Improving a RESTful Framework:
Utilizing Client Capability Detection, Incremental Loading and Tree Minimization

Will Lamond

advisor: Professor Sudarshan S. Chawathe

***Abstract*—**

Designing and deploying Web applications that a majority of browsers can utilize is an increasingly difficult problem as new browsers, browser features, and browser versions are developed. We present a method for building robust, extensible, and feature-rich Web applications by introducing the REST Framework for Dynamic Client Environments (RFDE) and covering our contributions to RFDE in detail. We begin by highlighting the problem of developing applications with the aforementioned traits. Two existing solutions to these problems, namely progressive enhancement strategies and representational state transfer (REST) are defined, and their roles in RFDE are examined. A number of contributions have been made, namely the introduction of AJAX support, an improved system for client capability deduction, expansion of the framework's provided feature set, and object tree clustering. We show how the new features improve the framework's robustness, reduce memory and network footprints, decrease running time, and improve developer usability.

I. Introduction

As people and technologies become more Web-oriented, creating robust and extensible Web applications and services becomes a problem of increasing importance. These systems may be accessed by many different devices with many different capabilities and needs. This uncertainty causes many developers to target subsets of these devices, leaving the remaining systems in the dark; either using a crippled version of the service, or worse, unable to use the software at all. The problem can be formulated with the following question: how can content on the Web be designed to reach the largest audience possible? The optimal solution would be applications and services that use information about the client to decide how to serve it best.

First, let us consider this problem's prevalence. We can count the number of operating systems, OS versions, browsers, browser versions, different scripting language interpreters and features, different document object model (DOM) traits, and different HTML standards [11] acceptance. Even though browsers and HTML standards abstract many of the details from developers, this laundry list of components still results in an explosion of possible combinations of system traits. As an example, Google's search engine, a system that is generally considered very reliable, has issues with older browsers being unable to interpret embedded scripts in their site's markup! In the case of Google, this just results in JavaScript code being belched into the browser window, but situations where much worse effects occur can easily be imagined.

Arguably, the most robust way to handle this large set of possible combinations is to utilize a strategy called *progressive enhancement* (PE) [19], which is the technique of detecting browser capabilities (typically through feature testing) and enhancing the served content as features are found to exist. PE is an excellent strategy since, in theory, an application can start with a baseline functionality that is recognized by all systems, such as an application that manifests in entirely HTML 2.0 [22] compliant markup. This guaranteed basic functionality provides assurance that any system can access the application, regardless of the most advanced capabilities the client may possess.

To tackle the problems of generality and scalability, many software systems on the Web utilize the representational state transfer (REST) system architecture. [18] REST was defined by Roy Fielding in his 2000 Ph.D. dissertation. The architecture provides a means for designing applications and services that behave in a manner that lends itself to scalability and generality. REST imposes constraints on client-server interface separation and generality, stateless servers, and an opaque, multi-layered server architecture. A system that conforms to the REST architecture is referred to as *RESTful*, with the largest RESTful software system being the Web itself.

The RESTful Framework for Dynamic Client Enviornments (RFDE) [13] was developed to utilize both PE strategies and the REST architectural design in a novel way. RFDE was originally created by Erik Albert to facilitate the development of an application for displaying scientific data to nonspecialists, and takes a two-pronged approach by using PE strategies to place clients in capability *tiers* and the REST architectural style for scalability and efficiency. Services written with RFDE use *widgets*, which are building blocks that fit into a specific niche of functionality. Widgets can perform diverse roles, such as offering user interface elements, computing map projections, or answering database queries. Widgets use the detected client tier to manifest themselves as an approximation of the best representation for the resource offered[1].

When we began working on our contributions to RFDE, the state of the framework was quite good; the system was thoughtfully designed, had a respectable set of widgets already in place, and used the two strategies discussed earlier in an effective way. Understanding that RFDE was originally made for a specific application guided our investigation, and upon closer inspection we found a number of places the framework could be improved; specifically, the way capabilities were detected and how that information was utilized, the way the application was loaded when a client supported JavaScript, and certain runtime behavior of the tree data structure that manages the widgets.

