility:

1.2.1 Accessibility

- Improving ease and cost of setup/configuration for system administrators by leveraging existing local area networks, allowing the system to be configured once per site, rather than requiring setup for every presentation.
- Facilitating ease of access by presenters by supporting the use of the presenter system on existing network-connected systems without the requirement for additional software installations.
- Extending the use of ARS systems for attendees by using existing devices with a relatively modern web-browser and the ability to access the local network as clients. Allowing attendees to use their existing smartphones, tablets or laptops rather than using proprietary hardware.
- Removing the requirement to have dedicated hardware installed and available when an ARS is required.
- Simplifying the experience of interacting with the system for both the presenter and the attendee.

1.2.2 Interaction

By using the capabilities of powerful modern portable hardware, the system aims to provide multiple ways by which attendees can interact with the presenter during a presentation. While a typical ARS offers only 'Question and Answer' style interaction, this system will allow a large variety of content to be broadcast and responded to, including images, puzzles and games.

As with standard ARSs, the system is intended to facilitate near-instant, actionable feedback to the presenter from interactions with attendees.

1.2.3 Flexibility

Building the system around an extensible framework allows additional types of content to be created for broadcast by developers, creating infinite possible types of interaction supporting the use of system in many other use cases. By refraining from using any device-specific implementations/code, extending the system to support future devices should be trivial.

1.3 Relevant Literature and Existing Systems

A number of varying ARS systems were researched during project planning, full details on which can be found in Appendix D. The general conclusion reached from this research was that all current ARS systems have not developed beyond the simple 'Question and Answer'

style interaction model, likely due to the restrictions of requiring legacy device support in their software.

2 Development Process

2.1 Process Model

For this project, being able to enjoy the development of the system was important, so a process model which best suited the way the author wrote software needed to be adopted. As the author's typical development methodology is 'hack at it until it works', a process model that injected a degree of organisation in to the process, providing a way to plan and view progress while maintaining focus on the code rather than the documentation needed to be found.

For this reason, the iterative development[75] approach was chosen, as it best suits how the author builds software, allowing blocks of time in which the focus is on writing code, while adding a sense of organisation and planning to the process. While the waterfall model[80] might also have been suitable, the requirement to complete the design before moving on to development was considered too restrictive as it is believed that any software project needs room to evolve during its implementation.

The iterative development model was adapted slightly to suit the project, following this cycle:

1. Produce overall specification document.
2. Begin iteration:
   (a) Produce testing plan for iteration.
   (b) Develop iteration.
   (c) Test iteration to specified testing plan.
   (d) Revise specification from what was learnt during development.
3. Begin next iteration (return to step 2).
4. Re-execute all test plans.
5. Perform additional testing.

2.2 Planning

When planning the system requirements, it was important to decide the scope of the system and what features would be important to create a system which met the desired objectives, created an interesting, distinguishable project, and was fun and educational to build, while maintaining a tight schedule (see Appendix A) and minimising risk.

This was done by taking advantage of the evolutionary process in the chosen development model, planning to incrementally add features based on the success of previous iterations. To begin with, the basic form of the system was decided. This would be a defining 'core'

of the project and would ideally meet the primary objective of the system, to create a networked, mobile device based ARS.

The core consisted of the basic operations of the three planned systems:

Presenter system
Ability to create and manage content, prepare content for distribution and push content to a distribution channel.

Distribution system
Cache and forward pushed content to connected clients.

Client system
Enable the user to authorise themselves to receive content and have the client update to display messages pushed from the distribution server.

These basic operations were decided by establishing what elements distinguished this project from existing ARS systems; primarily the ability to use an existing infrastructure to broadcast content in real-time to existing mobile hardware. These operations would also be the initial implementation of the basic design of the system, around which all additional features would be built. Attempting to implement the basic design early on provided early feedback as to whether any parts of the design would need revising in later iterations.

This core was planned to be built in the first three of six total iterations, intended to be completed by the end of the first two months of development. Had there been personal or implementation issues which would prevent or hinder development of the project after this point, this would still serve as a functional system, reducing the risk of having no working system by the end of the project.

The fourth and fifth iterations were planned to meet additional, less vital objectives. These included: supporting responses from the client, displaying results to the presenter, creating custom mote types and abstracting away any device-specific code. These iterations were expected to deviate from the original plan based on the success of and discoveries made in developing the core.

The sixth iteration was entirely optional, planned for polish, tidying and features which were not necessarily required to meet objectives, but would make the system seem more complete for demonstration and reporting purposes. These changes included tidying the UI, adding additional demonstrable mote types and adding support for additional devices. This sixth iteration would be used if the project was largely on schedule, leaving time for additional polish.

2.3 Implementation Tools

The system was implemented in three parts, each communicating over network services. The presenter system was primarily developed in Python[19], using the Django[18] 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[29] 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[30] library.

Redis[54], 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. Additionally, it was 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 database systems, the choice of RDBMS is not as important as it would be if a framework was not being used, therefore, the RDBMS with which the author is most most familiar was chosen, MySQL[11].

A utility library, django-annoying[2] was used to simplify some elements of Django development, particularly through the use of the render_to decorator which is used to replace the usual requirement to return a render_to_response object in Django views. Additionally, django-debug-toolbar[23] was used to aid in developing and debugging the presenter system, providing varying useful debug information on rendered output.

The system was developed on a VPS provided by Linode[35] on which Ubuntu 10.04[36] was configured with all the tools mentioned above. Nginx[59] was configured with Gunicorn[6] to serve the Django application, it was also used to proxy to the Node.js server. A development environment using tmux[37] (a terminal multiplexer, similar to GNU Screen) was also set up so that development sessions could be maintained across multiple connections, as well as allowing quick switching between terminals.

Vim[42] was used as the primary editor for development of the system, as it was considered the best option for developing in the terminal on a remote server and configuring it with plugins such as Command-T[9] and Pyflakes[1] vastly improved productivity.

With this configured, development was done by SSHing to the server from the computer currently being worked on and resuming the tmux session, this supported changing from a University computer to a home computer while being able to immediately resume where development left off.

The Git[63] version control system was used for versioning and backing up the project code and documentation. This was mirrored on a private remote hosted by GitHub[26] on a free Student/Education package. The repository was periodically cloned to a local computer, a separate server and a USB drive for backup, this meant that the code was available in five places for redundancy, should any failures occur.

3 Design

3.1 Overview

To complement the iterative development model chosen, a form of iterative design was also used. This began with a basic outline system design before anything was programmed. As sections were implemented, the design was revised and added to as required. The following describes the final design after implementing the system, and, where relevant, how it changed from the base design.

While the system was intended to be platform and device agnostic, the increasing popularity of Apple's iPad[53] and other tablet computers to enhance textbooks in education provided an ideal device around which to build and test the system within the time constraints (although it would not use any tablet or device specific implementations). Building interfaces for additional platforms was considered an 'iteration six' task, as long as there were no tablet or device specific implementations, the device could still be tested on other devices, so additional interfaces were not required to meet the objectives of the system.

To create a naming convention for system elements, models and documentation, the system was given the name *Diffuse*, coined after the concept of 'diffusion' - "the spread of particles through random motion from regions of higher concentration to regions of lower concentration"[72], drawing comparisons between the diffusion of particles and the diffusion of knowledge/information.

The system design details three distinct systems: the presenter system, the distribution server and the client system, however, as they appear to the user as two, they 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.

The three system parts were designed to operate independently, communicating over HTTP, WebSockets and the Redis PubSub[78] protocol. Non-system specific communication protocols were chosen so that that if required, components could be switched out later in development if one didn't work as required; for example, if the web-based client system did not work, it would be possible to fall back to a native client. An additional benefit, although outside the scope of the project, would be the potential to implement the endpoints in different ways, as long as they adhered to the same basic protocol. For example, the same system would be able to support proprietary handheld devices, device-native clients (e.g. an iOS device written in Objective-C), etc.

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. The initial design had the user enter their organisational login credentials to access plans. This was changed during the first iteration (while creating the presenter system) for a number of reasons:

- The added implementation complexity of gaining access to an organisation's authentication service, considering each may implement a different service (Shibboleth[28], OAuth[7], etc.).
- Added complexity for the presenter, requiring them to add a list of authorised users to each plan.
- Added complexity for the attendee, requiring them to login to access plans.
- Reduced use in environments where there may not be an established account system (informal meetings, smaller organisations).
- Reduced perception of anonymity - requiring logins means attendees may think their individual responses are being stored with their username.

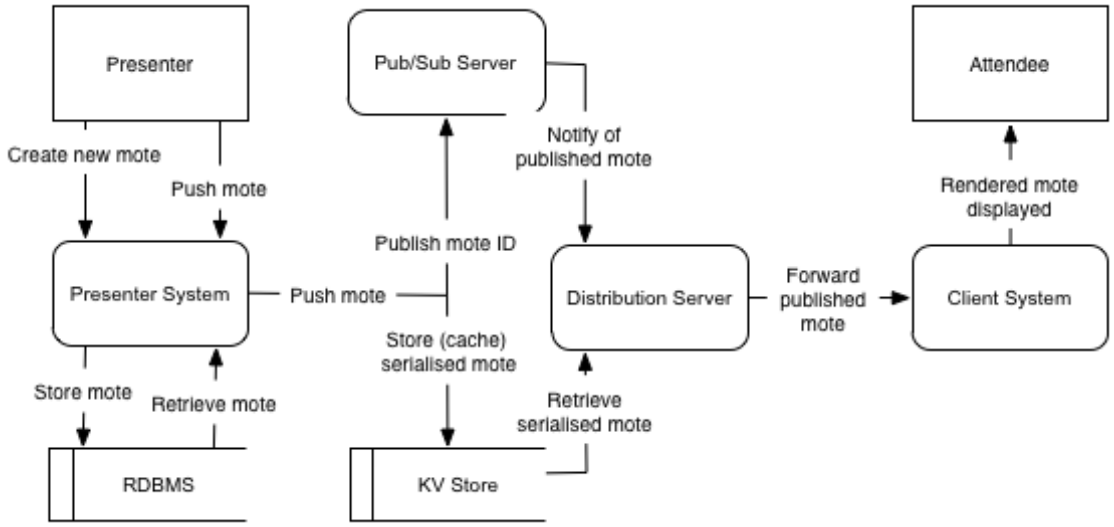
When a mote is pushed, it is first passed to the distribution server and cached. The distribution server then attempts to push the mote forward on to subscribed clients. Figure 1 shows how data flows between the three system components.