The framework originally made the sole distinction between clients capable of executing JavaScript code and

[1] This deserves some qualification, which is offered in section 4.

those that are not. While this is an important capability to detect, increasing the granularity of capability detection widens the set of clients that are equipped to handle RFDE-based services. Utilizing the HTTP request metadata, specifically the user agent string [23] and the history of browser and HTML standards development, we can deduce client capabilities with greater precision. This improvement was focused on application-oriented services, but offers valuable insight for other services, too. We discuss the improved granularity and this detection methodology's implications further in section 4.

Additionally, when a client supported JavaScript, the entire application was generated (on a client-per-client basis (an important caveat we return to later in discussing the widget tree data structure) and sent to the client as a whole. This greatly improves the application's user-perceived responsiveness, but at the cost of significantly reduced performance during the application's initial loading phase. Instead of this bulk loading strategy, we utilize the jQuery AJAX APIs [2] to incrementally load requested portions the application into the existing DOM. The details of this solution and the experimental results are featured in sections 5 and 7.

Widgets are organized in a general tree data structure in order to facilitate the mapping process from Java objects to HTML, XML [17], and other Web-oriented data. The servlet performs a depth-first traversal of the widget tree during the request-response cycle to perform the required mapping, and each widget is visited during the traversal. We present a method for detecting clusters of widgets that do not change during the lifetime of the application, and can be clustered to reduce the traversal space.

The paper is organized as follows. In section 2 begin with an introduction to the anatomy of an application written with RFDE. We pay careful attention to the widgets' structure, widget interactions, the application state, and the client-server transaction model. These specific traits will act as a segue into our improvements. Section 3 discusses the expansions to the set of widgets, using Tabs as a case study. This section serves two purposes: to give the reader a concrete view of RFDE, and to provide details of the original bulk loading strategy. Section 4 delves further into the capability detection methodology, specifically the process by which a client is examined to deduce a capability tier, and how this information is used to progressively enhance the served application or service. Section 5 discusses the changes to the client-server interaction model, specifically the incremental loading strategy. Section 6 discusses the widget tree compression techniques used to improve server performance. The methodology for detecting compressible subtrees and the compression technique itself is discussed in detail. Section 7 presents experimental results showing the performance improvements introduced by the incremental loading and widget tree compression. We conclude with section 8, where future work is introduced and discussed in some detail.

II. What is RFDE?

RFDE is an extensible framework for developing robust, scalable web applications that are accessible by a wide range of clients. RFDE includes both a client/server model and a collection of widgets (RFDE terminology for modules, functional entities, or resources) that can be used, extended, and created by Web programmers to develop RESTful applications and services that adhere to progressive enhancement strategies. RFDE's original focus was on application-oriented Web services, so our discussion reflects this focus. As we show later, this does not reduce the generality of RFDE, but will simplify our discussion for now.

Clients are placed in a tier of feature support that is based on a client's capabilities. Once a client's tier is determined, widgets will manifest in an approximation of the most modern way the client can safely handle. Two illustrative examples of clients falling into different tiers are:

- A client supports client-side scripting. A mapping of widgets to JavaScript can be used to further enhance the widgets' aesthetics and usability.
- A client supports only HTML 2.0, and is trying to access a critical application aimed at the general public. The all of the same widgets from the previous example will generate HTML 2.0-compliant markup that is accessible from any browser.

These two extremes, as well as any system in between, can access applications that utilize RFDE.

An application developed with RFDE is defined as the following. A Java servlet creates an instance of the `AppTemplate` class (known as a *template*) and a set of widgets that implement the application's features and resources. Each of the application's widgets are instantiated with the desired default values (consisting of content, events and actions), and structured as a collection of parent-child relationships in the widget tree. The widget default values allow the template to build a default application *state*, which can be used at request time in conjunction with client state information to determine precisely what state a client is in. Once the widget tree is laid out, the template's `init()` method is called and the application is ready to handle requests.

It should be noted that this is the common pattern for RFDE: all structuring of the application is done in the servlet's constructor.

A. Template

The template is the root of the widget tree and provides facilities for performing the application's initialization processes, placing clients in tiers, application component communication, and executing request procedures. The template's responsibilities are the following:

- Determining which tier a client belongs in.
- Determining which events were triggered by parsing the application's *state* encoded in the universal resource identifier (URI) and firing the actions associated with the triggered events.

- Composing a JavaScript file containing initialization code for each widget that has a JavaScript mapping.
- Computing results for AJAX calls.
- Invoking the correct (based on the tier) `writeHTML()` method for each widget that is a direct child of itself.

When the servlet receives a request, the template's `execute()` method is called. All of the tasks that the template is responsible for are performed in this method, and it is important to note that nearly all of the servlet's computation time is spent in this method, or is called from this method.

B. State

The state is constructed from two parts: a default state that is built at application initialization time from the widgets' given values, and a URI-encoded string that is responsible for holding the application's last configuration on a given client.

Some widgets determine their output from the current state, namely from the state variables associated with the widget. A state variable may be referenced by ID of the widget and the name of the widget's state variable that you wish to obtain. The state may be modified by actions performed by widgets (not necessarily the widget that the variable belongs to, which is one method of inter-widget communication), and if the output of the widgets is based on the state, the widgets reflect the changes.

State variables should capture the critical information that is passed back and forth from the client to the server. Minimizing the amount of state variables a widget maintains is advantageous for not only simplicity of the widget, but also to minimize the memory footprint the widgets leave on the URI.

C. Widgets

Widgets are functionally specific "blocks" application developers can use to construct their applications. Widgets are responsible for presenting a specific domain of functionality. That is, widgets should be treated like any object in an object-oriented system: their responsibilities should be as minimal and specific as possible.

It is useful to consider widgets from a REST point of view. The functionality that a widget implements (say, the weather at a given zip code, or an image) is referred to as a *resource* in REST terminology. From Fielding's dissertation, "a resource R is a temporally varying membership function $M_R(t)$, which for time t maps to a set of entities, or values, which are equivalent." This definition allows resources to vary over time, such as in our weather example, or remain static after an initial point in time, such as the image. It is important to note that two distinct resources may map to a single representation at some points in time, but that does not mean they are the same resource.

All widgets have two basic responsibilities:

- Provide a mapping of the widget to each tier. This involves implementing one `writeHTML()` method for each tier, and a client-side.
- Maintain given base values, such as labels, IDs, and subwidgets. Additionally, these values are encoded in a string constructed in the widget's `init()` method that is used to map the widget to a JavaScript representation.

This mapping of the resource the widget maintains is called a *representation* in REST terminology. A representation is the sequence of bytes that represent the resource at a given time, along with the metadata that is associated with that sequence of bytes. A representation can take the form of a plain text document, an XML file, an image, or any other representable entity.

D. Interaction

D.1 Events

In order for a widget to have an event, it must implement an event interface. The event interfaces specify the sort of events that will occur with the widget. For example, the `Tab` widget has a click event, so it must implement the event interface `HasClickEvent`. Widgets can implement multiple event interfaces, and thus have multiple events associated with them. These events can all be tied to actions which are fired, by the template, when the events are triggered.

D.2 Actions

Actions are tied to events at the same point the widgets are created and structured in the template: the servlet constructor. The application developer can tie as many actions to an event as he chooses, and they will all be fired when the event is triggered.

The actions are added to an event using the event's `addAction()` method. While the point of an event is to detect something that happens, an action is supposed to *do* something once an event is fired. This is included but certainly not limited to animating part of the application, making a pop-up alert, panning a map, or even modifying part of the internal representation of the application (*i.e.* a state modification).

An action can act on a widget other than the widget whose event is tied to the action. This allows widgets to change the state of other widgets, allowing inter-widget communication.

III. Feature Set Extension: Tabs (First Version)

RFDE's original state had widgets for the application it was originally developed to build. This set of widgets supplied images, buttons, alert windows, maps, and more. The biggest missing feature we noticed was a way to build multi-faceted applications that allowed users to navigate freely from one application component to the next. An example of such an application is LibreOffice, which provides the typical menu at the top along with a tool bar with buttons that are associated with specific functions. We aim to accomplish the same goal with `Tabs`.