3.2 Presenter System

The presenter system was designed as a web application using the Django framework, using 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'[73].

Implementing the presenter system as a native, 'desktop', non-web-based application was also considered. However, it is more likely that a presenter will have access to a browser than the ability to install and configure software on systems. The author also has more

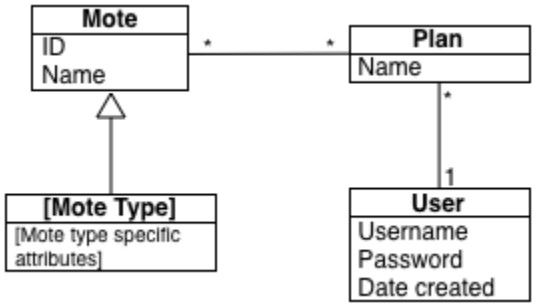
Figure 1: Overview data flow diagram



experience developing web based applications than native applications, and felt the simple database interactions planned would be more suited to a web-based system.

The presenter system was designed to be the primary system for managing and storing motes and plans, it was therefore necessary to establish an idea of the models which would be used for each. Figure 2 shows the initial basic model structure concept.

Figure 2: Original concept model diagram



A base 'Mote' model serves as the central model around which the whole system is designed. This base Mote model would only contain Name and ID attributes (as they would be the only common attribute between all motes), as well as some utility methods. Mote types derive from this model, adding their own fields where required. This polymorphic[77] design, combined with Django's concept of 'apps' both aid with the requirement to create a modular, extensible system.

Also considered was implementing a more abstract way of defining and extending content types, perhaps using an EAV[74] (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 it was felt that this would complicate the project beyond its scope.

A Plan model would be used with a many-to-many relationship to the base Mote type, allowing it to refer to a list of any combination of mote types, using polymorphism to retrieve their full attributes when required. The Plan model would also 'belong', through a one-to-many relationship, to a User, modelled by a username, password and date created. During development, the User model was implemented by Django's built-in 'auth' application, so the final implementation included a number of fields which were not initially planned.

The presenter system required the ability to add, edit and delete both motes and plans. Opting to use Django's built-in generic views, with additions where necessary, would reduce both the design and implementation work needed to create these basic operations.

3.2.1 Modularity and the Mote Model

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. Motes are all treated as their base class, specialised when additional mote-type specific data is required through the use of django-polymorphic[10]

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 should be the only place in the system where content types need to be defined, the distribution and client system should not need to be updated.

The Mote model was designed to use three methods, all of which are used to communicate 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[39] 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 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[16] was chosen for mote serialisation as both Python and JavaScript have support for it. It is 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. Alternative forms of serialisation, including XML[46] and 'pickling'[48] in Python. XML were considered too heavyweight, containing a lot of context data which was not required; it was also found more difficult to parse. Language-dependent serialisation methods such as pickling would have been another alternative, but would be hard to decode cross-language.

Also considered:

- The 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 this was switched to in order to simplify and speed up communication.

3.2.2 The User Interface

The presenter user interface was not planned or designed before development, it was considered that this method often creates interfaces that have had elements 'tacked on' to suit changing features and can often be hard to use due to a lack of understanding of how the system works. While building the system, it becomes evident where elements make the most sense, so creating the UI was an evolutionary process that was not considered fully 'designed' until the end of the last iteration.

One of the primary goals for the user interface during its evolution was to ensure it was simple and easy-to-use. This was achieved by a decision to reduce all operations down to two modes of interaction, inspired by Steve Krug's 'second law of usability': "It doesn't matter how many times I have to click, as long as each click is a mindless, unambiguous choice"[31].

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.

3.2.3 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.

The original design for implementing feedback rendering was to use HTML templates, as with the rest of the system, however, during their implementation in the interactivity iteration (iteration 4), it was noted that all the templates were very similar; a container which had content injected by DOM manipulation[61] in JavaScript. It was therefore decided to do away with the template system and implement them entirely in JavaScript.

The new design had individual mote types specify a JavaScript file in the motes' app containing a object prototype derived from Renderer. A set of events are bound to the active Renderer, which call methods on the object when they are triggered. Each method[1] needs to be overwritten by the derived, mote type-specific object prototype:

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 client's response from the display, this method is used to achieve this.

[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

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[58] 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.

3.3 Distribution Server

As this system is intended for use 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, it was felt that a traditional server would not be suitable, so the distribution server was introduced.

3.3.1 Dealing with Scale

The 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[60] 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.

An event-based server was chosen as the solution to this issue, which uses 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[70], 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, a distribution server was designed 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[33] (for Python), EventMachine[8] (for Ruby) and Node.js (for JavaScript).

As the author was not familiar with functional languages, implementing a server in Erlang[17] or Scala[15] was not attempted, but instead existing event-driven networking frameworks

were researched 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 seemed too bloated for what was needed. Node.js is a relatively new framework which is rapidly gaining popularity, primarily because of its speed and simplicity; it was particularly interesting due to its use of JavaScript which is also used by the client system (an opportunity for more code reuse and resource sharing) and is a language with which the author was familiar. It is also designed to be exceptionally easy to use to write a simple server.

Using the event based constructs[13] in Node.js, a server was designed tailored to the type of communication 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, the presenter system pushes serialised versions of motes to Redis and uses the distribution system to quickly retrieve and push these out to clients.

3.3.2 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 code which was not being executed in the browser was used, it would be possible to 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[12][71]:

WebSockets[22]
: 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[27]
: 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[14]
: 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[81] 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. The system could have been implemented 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, A library available for Node.js, socket.io[52] was chosen, 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.

3.3.3 Server Design

The design of the server was largely based around the use of the Connect[32] middleware framework for Node.js. Connect creates 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 designed. 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. Figure 3 shows a map of events triggered between system components.

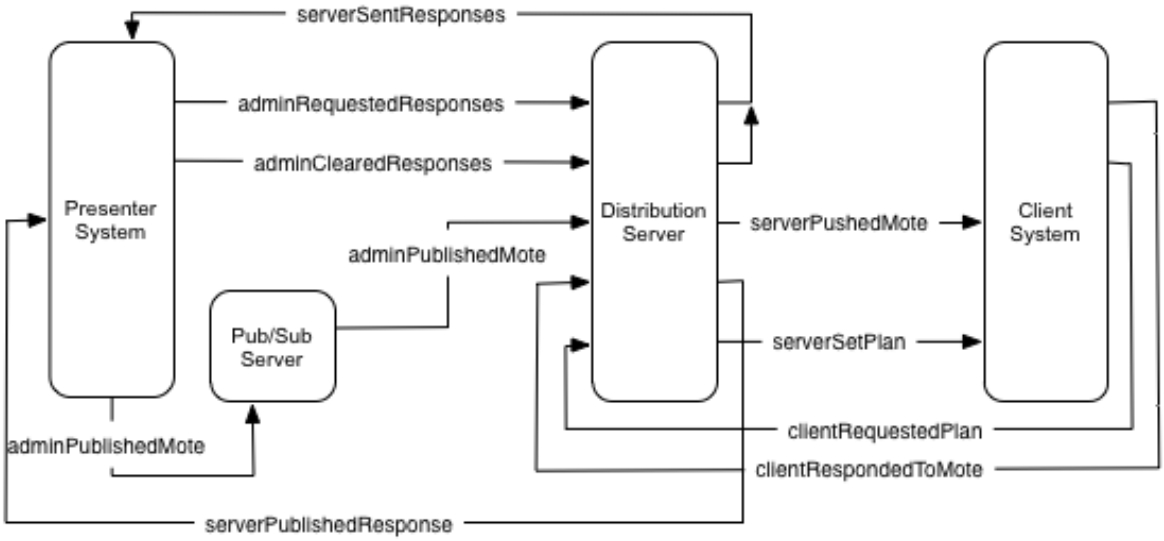
Operations were intended to be triggered by events from the Pub/Sub channel and from the connected sockets. For example, when the event with the event string 'adminPublishedMote' was published on the Pub/Sub channel, a function would be called which parses a 'data' object and forwards the parsed mote 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[51] 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 heavily in the design 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

Figure 3: System Event Map



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.

Due to the design changes made to client authentication so that 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[40] session backend to store user sessions in Redis was used to maintain sessions. 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 user's response and notifies the presenter, which will in turn call the 'client_disconnected' method on the currently active renderer, removing their answer in realtime.

Initially, there was no design or convention for the naming and structure of keys in Redis. During development, it was found that attempting to get/set various keys in multiple places in the application was confusing and hard to follow without a basic design. This lead to the documentation of an outline structure of keys:

Key Structure	Intended Value
mote:[mote_id]	Basic string value containing full JSON for mote with ID of [mote_id]
plan:[plan_id]:mote:[mote_id]	Hash value containing all client responses to [mote_id] subscribed to [plan_id], hashed by client ID.
plan:[plan_id]:latest_mote	ID of latest mote on [plan_id]
plan:[plan_id]:name	Full human-readable name of [plan_id]
plan:[plan_access_code]	ID of plan with access code of [plan_access_code]

Rather than using a separate key-value store, caching serialised motes directly in memory on the distribution server was also considered. However, it was 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. There were also a number of benefits of using a separate store, particularly that multiple systems could communicate directly with it and that it would be simpler to maintain as a separate entity.

3.4 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 (HTML pages, JavaScript, etc.) for this component are designed to be 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.

3.4.1 Why web-based?

In platform research, it was noted that every platform studied had a proficient browser, most being based on the WebKit[68] 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, the author knew 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 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.

Therefore, a browser-based solution for the client was chosen as it was felt that it was the most appropriate option for a truly multi-platform system. While some of the flexibility of device-native applications may have been lost, this was considered an appropriate compromise. The author's past experience with web applications using JavaScript and an

understanding of how to implement a real-time application also contribute to this decision.

A number of alternative options were researched, including common libraries, cross-platform libraries and rich multimedia platforms. A brief discussion on these can be found in Appendix C.