The `Tab` widgets placed on the page are each linked to unique content. This allows the application developer to control how much information is displayed, and group similar information in a single section of the application. Tabs are grouped together in a TabPanel, which is responsible for hiding and displaying content associated with each Tab, and for styling the tab that is deemed "active."

Tabs in tier 1 are composed of horizontally aligned anchor tags that send an HTTP GET response when clicked that fires its associated event. The TabPanel container uses this event to set the clicked tab's associated content as 'active.' Since a page refresh has occurred, the Java servlet calls all the widget's writeHTML() methods. By default, all content hosted by a tab is non-active, so no HTML is written except for the last clicked tab.

With HTML 4.01 in tier 2, we have a bit more flexibility. A div tag wraps around an unordered list of anchors (a common way to build tabular navigation) and uses CSS. The styling options allow the Tabs to look like more tab-like instead of anchors. The content of each tab is styled with "display:none" when the active tab does not have the same index as the content. The result is that the only displayed content is the active Tab's, with the rest of the content hidden on the page.

The benefit of doing it this way is that the content is there for the JavaScript '+' tiers. When JavaScript is enabled, it takes over the hiding and showing of content, and so in order to take advantage of JavaScript's speed, it must have access to all of the content without using a page refresh. Sending the client all of the content also allows the JavaScript to make use of jQuery to perform sliding and fading animations.

The difference between tier 2 and tier 3 Tabs is the use of the HTML 5.0 nav tag to wrap the unordered list instead of the div used in tier 2. The nav tag is used for screen readers and web semantics.

Using the Tab widget as an example, their responsibility is to refer to specific content of the application. That tabs are given the ID of the part of the application (typically a content panel, but there are no actual restrictions) to refer to it in the document object model (DOM). Each Tab has an onClick event, which triggers a StateChangeAction with the TabPanel as the target widget. This changes the state of the TabPanel to reflect that the tab that was clicked is now the "active" Tab, as well as telling the object the Tab refers to in the DOM that is is to display its content.

The Tab's onClick event is tied to a StateChangeAction, which changes the state of a TabPanel to reflect that it (the Tab) has been clicked. This allows the TabPanel to mark the appropriate Tab as selected, and display the content associated with the Tab.

Continuing our example, the interface HasClickEvent stipulates that implementing classes must implement the method onClick(). We simply give the Tab class an Event named *onClick* as one of its fields, and have the onClick() method return the aforementioned event.

This event is initialized in the Tab constructor. The event constructor is passed a reference to the owner object (the Tab), and a string that declares the type of event (in the case of Tab, the event is a click event, so the string passed is "Click"). The event type string is used later when the template is writing the JavaScript versions of the widgets. The event type should be one expressive word for clarity's sake, but can technically be any arbitrary word.

IV. Client Capability Detection

Accurately detecting a given client's capability tier is arguably the most important aspect of RFDE. Without accurate tier placement, a client may receive content it is not equipped to handle, resulting in either a presentation that is not up to par with other applications on the Web, or worse, content that is completely unrecognized by the client.

As mentioned earlier, client's are examined and placed into capability tiers based on the information gathered during the examination. Tiers need to be defined in such a way that gives enough granularity to allow many types of browsers to receive content that is akin to what is expected, but that also does not introduce unnecessary complications to the system as a whole. A *hierarchical tier* approach was used that divides clients in to their respective tiers, some of which have branches that incorporate the REST aspect of "code on demand." Returning to the application-based RFDE service for the sake of discussion, we have defined three tiers, two of which allow optional code on demand. The basic idea behind each of the tiers is as follows:

- The first tier is designed to be so basic that even the earliest of web browsers should be able to understand it.
- The second tier tries to realize a middle ground, catching clients who have newer technologies, but do not have the latest and greatest the W eb has to offer.
- The third tier utilizes HTML 5.0 [21], and CSS3, providing the most current and advanced features.
- A JavaScript + branch is utilized whenever the client has JavaScript activated, and allows for improved application response time and aesthetics.