3.4.2 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 used 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[55] 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[67] template language and rendered using mustache.js[34]. 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.

3.4.3 User Interface

It was important that this part of the system was very easy to use. One of the ways this was achieved was by limiting the amount of choices a user had at any one point. On initially loading the system, the 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.

3.5 Mote Types

A number of mote types were designed to demonstrate the system, each for a different type of interaction and use. The only mote type initially designed was Question and Answer, the rest being designed in later iterations once the majority of functionality was in place.

3.5.1 Slide

The Slide is the most basic mote type available in Diffuse, created to demonstrate the basic ability to push non-interactive HTML content to Flux.

When creating a Slide, the presenter is prompted to complete a 'Content' field in which they can enter any HTML content for the slide. The administrator interface implements the TinyMCE[44] WYSIWYG editor with django-tinymce[5] to allow the presenter to generate HTML by using a simple editor.

When pushed to Flux, this mote type simply re-renders all the HTML defined by the presenter.

Uses of this mote type include:

- As title/divider content - if Diffuse is being used as the primary form of presentation, title motes can be pushed to users.
- Instructive content - for displaying instructions to the attendees.
- For content which requires physical interaction. For example, a presenter might still want to encourage the raising of hands.
- Content shown only to a subset of attendees. The presenter might be interested in dividing the group into multiple subgroups, each with their own plan and access code. The presenter could then broadcast some informational content to a subgroup without letting the other groups see it.

3.5.2 Question and Answer

The Question and Answer mote type was created to show a basic, immediate response type of interaction and as a direct comparison to the questions content that standard ARS systems are primarily used for. It defines a 'Question' model derived from Mote, as well as an 'Answer' model with a many-to-one relation between answers and questions.

Creating a Question and Answer mote requires the following fields to be completed:

Question Text The question shown to the attendees.

Answers A list of answers with a text string, and an option to select whether the answer is considered 'correct' or not.

This mote type is presented on Flux as a question with a list of answers. Selected answers are highlighted and immediately pushed back to the distribution server.

Answers are shown to the presenter in the form of a column graph (using HighCharts) showing the number of attendees that responded with each answer. As new answers arrive, this graph updates in real-time with the new respondent's answer. The graph's scale changes automatically if numbers begin to move off the original scale. Should a user change their answer or disconnect, their previous answer is dynamically removed from the graph and replaced with their new one (if they submitted one).

Answers marked as correct are shown on the response graph as a green bar. Although this option is not used for any other feedback, it is used to demonstrate how attributes can affect how motes are displayed.

Uses of this mote type include:

- Prompting attendees for feedback from a set list of answers (Enjoyed, did not enjoy, etc.).
- Asking a question related to materials discussed in the presentation. Useful in a lecture for understanding how well students understand the material, allowing the presenter to rapidly know what they need to explain in more detail, etc.
- As an anonymous alternative to 'hands up' situations, where attendees are prompted to select some relatively private information (such as what category they fall into) without having to alert all other attendees.

3.5.3 Association

The Association type demonstrates a more 'puzzle' type of question, showing the potential flexibility available for creating feedback mechanisms. The type defines an 'Association-Group' 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 separately, prompting the user to

'join up' the left side with its original right side. This is done by dragging a line between the two columns. The line drawing effect was implemented using Raphael.js[4] for drawing lines between boxes, and making use of the jQuery UI for iPhone, iPad and iPod Touch[66] plugin to allow map jQuery's dragging events to their equivalent touch events (so that dragging would work correctly on touch devices).

To demonstrate possibilities for delayed response, the client only submits the user's answers once the user has attempted to match all associations. The mote type is shown to the presenter as a HighCharts pie chart, displaying segments for the number of correct answers submitted. For example, a '4' segment would be displayed showing the percentage of users who correctly matched four associations. This is dynamically populated with the number of associations possible. A question with 20 possible associations will automatically generate a graph with 20 possible segments.

The primary use of this mote type is to prompt users to match two sets of related data - countries to capitals, equations to correct answers, etc.

3.5.4 Heatmap

The Heatmap type was created to demonstrate interactions with different types of media, in this case, images. It defines a simple 'Heatmap' model, extending Mote which contains an attribute for an image, Django and PIL[49] (Python Imaging Library) handle the image upload and storage operations.

When pushed to Flux, the full image is shown to the client. On clicking/touching the image, a small marker shows where they have selected. The presenter sees the same image in the presenter interface with the addition of a heatmap overlay, showing areas with low click density in yellows and greens, up to those with higher click density in oranges and reds. The heatmap code was based off the work of Patrick Wied[69].

Uses of this mote type include:

- Showing a screenshot, graphic or photograph and prompting the users to point out a certain feature/error on the image.
- Displaying a map and prompting users to select where they live, giving the presenter an idea of the geographical distribution of the attendees.
- As a more complex Question/Answer type, by presenting a graphic with multiple options and asking attendees to touch the correct answer.

3.5.5 Feedback

Feedback motes demonstrate a more complex interaction between attendees and the presenter, making use of a text field to prompt for any string of text. It is defined by a simple 'Feedback' model which only defines a 'description' attribute.

When pushed to a client, the description attribute is displayed along with a text entry form. Whenever the text entry form receives input, this is immediately returned to the distribution server.

The presenter interface is a live-updating list of all text entry fields from users - the presenter will see every user typing in real-time on their display. If a user removes all text from their entry field, this entry is also completely removed from the presenter interface. To reduce network load, changes to the client entry field are only checked and returned every 250ms (through the use of the jQuery Throttle[56] snippet).

Uses of this mote type include:

- Prompting the attendees for any questions they may wish to ask. While not intended to replace asking out loud, presenters may prefer this option as they are given the option to browse a list of questions for ones they consider important. It also gives attendees who may not wish to speak in front of a crowd the option to ask questions.
- Prompting attendees for presentation feedback or comments.
- Asking for the answer to a question which may not have a defined set of answers, or where preset answers are not appropriate.

4 Problems

4.1 Communication protocol

Once the distribution server was communicating in both directions with the client, the issue of making communications trigger operations on each system arised. There were issues with figuring out a way to establish a light RPC[79] protocol, until inspiration came from the nodechat.js[57] project, which used an event based model, including an event string as part of the communication between systems.

Implementing this was simple, and it made the system much more flexible, however, the triggered events were not fully designed before implementing them, so it was often confusing to figure out what was triggering which events, when to respond to them, and how to 'bubble' them through all three systems when required. To resolve this, a naming convention was established 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 experienced 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, there were issues figuring out how to know when the distribution server was responding. The 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.

4.2 Dynamic resource loading

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

The first issue encountered with this was a race condition where the system would attempt to instantiate the mote's ClientMoteRenderer object before it had successfully loaded. A fix was attempted 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. Eventually this was fixed by using yepnope.js, a script loader which had built in callback functions which worked perfectly.

The second issue with this was with maintaining mote type modularity. Originally, all mote-specific resources for Flux were stored in a separate directory from the main mote type definition, this allowed the distribution server to be used to serve these resources. However, to meet flexibility objectives, it was desired that all mote-specific items be in a single folder, regardless of whether they were used for Diffuse or Flux. To achieve this, these resources were moved 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. An attempted work around was to request the JavaScript resource using JSONP[76], but this added extra unnecessary complexity. The final solution was to use Nginx to proxy requests for the resources on the Flux domain to the Diffuse resource folder.

4.3 JavaScript programming paradigms

While developing the JavaScript elements of the system, a number of problems arose when dealing with paradigms in the language that were very different from those in more traditional object-oriented languages. Dealing with these required detailed reading and research in to the language.

Initially, confusion arose with attempting to implement a standard 'class' structure for the Renderer. However, JavaScript is a class-free language, making use of a prototypal structure where methods are defined as part of function prototypes. Classes are implemented as functions, with their prototypes modified to add methods to them. The initial implementation attempt had methods as variables in the 'class' function which did not work correctly as inherited objects would refer to the same instance of the function. The solution was to implement methods as variables on the 'class' function prototype, which were correctly inherited.

This lead to a problem experienced with creating 'sub-classes'. Again, as there are no true classes, there are also no true subclasses or implementation of traditional inheritance. To support the required behaviour, it was necessary to use prototypal inheritance by first creating the new 'class' function and overwriting its prototype with that of the 'superclass'. This seemed to work, but whenever a method referred to an attribute specific to the subclass, the system would throw an error. This issue was caused by the overwriting of the new class's constructor when its superclass' prototype was used. To fix this, the original constructor was set back to the object's prototype constructor property.

Another confusion caused by the different JavaScript paradigms was the concept of binding. An issue arose when attempting to pass an object's method as a parameter to another function so that it could be used as a callback. The passed method would lose its binding to its object, and would be unable to reference any attributes on the object. This was due to the use of the 'this' keyword in the passed method - while 'this' would usually refer to the method's object, when it was passed as a parameter, the binding of 'this' was

changed to the object it was passed to. This issue held up development for a while until it was posted on Stack Overflow[38], where a response was received within 5 minutes and the problem fixed. The solution was to pass the function wrapped within an anonymous function, thereby invoking the method on the correct object instance and maintaing the binding of 'this'.

5 Testing

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

5.1 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 the system was built.
- 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[62] 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 so the workings of the system could be clearly seen as it was built. It was not intended to ensure the system met specifications.

5.2 Iteration Testing

At the beginning of every iteration, a brief outline testing plan was created for use to ensure the requirements of the iteration were met. If there was a reason to change requirements during implementation, this was noted and feedback from development went in to planning the next iteration. The testing plan was executed at the end of the iteration, with any failures added to the requirements for a future iteration.

Appendix E details how each iteration was specified and tested, some of the more important items include:

- If a requirement in iteration 1, 2 or 3 would cause these iterations to overrun, these requirements were pushed to later implementations so as not to hold up development of the core.
- The end of iteration 1 saw a failure in implementing the requirement to edit created content. This was due to complications with the polymorphic Mote structure which was not able to be completed within the time allotted to iteration 1. In order to keep development of the core on schedule, this requirement was moved to iteration 5.
- A number of requirements were changed during development. While this might be considered bad practice, the intent of choosing a flexible development model was so that the system could evolve in this way. A full discussion of this can be seen in section 6.4.

The iteration testing was considered a suitable way for continually testing an evolving project. Creating a test plan in each implementation also meant they could all be combined at the end of development to be ran as a single test plan with full requirement coverage.

5.3 Automated Stress Testing

As this system is built around many clients connected simultaneously, it was necessary to test how the system would behave under normal use. As only a few client systems at a time were available, a set of scripts were wrote 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.

In most cases, these automated tests would emulate a set of connected clients, following this behaviour:

1. Select a random client
2. Generate a random answer from those available
3. Return the answer
4. Wait 0.8 seconds
5. Repeat 100 times

This process was considered to be an example of high-end usage of the system. In all cases, the system performed well with no slowdowns, and no significant memory increases. Only informal, visual analysis was performed, no formal profiling.

5.4 Acceptance Testing

Acceptance testing was performed to ensure the system met the objectives of the project. This primarily consisted of having the author and a set of other users use the system on a number of devices, instructed to perform a few informal tests:

- Attempt to create a new plan and a set of motes in the presenter system, subscribe to the plan in the client system and push/respond to the mote.
- Attempt to 'break' the system by entering invalid inputs and using the system in abnormal ways.
- Using the client system on their own mobile devices to test how well it works.
- Editing/deleting motes from their plan.

6 Evaluation

The system was successful in improving on the three areas defined in the objectives:

6.1 Accessibility

6.1.1 Improving ease and cost of setup/configuration

By implementing the system entirely in software and supporting communication over existing networks, the project successfully improves on ease and cost of setup/configuration for system administrators. There is no additional configuration required for individual presentations, so, as desired, the system only requires configuring once.

However, the system is dependent upon a large number of tools and libraries. Configuring these as the author has done in another environment is likely to be complicated and error-prone. If the system were to be distributed, it would likely need to be reconfigured to reduce external dependencies and distributed as some form of 'package'. There is also a number of areas of the system where some configuration variables are hard-coded (such as domains); this would need to be addressed by separating all of these variables out to configuration files.

The communication of system services entirely over web-based services means there is no need for additional firewall configuration as long as HTTP traffic is allowed around the network. Support for WebSockets does not require any additional ports due to HTTP's UPGRADE method.

A potential shortcoming in system communication is between the presenter system and the distribution server. This requires a shared Redis instance accessible by both systems so that the presenter system can publish to the pub/sub channel. This would require either the presenter system, the distribution server and the Redis instance to be on the same machine (presenting potential scalability issues), or requiring additional firewall configurations to allow the Redis instance to be accessed over the network (presenting potential security issues). This could have been avoided by having the presenter system publish content to the distribution server over the established socket, or over HTTP.

6.1.2 Facilitating ease of access by presenters

As specified in the system objectives, there is no requirement to install additional software on individual systems. Presenters are able to access a URL on the network in a browser for access to the presenter system.

An issue with this approach is that the presenter system is not fully operational on all browsers. Due to time constraints, it was only tested in modern browsers such as Mozilla Firefox and Google Chrome and uses features of CSS3 and HTML5 which are not supported in many versions of Internet Explorer. Considering a number of businesses do not support

modern browsers, this could potentially be an issue for them. However, considering the focus of the project was the way in which these systems communicate, optimising for old browsers would have been time better spent elsewhere.

One shortcoming of this system is the requirement that presenters login to access their plans. While this is likely a sensible way to secure and protect their content, it is also an extra, potentially unnecessary step and is further complicated by the requirement that they have a separate Diffuse account. Ideally, it would have been possible to integrate the presenter's organisation's authentication system so that they are automatically logged in or use shared credentials, however, implementing a system which authenticates with all of the varying technologies present in organisations would have been complicated and would have reduced time spent on meeting important system objectives.

6.1.3 Supporting existing devices

The Flux web client, while optimised for iPad, has been tested and runs well on iPhone, Android phones and most modern browsers (again, Chrome and Firefox). While this does meet the requirement that it runs on existing mobile devices, it was not tested on a large number of the platforms initially researched (Windows Mobile, Windows Phone 7, BlackBerry OS and Symbian) due to the authors' limited access to devices using these platforms.

However, should these devices not function correctly, the system was designed to allow for modifications to the client without affecting the rest of the system due to the low coupling between mote type definitions, Flux, and the presenter system. Modifying individual mote type JavaScript to include feature detection or bug fixes for certain platforms would be relatively easy and would not require restarting the server.

The flexibility offered by allowing the developer full access to the JavaScript used by each mote type could be an issue as it is possible that they will implement features not supported by some user agents. For example, one of the demonstration mote types, Association, requires the browser to support SVG, which is not supported in Android devices. If the developer of this mote type did not create some form of fallback, this mote type would not work correctly on these devices.

The solution to this would be to create a more limited subset of fully tested features which developers could implement in their mote types through a defined API. Implementing this would have involved designing, developing and documenting this API, which would have been time consuming and could be considered outside the scope of the project. It would also mean significantly less flexibility afforded to developers creating mote types as they would only be able to use the functions provided by the API.

6.1.4 Removing dedicated hardware requirement

There is no dedicated hardware required for using Diffuse, meeting the objectives specified for the project. The two servers can be configured on existing hardware, communication is designed to work over the existing network, and clients do not require proprietary 'handsets'

to interact with the system; they are able to use their own mobile, portable or desktop devices.

Should an establishment not have an existing network infrastructure, there would be significant costs and work required to set one up in order to use Diffuse. It is likely that there would be no incentive to configure a network solely to use Diffuse, but for smaller establishments, it is conceivable that the Diffuse servers could be configured on a single desktop PC with a wireless router attached and still be highly functional.

There is also the issue of the prevalence of devices which would run the client. While in some establishments it is likely that the vast majority of attendees will have a compatible phone or laptop, in others those attendees might be in the minority. Supplying all other users with a suitable device may be more expensive to the presenter than purchasing dedicated ARS hardware. Considering the problems Diffuse attempts to solve, this issue is unfortunately unavoidable, however, it is hoped that this system will be attractive to those establishments where attendees are likely to own these devices and will value the improved interactivity provided by Diffuse.

6.1.5 Simplifying interaction experience

Interaction with the client system is improved in Diffuse over typical ARSs through the use of a graphical interface. Being able to click or touch items/answers directly without the user having to deal with the abstract concept of matching answer numbers/letters with a slide on a different screen, or pressing 'send' buttons to respond makes the system much more intuitive to use. Being able to use a full colour screen also allows additional instructions and guidance to be presented to the user where required.

Presenting all the information required on screen on the users' own device also makes this system valuable for disabled users, for example, blind users using an iPad with VoiceOver enabled will be able to have the question and answers read to them. This benefit is unfortunately limited to some mote types, as VoiceOver will not be as valuable to the blind user with images, or drag-and-drop puzzles. It is left up to the presenter when each type of content should be used, and it would theoretically be possible to design new mote types making heavy use of accessibility features for disabled users.

An issue with the use of a touch display is the lack of tactile feedback from screens; there is no physical response to pressing a button, so a user may have issues with knowing when they have correctly submitted a response. To alleviate this, visual cues such as highlighting are used to show a user what options they have selected.

The presenter interface is designed to be simple and easy-to-use. During informal testing, testers found it simple and were able to create and broadcast plans and motes without any instruction. Many existing ARSs integrated with PowerPoint to create content matched with slides - these often involve a plugin used for turning slides in to questions. This type of use might be more familiar to those used to PowerPoint, to whom Diffuse may be initially confusing. There are also likely to be those who do not like the idea of creating two sets of content, one for presentation and one for interaction.

6.2 Interaction

By leveraging the flexibility of the web browser found in targeted devices, Diffuse is able to offer a large variety of types of interaction. From simple Question and Answer interactions found in standard ARS systems, through complex puzzle-like association questions, to interactive images, as detailed in Section 3.5.

The options for developing additional interactions are potentially limitless, some possible examples include:

- A game (HTML5, Flash, etc.) which can be played by the user and have their performance information sent back to the presenter.
- Interactive content, such as a diagram or animation which can be rotated/manipulated.
- 'Fill in the blanks' style puzzles, displaying text or mathematical puzzles prompting users to enter the correct answers in the blanks on the display.
- Using the system to push content to large 'surface'-style displays to display content such as group activities, such as a jigsaw or puzzle for multiple users which returns the time it took each group to complete the puzzle to the presenter.
- Restaurant menus either displayed on tables or on surfaces, returning the user's order to the presenter (waiter).
- Making use of hardware features in some devices, for example, using the JavaScript rotation API on tablet/mobile devices that support it to detect which way the user is holding the device. Could be used as a virtual 'thumbs up/down'.
- Allowing multiple users to control some entity on the presenter interface. For example, the presenter interface could be projected on to a larger display and render a maze game containing a character for each connected user. Individual devices display a control pad to move their character, which updates their movement on the projected screen in real-time.
- Another game projected on to a larger screen which requires all users to vote on the next movement of the character.

6.3 Flexibility

Developing the system around a modular framework from the start meant the final version of Diffuse is very flexible and extendable. All final mote types were written after the main system was developed without any additional changes needed to the core system. This demonstrated that any developer with knowledge of the workings of the system could create almost any type of mote type they required.

Apart from the issues this flexibility causes with device compatibility detailed in section 6.1.3, there are also a few shortcomings with the implementation. Primarily, in order to set up a new mote type, the system administrator must run a few Django administration

commands in order to generate the correct database tables. While these commands are relatively simple and require very little learning, this is an awkward step which could be alleviated by creating some form of plugin framework which allowed the administrator to enable mote types from the administration interface.

6.4 The Development Process

Although the development model chosen for the project was iterative development, it was felt that the actual development process was significantly less structured than typically imposed by the model for a number of reasons.

Primarily, there were no full system requirements set out at the start of the project, only at the start of each iteration, meaning there was never a full set of requirements for the system until the last iteration. While this does not mean the objectives of the system were not met, it does mean requirements were never entirely clear. This was not considered detrimental to the system, but had it been developed in a team or for a client, it would likely have been unsuitable.

Requirements were also often changed during iterations; this suggests requirements were not being used for their intended purpose as an outline of the structure of the system, but instead as 'hints' for the direction of each iteration. Again, it is not thought that this caused any negative effects on the outcome of each iteration, but if it weren't controlled, it could have been potentially harmful to the project schedule if the requirements were constantly changing.

The intent of choosing a structured development model was to create some organisation when combined with the 'hacking' style of programming so that design and requirements aren't entirely created while coding. Although the model did create some organisation (as is evident by the existence of requirement/testing plans), it was felt that not enough emphasis was placed on the planning and testing elements, so the actual development model could be better described as 'hacking' combined with a few reluctant breaks required to write the next set of requirements.

It is thought that the reason the development model was not used to its full potential was to do with the author's reluctance and difficulty in writing requirements on paper when they were already fully mentally planned. The desire to write code and experiment with technology to see how best to implement the project was generally stronger than the desire to plan everything beforehand.

Despite these issues, it is considered that the process adopted was largely beneficial to the outcome of the project:

- Flexible requirements allowed the system to evolve in directions other than that specified. Had the requirements been followed to the letter, there were a few cases where the design would have meant an awkward or unsuitable implementation, causing issues in later iterations.

- The lack of a rigid design at the start of development meant there was freedom to experiment with technology, tools and libraries to find the most suitable way to implement requirements without the restrictions of a design established before the author had any experience with the tools available. However, adding a more structured prototyping phase to the development cycle would likely have had the same benefits.
- Focusing on code made the system very enjoyable to develop, the chance to experiment with the technologies used significantly increased the author's desire to continue development on the system. While this is purely personal and the results speculative, if a large portion of this time was dedicated to writing requirement specifications and testing documentation, it is likely that morale and incentive would have dropped, reducing the overall quality of the project.

Although the project did not suffer from the loose interpretation of the model used, had all the variables not been as they were, it is predicted that the outcome might have been significantly different, for example:

- If this project was intended to be developed in a team, there would be a very large gap in the understanding of requirements between team members leading to disorganised and chaotic development.
- If the author was not able to develop each separate system with relatively few problems. Had a problem occurred and there were no requirements specified, the final system would be missing components with no documents detailing how they were intended to be structured.
- If the author had no self-control over requirement changes, causing requirements to continually be changing, never reaching the final state of the system.
- If there had been an issue which caused development to be off schedule, recovering would have been complicated by the lack of structure.

Overall, while development did not follow the planned process model as strictly as it should have, a combination of luck and minor benefits from the flexible process adopted still managed to result in a complete system which successfully met all of the defined objectives.

6.5 Testing

Testing is considered to be one of the major failing points of this project as there was generally very little testing done, and what was done was informal and not entirely comprehensive.

While the testing strategies adopted and described in section 5 would be sufficient if they were used to their full extent, it was felt that there were a number of shortcomings in how each strategy was executed.

The concept behind using the iteration testing strategy was that each iteration would be a self-contained unit with defined requirements and a full-testing plan for each. In reality, each iteration was not as self-contained as it should have been and the loose interpretation

of the planning process (discussed in section 6.4) caused listed requirements to be short and insufficient. These issues meant the testing plan was very brief, usually consisting of about five tests per iteration.

Those tests that were documented were very broad with results that would be difficult to quantify. There was also no attempt made to test edge cases in the majority of situations. On the positive side, the tests that did exist served to show what was actually achieved in each iteration.

Stress testing was used to ensure the system could cope with high loads - while in this regard, it served its purpose, it could have been formalised and extended in a number of ways:

- Creating more dynamic scripts for all types of content and for all potential motes, rather than hardcoding particular motes. Allowing for testing more scenarios.
- Detailed formal profiling; taking records of latency, time taken for the presenter to update, memory increase in all systems, etc. so that an analysis could be performed on the affect of changing various variables on the performance of the system.
- Using automation to test other parts of the system using tools such as Selenium[47].

There were also a number of more comprehensive tests planned but never executed due to time and resource constraints. As discussed in section 6.1.3, testing the client on multiple platforms was not possible due to limited access to devices of various types, had there been a larger time period allotted in the original plan for a testing phase, it might have been possible to source devices for testing.

While the acceptance testing performed did help to test the suitability of the system, it would also have been beneficial to test the system in a real-world situation, possibly by asking a lecturer at Aberystwyth University to try the software in a lecture, or by arranging a larger scale demonstration. This would have provided an opportunity to see how well the system works in the real world, as well as receive feedback from users on their experience with it. It would have been particularly useful to demonstrate the system to lecturers with experience using existing ARS systems to see their feedback.

As it was, the system was only tested with at most five simultaneous real users, although these users did provide useful informal feedback on their experiences with the system, with their overall opinions suggesting that:

- Both the presenter and client system were easy and straightforward to use.
- They would like to see the system in use in lectures.
- Use of the system would make them feel more involved in lectures.
- That the flexibility of the system made it particularly interesting. Most users after using the system would suggest a new mote type they think would be interesting, or an environment in which they thought the system would be useful. A very common suggestion was the potential for use in primary schools.

If this project was started again, a number of things would be changed in regards to testing:

- Dependent on a more strict following of the development model, more detailed requirements would be written along with significantly more detailed test plans for each requirement when using iteration testing.
- Heavier and more detailed use of automated testing.
- Making use of unit testing for all systems to ensure code meets requirements.

6.6 Future Developments

The basic broadcast-respond communication model implemented by the project is very open-ended, providing many opportunities for future development of the project, both through the development of new mote types and through adding new features to the main system.

The flexible mote framework used provides limitless possible extensions to the system (as discussed in section 6.2) potentially allowing the system to work in ways beyond the planned uses in this project; including retail, restaurants and primary education.

From testing with real users, a common point of feedback was their wish to see the ability to save user responses and 'replay' how the answers arrived at a later time. As all responses are already stored in the key-value store, implementing this would not be complicated, involving dumping all the response JSON to a file/database. The methods which populate mote type rendering on the presenter interface already deals with 'replaying' a set of answers, so reloading them would not require much additional implementation. Being able to save users' responses would be useful in situations where a presenter is using the feedback as a long-term data set, seeing how feedback improves over time, or where they wish to compare two sets of feedback.

A more complex evolution of the system would involve supporting peer-to-peer interactions; for example, allowing a 'chat' mote type to be displayed where users on the same plan could chat with each other, or a collaborative drawing interface where users could work together on a drawing across multiple devices. This change in infrastructure would also support targeted mote pushing, so the presenter could push a different mote to each user or even to groups of users, widening the uses of the system even further, such as to co-ordinate group activities or to push order information to diners at a particular table, etc.

6.7 Starting Again

If this system were to be started from scratch, there are a number of things that would be changed to streamline development and improve the final product:

- Follow the development model more rigidly (see section 6.4).
- Significantly improve testing methods and coverage (see section 6.5).

- Develop with a particular client in mind, perhaps by involving a lecturer/teacher interested in the use of ARS software.
- Improve testing plan to include testing for more mobile devices and to test in a real-world situation.
- Demonstrate the system to users earlier on in development so that feedback could be incorporated in to the system.
- Create mote types which would better represent how the system could be used in other situations.

6.8 Final Words

Overall, it is felt that the implementation of the project meets the initial objectives very well. The final product is a system which was very enjoyable and interesting to build and one that could not only theoretically be used for its intended purpose as it is, but could be used in situations way beyond those which the project initially targeted. Even by the end of development, after some research it still seems that the final system is a unique product in its goals and implementation.

Unfortunately, while the development process was successful, some of the secondary parts of the project were considered inadequate, particularly in regards to following a process model and testing. Although these did not affect the final system, as with any good software product, the project would have benefited from better documentation and more comprehensive testing.

References

[1] Moe Aboulkheir. Pyflakes. `http://pypi.python.org/pypi/pyflakes`.

Pyflakes was used with the pyflakes.vim plugin to detect errors in Python source while developing.

[2] Anderson. django-annoying on BitBucket. `https://bitbucket.org/offline/django-annoying`.

This utility library was used to simplify some Django development.

[3] Apple. iOS technology overview. `http://developer.apple.com/technologies/ios/`.

Technical details about iOS - used to confirm development support

[4] Dmitry Baranosvskiy. Raphael - JavaScript Library. `http://raphaeljs.com/`.

Raphael is a JavaScript library used for creating SVG graphics. This was used in the Association mote type to draw lines between options.

[5] Joost Cassee. django-tinymce. `http://code.google.com/p/django-tinymce/`.

django-tinymce was used to implement TinyMCE as a widget for editing the Slide mote type.

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

A Python WSGI server used to serve the Django application.

[7] Chris Messina and Blain Cook. OAuth. `http://oauth.net/`.

Used for research in to potential implementation of OAuth for logging in users.

[8] Francis Cianfrocca. EventMachine. `http://rubyeventmachine.com/`.

EventMachine was another consideration for implementing the distribution server. This site has full documentation on its use which was used in deciding whether it should be used.

[9] Wincent Colaiuta. Command-T. `https://wincent.com/products/command-t`.

This Vim plugin was used for quickly navigating the projects' file structure while developing.

[10] Bert Constantin. Polymorphic Models for Django. `http://bserve.webhop.org/django_polymorphic/`.

The structure of mote types depended on the use of django-polymorphic to support specialisation of the Mote model when required.

[11] 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.

[12] 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.

[13] Ryan Dahl. Video: Node.js by Ryan Dahl. `http://jsconf.eu/2009/video_nodejs_by_ryan_dahl.html`.

This presentation helped explain the effects of blocking I/O, and how it is solved in evented systems such as Node.js.

[14] dbr. Simple "Long Polling" example code? `http://stackoverflow.com/questions/333664/simple-long-polling-example-code/333884#333884`.

This answer detailed the requirements for implementing a long polling style of communication.

[15] Ecole Polytechnique Federale de Lausanne (EPFL). The Scala Programming Language. `http://www.scala-lang.org/`.

As with Erlang, Scala was also briefly considered and researched using this site.

[16] Douglas Crockford. Introducing JSON. `http://www.json.org/`.

JSON was used as the data interchange format in communication between the three parts of the system. This page was used as a reference for the structure of the JSON format.

[17] erlang.org. Erlang Programming Language. `http://www.erlang.org/`.

When deciding what language should be used for implementing the distribution server, Erlang was briefly considered and researched using this site.

[18] 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.

[19] 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.

[20] Symbian Foundation. Symbian foundation website. `http://symbian.org/`.

Provided details on the history and development for Symbian OS

[21] Google. Android developers. `http://developer.android.com/index.html`.

Details the technology and development APIs for Android

[22] Ian Hickson. The WebSocket API. `http://dev.w3.org/html5/websockets/`.

The WebSocket API documentation was used for researching potential implementation requirements.

[23] Rob Hudson. django-debug-toolbar on GitHub. `https://github.com/robhudson/django-debug-toolbar`.

This utility library aided in developing and debugging the presenter system. It provides debug information panels on rendered views containing headers, available variables and the context used.

[24] Research in Motion. UK blackberry website. `http://uk.blackberry.com/`.

Used to find details on recent BlackBerry phones and the OS

[25] Gartner Inc. Gartner Says Worldwide Mobile Phone Sales Grew 17 Per Cent in First Quarter 2010, May 2010.

Used to find data on worldwide mobile operating system usage.

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

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

[27] Adobe Systems Incorporated. flash.net.Socket - ActionScript 3.0 Reference. `http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/net/Socket.html`.

The Flash Socket documentation was used to research the potential requirements for implementing Flash Sockets.

[28] Internet2. Shibboleth. `http://shibboleth.internet2.edu/`.

Used for research in to potential implementation of Shibboleth for logging in users.

[29] 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.

[30] 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/

[31] Steve Krug. *Don't Make Me Think Second Ediiton.* New Riders, 2005.

Lots of interesting information about usability which was used for improving both the Diffuse and Flux interfaces.

[32] Sencha Labs. Connect. `http://senchalabs.github.com/connect/`.

Connect was used to assist with creating the HTTP component of the distribution server, serving static files and dealing with sessions.

[33] Twisted Matrix Labs. Twisted. `http://twistedmatrix.com/trac/`.

Twisted was considered as an option for the distribution server - this site was used to research its use and implementation.

[34] 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.

[35] LLC. Linode. Linode. `http://www.linode.com/`.

A VPS provided by Linode was used as the development platform for the site. I had past experience with hosting with Linode and was therefore comfortable developing on a server provided by them.

[36] Canonical Ltd. Ubuntu. `http://www.ubuntu.com/`.

Ubuntu was the primary operating system used for developing the system.

[37] 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.

[38] Matt. Javascript: Creating a persistently bound function. `http://stackoverflow.com/questions/4967689/javascript-creating-a-persistently-bound-function/4967732#4967732`.

A problem with the binding of the 'this' keyword when passing functions as parameters hindered development for a while. This solution was posted to my question on Stack Overflow which resolved the issue.

[39] 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.

[40] 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.

[41] Microsoft. MSDN windows mobile. `http://msdn.microsoft.com/en-us/library/bb847935.aspx`.

Used for information on developing for Windows Mobile

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

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

[43] Research In Motion. Blackberry 6: Inside the new blackberry browser, August 2010.

Details the features of the BlackBerry 6 browser, including how it is built on WebKit

[44] MoxieCode. TinyMCE. `http://tinymce.moxiecode.com/`.

TinyMCE is a JavaScript WYSIWYG editor used for editing HTML content in web pages. This was used for the administration interface for the Slide type.

[45] Nokia. Nokia develops a new browser for series 60 by using open source software, June 2005.

Details on how Nokia's Series 60 browser uses WebKit's WebCore and JavaScriptCore

[46] Norman Walsh. A Technical Introduction to XML. `http://www.xml.com/pub/a/98/10/guide0.html`.

XML was briefly considered as the data interchange format for the system. This page was used to find more information on its structure and implementation.

[47] OpenQA. About Selenium. `http://seleniumhq.org/about/`.

An automated GUI testing tool considered for automated testing.

[48] Python Software Foundation. pickle - Python object serialization. `http://docs.python.org/library/pickle.html`.

Pickling was an alternative form of serialisation considered for data interchange. This Python documentation page was used for a reference on its implementation.

[49] Pythonware. Python Image Library (PIL). `http://www.pythonware.com/products/pil/`.

PIL was used with Django's built-in ImageField support for storing images for the HeatMap data type.

[50] Qwizdom. UK qwizdom website. `http://www.qwizdom.co.uk/`.

Used to research details on the Qwizdom ARS system

[51] 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.

[52] 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.

[53] Sammy The Walrus IV. Business Insider - Steve Jobs Wasn't Lying: Apple's iPad Market Share Was Really More Than 90% Last Year. `http://www.businessinsider.com/ipad-share-2011-3`.

For additional information on the current iPad market share.

[54] 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.

[55] 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.

[56] Remy Sharp. Throttling function calls. `http://remysharp.com/2010/07/21/throttling-function-calls/`.

This snippet was used to throttle checking the input field for the Feedback mote type to prevent network traffic on every key press.

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

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

[58] 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.

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

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

[60] The Apache Software Foundation. Apache HTTP Server Project. http://httpd.apache.org/.

Apache was researched as a potential distribution server candidate, but was considered unsuitable.

[61] The jQuery Project. Manipulation - jQuery API. http://api.jquery.com/category/manipulation/.

DOM manipulation was used heavily for both the presenter and client interface. This jQuery API page was used to reference functions available for modifying the DOM.

[62] 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.

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

The project was versioned using the Git version control system.

[64] TurningPoint. Turnintpoint responseware product site. http://www.turningtechnologies.com/studentresponsesystems/mobiledistancelearning/highere

Used for details on the ResponseWare web-based ARS solution

[65] IML UK. IML UK website. http://www.iml.co.uk/.

Used to research details on the ARS systems provided by IML

[66] Stephen von Takach. jQuery UI for iPhone, iPad and iPod Touch. http://code.google.com/p/jquery-ui-for-ipad-and-iphone/.

This plugin was used for the Association mote type, enabling a mapping between jQuery click/drag events to their equivalent touch events.

[67] 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.

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

Provides details on support for HTML5 and JavaScript technologies in WebKit

[69] Patrick Wied. Heatmap.js. http://www.patrick-wied.at/static/heatmapjs/.

Now a fully-fledged plugin, parts of heatmap.js code was used to generate heatmaps for the Heatmap mote type. At the time of development, this was not a finalised plugin.

[70] Wikipedia. Asynchronous I/O. http://en.wikipedia.org/wiki/Asynchronous_I/O.

When researching the effects of blocking in a real-time server, this page was used for information on solutions for asynchronous I/O.

[71] Wikipedia. Comet (programming). `http://en.wikipedia.org/wiki/Comet_(programming)`.

Used for additional research in to the Comet web application model.

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

Used for inspiration for project name.

[73] 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.

[74] Wikipedia. Entity-attribute-value model. `http://en.wikipedia.org/wiki/Entity-attribute-value_model`.

An alternative way of structuring mote types, using an EAV model was researched. This paged helped with understanding how they work and are constructed.

[75] Wikipedia. Iterative and incremental development. `http://en.wikipedia.org/wiki/Iterative_and_incremental_development`.

Used for researching the iterative development model.

[76] Wikipedia. JSONP. `http://en.wikipedia.org/wiki/JSONP`.

JSONP was researched to solve the dynamic resource loading issues.

[77] Wikipedia. Polymorphism in object-oriented programming. `http://en.wikipedia.org/wiki/Polymorphism_in_object-oriented_programming`.

The structure of the Mote design is built around the concept of polymorphism; this page was used to clarify the structure of a polymorphic design.

[78] Wikipedia. Publish/subscribe. `http://en.wikipedia.org/wiki/Publish/subscribe`.

Used for information on the publish/subscribe pattern used in Redis.

[79] Wikipedia. Remote procedure call. `http://en.wikipedia.org/wiki/Remote_procedure_call`.

Used for researching potential implementations of a light RPC protocol.

[80] Wikipedia. Waterfall model. `http://en.wikipedia.org/wiki/Waterfall_model`.

Used for researching the waterfall development model.

[81] Wikipedia. XMLHttpRequest. `http://en.wikipedia.org/wiki/XMLHttpRequest`.

Used for researching the use of XMLHttpRequest for implementation of multipart streaming.

[82] 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

[83] Genee World. Genee world website. `http://www.geneeworld.com/`.

Used to research details on the ARS systems provided by Genee World

Appendices

A Project schedule

Task	Description	Days
Iteration 0	Initial Planning & Configuration	26-28th Nov
Produce project requirements	Create a document expanding on this list with project requirements, break them down in to iterations	26-28th Nov
Produce test plan	Create a test plan for each iteration to test whether requirements for that iteration have been met. Include integration tests and test for the complete system.	26-28th Nov
Configure server	Install Nginx, Node.js, MySQL and Redis on VPS and configure as required	28th Nov
Iteration 1	Presenter System	29th Nov - 10th Dec
Content models	Create definitions for content models in the system	30th Nov - 1st Dec
Administration interface	Develop basic interface for managing and creating content	2-6th Dec
Serialisation system	Create system for serialising content objects in to JSON for transfer to the distribution server	7th Dec
Authorisation system	Develop system for authorising presenter users	8th Dec
Pushing content	Create interface with Redis pub/sub channel for pushing content as required	9th Dec
Testing	Execute test plan, produce updated documents as required	10th Dec

Iteration 2	Distribution Server	11 - 18th Dec
Redis interface	Configure interface to Redis pub/sub channel for receiving updates	11th Dec
Caching system	Use Redis for caching content to be broadcast	11-13th Dec
Broadcast configuration	Use socket.io for broadcasting content to the required channels	14-15th Dec
Content server	Enable client to connect to server for accessing resources for thick client	16-17th Dec
Testing	Execute test plan, produce updated documents as required	18th Dec
Iteration 3	Client System	19 - 30th Dec
Authorisation	Enable users to authorise themselves to receive content	19th Dec
Real-time connection	Use socket.io to listen to content from distribution server	20th Dec
Templating system	Create system for parsing the JSON, fetching a template from the server and displaying the content as required	21-23rd Dec
History	Create interface for navigating broadcast history on the client	27-29th Dec
Testing	Execute test plan, produce updated documents as required	30th Dec
Iteration 4	Interactivity	1 - 13th Jan
Specify interactivity	Allow templates to specify how a user can interact with them	1-3rd Jan
Serialise interactions	Create module for serialising interactions to JSON for returning to the server	4-5th Jan
Distribute responses	Enable distribution server to return responses to presenter	6-9th Jan
Parse responses	Enable presenter system to parse and store responses from content	10-12th Jan
Testing	Execute test plan, produce updated documents as required	13th Jan

Iteration 5	Additional Features	14th Jan - 8th Feb
Add content types	Create new content types for broadcast	14-17th Jan
Examination Period		17th-29th Jan
Improved presenter interface	Create Ajax-style interactions to improve content creation process	30th Jan - 2nd Feb
Device independent client	Abstract device-specific code (CSS) on client away	3rd Feb
Create iPad style	Develop CSS for laying out content on an iPad display	4-7th Feb
Testing	Execute test plan, produce updated documents as required	8th Feb
Iteration 6	**Polish System**	**9 - 18th Feb**
Support other devices	Create additional CSS files to demo support on other devices	9-11th Feb
Clean presenter interface	Tidy up presenter UI for ease of use	12-13th Feb
Clean client interface	Tidy up client UI to ensure content is easy to see and interact with	14-15th Feb
Add more content types	Add a content type which could be used in a different situation, such as a business meeting, for demonstration purposes	16-17th Feb
Testing	Execute test plan, produce updated documents as required	18th Feb
Project Completion	**Test, evaluate and write report**	**19th Feb - 5th May**
Testing	Thoroughly test system to test specification	19-22nd Feb
Write dissertation	Write dissertation and supporting material	23rd Feb - 5th May

B Report on operating systems

While the goal of my project is to develop a device/OS agnostic system, all the devices supported will need to be able to facilitate running the features required in an ARS with rich content support. This means they all need:

- A way to connect to a network, either over a cable, wi-fi or data connection (3G).
- A reasonably sized colour screen - for the purposes of displaying rich content, an initial suggestion of a minimum resolution of 480 x 320 seems reasonable.
- A flexible input mechanism; either touch or a pointing device and keyboard.
- Some way of executing code/applications.

I will take a closer look at common platforms, devices and operating systems which meet these requirements in order to find a common set of features on which to build the project. Ideally, the system should use a minimal amount of device-specific code and should any be required, it should be easily changed without needing to re-deploy to client devices.

B.1 Mobile Operating Systems

First, I look at mobile operating systems. To determine what were the most popular mobile OS', I used an analysis of the smartphone market published in May 2010 by Gartner Inc.[25]:

Company	Market Share 2010 Q1(%)
Symbian	44.3
BlackBerry OS	19.4
iPhone OS	15.4
Android	9.6
Microsoft Windows Mobile	6.8
Linux	3.7
Other OSs	0.7

I will take a more detailed look at all listed OSs with a market share of 5% or larger.

Symbian

Symbian[20], the most popular mobile operating system has gone through many iterations since its original release as Symbian OS 6.0 in 2001 and is now found in all Nokia phones, as well as various mobile devices from Sony Ericsson, Sharp and Samsung.

The majority of Symbian devices are basic mobile phones which do not meet the necessary specifications (particularly a lack of a 3G data connection and large screen), but newer Nokia devices running Symbian OS 9.1 and later seem to be suitable, so this is the version of the operating system which I would likely need to target.

Software can be written for Symbian in various languages, primarily C++ with Qt but including Python, Java ME, Ruby and .NET. Developing a UI however, would

need to use Symbian specific APIs, something which I would like to avoid. Device security is also left up to the vendor, meaning most devices can not run custom code, or only code approved by the vendor (such as through Nokia's Ovi store).

While various browsers are available for Symbian devices, they typically use the built-in browser supplied by the vendor. In Nokia's case, the Nokia Browser in Series 60 devices and above is built upon Apple's WebKit project[45], particularly the WebCore and JavaScriptCore components - suggesting that the majority of newer Symbian devices use relatively modern browser components, including support for JavaScript. Some later versions of Symbian on Nokia devices can make use of Flash, Silverlight and JavaFX.

BlackBerry OS

RIM develops the BlackBerry[24] line of business smartphone devices, all of which run on BlackBerry OS. Most recent BlackBerry devices meet all the specifications listed above.

BlackBerry OS applications are typically written in Java using a set of APIs to interface with the operating system - this suggests a lot of device specific-code, so this approach is unlikely to be suitable.

As of BlackBerry OS 6, the bundled browser is also based on WebKit[43], and therefore supports modern web features and JavaScript. Later versions of BlackBerry OS also support Flash content, with plans to support Silverlight

iPhone OS

iPhone OS[3], now known as iOS is Apple's mobile operating system which runs on all mobile devices built by Apple; iPod touch, iPhone & iPad.

Applications are built using the iOS SDK and written in Objective-C. They can only be distributed through the Apple Store, which requires approval from Apple and a fee to be a member of the iOS Developer Program. Again, as this requires device independent code in an entirely different language from the OS' I've researched so far, it doesn't seem like a suitable solution.

The browser on iOS is a version of Apple's Safari, also based on WebKit (Apple being the founder of and contributor to the project). iOS does not allow any third-party modifications to the operating system and built-in applications, so browser extensions such as Flash, JavaFX and Silverlight can not be used.

Android

Android[21] is Google's open source mobile operating system licensed under the Apache License, allowing vendors to freely use and extend the OS, and as a result, it has been widely adopted by multiple manufacturers since its initial release.

Applications are typically written using Java and the Android SDK, although due to the open nature of the operating system, most languages can be used. Applications can either be distributed through a store, such as the Android Store bundled with the majority of phones, through a vendor-specific store, or transferred directly to

the phone. While the flexibility of this platform could mean only a thin UI layer would need to be written for it, it would still require quite a significant amount of device-specific code to do so.

The bundled browser on Android devices is also based on WebKit, using Google's V8 engine for fast JavaScript execution. The browser can also be extended to use many web application platforms including Flash, Silverlight and potentially JavaFX (although there is no implementation currently available).

Windows Mobile

Microsoft's mobile OS, Windows Mobile[41] has been used on various mobile devices built by a number of the world's largest mobile manufacturers. It is now being phased out in favour of the recently released Windows Phone 7.

Developing for Windows Mobile requires the use of Visual C++, or code on the .NET Compact Framework - more device independent code.

The Windows Mobile browser has had seen many iterations, although they are all based on Internet Explorer and therefore support most modern features and JavaScript. While JavaFX and Silverlight are supported on later versions of the platform, Flash support is not available.

Note regarding Windows Phone 7: *For the purpose of this assignment, as it was released less than two weeks before this report was written, market share figures are not yet available, and based on initial reviews of the operating system, we will assume that Windows Phone 7 has a significant enough market share to be considered, and that it has feature parity with iOS, namely restrictions on custom applications and a modern browser.*

Other

There are a number of mobile operating systems which do not currently have a market share above 5%, either because they have not yet been widely adopted, or because they have only recently been released. These include Nokia's Maemo (based on Debian), Blackberry's upcoming QNX-based Blackberry Tablet OS, and Palm's webOS. While I will not specifically research the applicability of these devices, I am confident that as the majority are either Unix-based, or released within the past two years, they all have a relatively modern web browser.

B.2 PC Operating Systems

Ensuring support for popular personal computer operating systems should not be as significant a task as supporting mobile operating systems. New innovations and technologies are almost always first implemented on PC OS' before eventually making their way to mobile devices, a transition which can take many years due to limited or differing hardware and alternative UI paradigms used on mobiles. PC OS' also tend not to be as restrictive as to what software can be compiled on them, usually down to compilers being available for most popular processor architectures.

For this reason; I will consider the three largest PC operating systems to be suitable for running this system, those three being Windows, Mac OS and Linux. Although all three come in various versions and incarnations, we can assume that given sufficient hardware, they can run any required code.

C Implementation options and choices

C.1 Client System

Research in to mobile platforms indicate that there is very little similarity between development solutions for the majority of platforms. I will briefly look at my options for the client application and detail the option I have chosen.

C.1.1 Common Library

Although the majority of the systems I have studied require development in various languages and using device-specific APIs, it is likely that I could build a library in a low-level language such as C++ and create thin UI layers for each device. While this means I could take advantage of the benefits of each platform, it does have a number of disadvantages: /beginitemize /item Time required to create and maintain clients for each platform is prohibitive. It pushes the boundaries of the 'multi-platform' philosophy too far. /item I am not proficient in C++ or C, learning these languages to the level required for implementation of these network protocols would take a lot of time. /item This system uses many high-level ideas not suited for C/C++. /enditemize

C.1.2 Cross-Platform Languages

Many devices that I studied support Java. As Java code runs in a virtual machine, there is a significant 'write once run anywhere' benefit with using Java. There would still be a requirement to write device-specific code to interface with their layout APIs. I have used Java extensively, so I would feel comfortable writing an application using it, but I would still need to learn a lot about mobile operating system APIs in order to ensure the system is flexible enough. Java is also not supported on iOS, one of the largest mobile operating systems and the one I use most myself.

C.1.3 Rich Web Multimedia Platforms

Rich web-based platforms such as Flash, Siverlight and JavaFX have implementations on multiple devices. With these platforms, everything from the operational code to the interface are entirely interpreted by the platform. This would mean that writing, for example, a Flash-based solution once, would run on every platform which supports Flash without any additional code. Although I feel the idea behind these systems is good, I don't personally believe in requiring proprietary extensions to run the application and I can not find one of these platforms which is available across all platforms I have studied.

C.1.4 Browser-Based

Browsers are available on the vast majority of user agents in variations incarnations - in my research I noted that every platform I studied had a proficient browser, most being based on the WebKit[68] 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.

C.1.5 Decision

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 retrieve content.

D Existing ARS systems

A quick Google search for 'Audience Response System' quickly reveals the extent of the ARS market, with many companies producing their own specific hardware and software implementations. While the majority of products available are similar, I will take a more detailed look at a few.

D.1 Qwizdom

Qwizdom[50] is the ARS system used by the Computer Science department at Aberystwyth University and is therefore the only one with which I have any direct experience. They provide two hardware solutions for learners:

- A keypad with a small E-ink display which shows the currently selected answer (the units currently used by Aberystwyth University are older versions of these units with an LCD display).
- A keypad with a larger LCD display with the ability to display the current question and answers.

These interact via RF to a unit attached to a computer. The computer needs to be running ActionPoint, a PowerPoint/Keynote plugin which instructors can use to integrate questions in to their slideshows. When an ActionPoint slide is being displayed during a lecture, students' can submit answers to the question which is then immediately updated in ActionPoint. The lecturer can choose to display a graph of responses which can be reacted on immediately, or saved for reporting.

Qwizdom also supplies additional hardware for instructors to interact with their slides, including a tablet-style device.

This keypad style of ARS seems to be the most basic, tried and tested implementation, with other companies providing similar solutions to Qwizdom include, KEEPad, ShowMode, Votech, Group Dynamics and PowerCom, all of which do not offer anything beyond this type of interaction.

The issues with this keypad-style are what I noted in the introduction of this report; limited displays and the requirement for dedicated hardware. The integration with PowerPoint seems to be a good way of encouraging the use of these devices as a supplement to existing slideshow-based teaching, but I feel it makes the interactivity too oriented around slides where they could easily exist as separate entities if the device could display more information.

D.2 IML, Genee World

IML[65] and Genee World[83] are companies which both provide standard keypad based ARS', but have additional systems which are more in line with what I am trying to achieve

with my project.

"IML enotes" is a product focused on group meetings; allowing users on laptops to type feedback on a discussion topic and submit them anonymously over wi-fi to a central computer where they can be collated. The use of laptops is interesting but it appears to require specific software, laptops rented from IML, and an 'IML producer' to be present during the meeting. All of which seem very restrictive.

Genee World provide their "Virtual G Pad" solution; a web-based interface which appears to be emulating a keypad, interacting with the same server software (ClassComm) as their hardware based systems. The system can be accessed from modern browsers and should therefore run on tablets, phones and laptops, although the former two are not mentioned anywhere in the product description. As this system can run alongside their hardware, it is limited in the same ways.

The use of a web-based interface for the "Virtual G Pad" seems like an excellent way to ensure cross-device and multi-platform compatibility, definitely something to consider when planning my system.

D.3 TurningPoint ResponseWare

Again, TurningPoint[64] offer hardware keypad solutions, but now have their "ResponseWare Web" solution. Similar to the "Virtual G Pad", this is primarily a web-based interface but it seems device specific applications for iPhone, BlackBerry and Windows Mobile devices are available. The website does not mention whether these applications are required to use this system on these devices or whether the web-based interface will adjust to be used on a touch-based device. There is also no mention of tablet devices or the Android mobile operating system.

The system runs over Wi-Fi or a mobile data connection - as it is likely that the majority of educational establishments to which this system is targeted are likely to have an established wireless infrastructure there would be no additional hardware or configuration required. This method of communication seems appropriate for the hardware independent solution I am working towards.

The main notable limitation in the ResponseWare system are the limited content types which can be broadcast, restricted only to simple Q&A style content. In an improved system, the larger, more detailed screens on modern devices could be used to display a variety of rich content.

D.4 Summary

From researching existing ARS solutions, I have noted how these systems have evolved from simple keypads, through to web-based clients which can be used on modern mobile phones. Despite the software and hardware evolving, the companies I researched have not extended their solutions beyond basic Q&A interactions, possibly due to the need to support legacy

devices in their software. To create an ARS system which sets itself apart from the Virtual G Pad or ResponseWare, I will need to support additional types of rich content, such as images, websites, and interactive activities.

Existing solutions have also provided me with some suggestions as to how to build my implementation; particularly the use of a web-based interface rather than an application for every device I want to support, and the use of Wi-Fi/Data over HTTP for communication with the server.

E Testing documentation

During development, each iteration began with writing requirements and a testing plan for that iteration. After it was complete, any changes to the requirements during implementation were noted and the testing plan was executed. The following contains the documentation written for each iteration, restructured to be more concise and consistent, changes from the original notes are:

- The testing plan tables include results, this column was not completed until the end of the iteration.
- Tests where there were no results due to requirement changes during implementation have been removed from the test tables.

E.1 Iteration 1 - Presenter System

E.1.1 Requirements

R1.1 Create new Plans.

R1.2 Create new Motes within a Plan.

R1.3 Delete Plans/Motes.

R1.4 Edit Plans/Motes.

R1.5 Output JSON for Motes.

R1.6 Authorise presenter users with establishments own authentication system.

R1.7 Ensure only presenter who created Plan can access/edit it.

R1.8 Push content to distribution server.

E.1.2 Requirement changes during implementation

R1.6 Decision was made not to use establishments own authentication system, opting to use own account system.

R1.8 Decision was made to push content to Redis pub/sub server, rather than directly to distribution server.

E.1.3 Testing

Test Ref.	Req.	Method of Testing	Expected Result	Result
T1.1	R1.1	Click 'New plan' button, complete form, click 'create'.	New Plan available in Plan list, new Plan entry in database.	Pass
T1.2	R1.2	Select plan, click 'New mote' button, complete mote form, click 'create'	New Mote available in Plan's Mote list, new Mote entry in database.	Pass
T1.3	R1.3	Click 'delete' button next to Plan/Mote	Plan/Mote removed from list, entry removed from database	Pass
T1.4	R1.4	Click 'edit' button next to Plan/Mote	Same form as used with create action should appear pre-populated with existing values. Saving should modify existing Plan/Mote, not create a new one	Failed, difficulties with creating form for specific Motes. Implementation delayed until Iteration 5.
T1.5	R1.5	Use testing to call JSON output function on Mote	Function returns valid JSON containing all attributes for Mote	Pass
T1.6	R1.6	Use login form to login with demonstration account.	Login successful	Pass
T1.7.1	R1.7	Use login form to login with demonstration account	Ensure only plans created by demonstration account are visible	Pass
T1.7.2	R1.7	Use logout button to logout of account	Ensure no plans are visible	Pass
T1.8	R1.8	Click 'Push' button next to Mote	Use Redis console to check whether content is being received	Pass

E.2 Iteration 2 - Distribution Server

E.2.1 Requirements

R2.1 Receive messages published on Redis pub/sub channel.

R2.2 Use Redis to cache content.

R2.3 Facilitate resource serving for client.

R2.4 Implement socket.io to broadcast content to listening clients.

E.2.2 Testing

Test Ref.	Req.	Method of Testing	Expected Result	Result
T2.1	R2.1	Push mote from presenter system	View console log for distribution server, ensure mote appears.	Pass
T2.2	R2.2	Push mote from presenter system	Use Redis console to check whether content has been stored	Pass
T2.3	R2.3	Attempt to access test client	Test client should load	Pass
T2.4	R2.4	Open test client, push mote from presenter system	Test client should show broadcasted content	Pass

E.3 Iteration 3 - Client System

E.3.1 Requirements

R3.1 Enable users to authorise themselves for access to plans.

R3.2 Use socket.io to listen to content from distribution server.

R3.3 Display content based on template fetched from server.

R3.4 Navigate pushed content history.

R3.5 Prevent unauthorised users from accessing plans.

E.3.2 Requirement changes during implementation

R3.1 Decision was made to not require authorisation, but a plan-specific access code. Requirement changed to: Enable users to enter plan access code to subscribe to plan.

R3.4 Requirement delayed until Iteration 6 as client development overran schedule.

R3.5 Due to change in R3.1, this requirement was changed to: Ensure users only receive content for the plan they are subscribed to.

E.3.3 Testing

Test Ref.	Req.	Method of Testing	Expected Result	Result
T3.1	R3.1	Enter access code for plan	Latest plan content appears	Pass
T2.2	R3.2	Open client, subscribe to plan, push content from presenter	Content should appear in client	Pass
T2.3	R3.3	Open client, subscribe to plan, push content from presenter, try with different types of content	Content types should show with their own defined template	Pass
T2.5.1	R3.5	Open client, do not subscribe to plan, push content from presenter	No content should be displayed	Pass
T2.5.2	R3.5	Open client, subscribe to plan A, push content from plan B	Content should not change	Pass

E.4 Iteration 4 - Interaction

E.4.1 Requirements

R4.1 Allow client to return responses to distribution server.

R4.2 Support different structure of response for every mote type.

R4.3 Enable the presenter interface to listen for new responses.

R4.4 Support rendering of responses in the presenter interface.

E.4.2 Testing

Test Ref.	Req.	Method of Testing	Expected Result	Result
T4.1	R4.1	Respond to a pushed mote from the client system	Check for response in presenter interface	Pass
T4.2	R4.2	Try to respond to different types of pushed mote	Each returns different data to the presenter interface	Pass
T4.3	R4.3	Respond to a pushed mote from the client system	Check for response in presenter interface	Pass
T4.4	R4.4	Respond to a pushed mote from the client system	Response is rendered as specified in the mote type definition	Pass

E.5 Iteration 5 - Additional Features

E.5.1 Requirements

R5.1 Create additional mote types

R5.2 Create ajax-style interactions for content creation process

R5.3 Remove any device-specific code.

R5.4 Develop CSS for iPad-size display.

R5.5 Implement edit forms for Motes/Plans

E.5.2 Requirement changes during implementation

R5.2 Ajax-style interactions were not required, requirement changed to: Streamline content creation process.

R5.3 No device-specific code was found.

E.5.3 Testing

Test Ref.	Req.	Method of Testing	Expected Result	Result
T5.1.1	R5.1	Create motes of new mote types in presenter system	New motes created	Pass
T5.1.2	R5.1	Push motes of new mote type	New motes appear on client system	Pass
T5.1.3	R5.1	Respond to motes of new mote type	Responses displayed as expected on presenter interface	Pass
T5.2	R5.2	Present presenter interface to other users, instruct them to create plans and motes	Users able to easily create new content	Pass
T5.4	R5.4	Open client on iPad	Ensure all elements fit	Pass
T5.5	R5.5	Attempt to edit motes of varying types	Edit form, prepopulated with existing mote values appears, submitting form edits existing mote.	Pass

E.6 Iteration 6 - Polish System

E.6.1 Requirements

R6.1 Create additional CSS files to support other screen sizes.

R6.2 Streamline client user interface.

R6.3 Develop further content types, particularly for demonstration of other uses of the system.

E.6.2 Requirement changes during implementation

As features in this iteration were optional, it was considered a priority at this point to refactor code and improve the user experience. As such, the above requirements were not implemented, but the duration of the iteration was used to tidy the entire system to make it presentable.