A. Application Tier Mapping

A suitable mapping of these requirements to actual Web applications requires the use a variety of technologies. The obvious choice is varying the HTML standards by which the application manifests. Additionally, we would like to style our application and use client-side scripting when a client is capable of doing so. We utilize the following tier mapping for applications:

- Tier 1: HTML 2.0
- Tier 2: HTML 4.01 [25], CSS
- Tier 2+: HTML 4.01, CSS, JavaScript
- Tier 3: HTML 5.0, CSS
- Tier 3+: HTML 5.0, CSS, JavaScript

Tier 1 is composed of widgets constructed with entirely HTML tags allowable in the HTML 2.0 standard, released in November 1995 and specified in RFC 1886. [22] The purpose of using this version of HTML is that *almost all* browsers can handle it, since it specifies all HTML used in the Web's earliest days.

Upgrading to Tier 2 reconstructs the widgets using elements found in the HTML 4.01 standard, released in 1999 by the World Wide Web Consortium (W3C). The jump from HTML 2.0 to 4.01 was chosen because the features added in the versions released between these two were too insignificant to dedicate an entire new set of widget versions. For example, HTML 3.2 [24] included `div` and `id` attributes, but stylesheets were not fully specified until a year after its release, so a browser that supports HTML 3.2 might not support stylesheets, rendering its new tags useless. HTML 4.01, on the other hand, has many usable tag attributes, `div` and `table` tags, and has complete support for styles.

As noted earlier, the distinction is made between tier 2 and tier 2+. The difference is the introduction of JavaScript to allow for client-side event handling, effects, and incremental loading of the application. JavaScript is given its own tier because it is possible for a browser to support HTML 4.01 and CSS, but not support JavaScript, which is rather unlikely, or the client may have JavaScript disabled, which is slightly more likely but still must be accounted for.

Tier 3 introduces HTML 5.0. This allows for the newest features of the web to provide enhanced functionality, improved HTML semantics. HTML 5.0 is used in the highest tier since it is one of the newest Web technologies available, and will most likely become a standard in the very near future.

Again, the distinction is made between tier 3 and tier 3+ in the same way as in tier 2. In general, a *+* is used to indicate that JavaScript is supported at the given tier level.

B. Tier Detection Methodology: Feature Testing

Accurately determining the client's tier is an important problem. If the tier is not accurately determined, a client may suffer from receiving content that is not up to snuff with other applications on the web, or worse, receive content that the browser can not interpret at all.

Applications that strive for cross browser support typically use a technique called *feature testing.* [12] Feature testing is a scripting language method that involves testing the browser for implemented features. An example of this is in DOM manipulation, where Mozilla-based browsers differ from Internet Explorer in the way the DOM elements are selected and thus manipulated. The following JavaScript code snippet illustrates the technique:

```
if (document.getElementById) {
// If the W3C DOM API is supported
}
else if (document.all) {
// If the IE 4 API is supported
}
else if (document.layers) {
 // If the Netscape 4 API is supported
}
else {
// Otherwise, DHTML is not supported
}
```

The first condition checks if the browser supports the `document` object, specifically the `getElementById` function. If the `getElementById` function is supported, we can continue to perform our DHTML techniques in the usual way. Similarly, the second and third conditions test for the `all` and `layers` functions, respectively. If they are found to exist we can use them in the typical ways.

Feature testing works well for JavaScript, but not for HTML standard support recognition, which, as noted earlier, we are interested in for full-fledged application support. Instead of relying solely on feature testing, we utilize other observations for HTML support detection. An integral part of this method is the *user agent string*

C. Tier Detection Methodology: User Agent Identification

The user-agent string [23] is a value in an HTTP request and contains information about the user that the request originates from. The string is used for statistical and content selection purposes. The user agent string typically contains information about the request machine's operating system and browser, which can be used to decide what content is supplied in the response.

The user agent string alone does not supply enough information about the request origin to accurately place a client in a tier; simply knowing information about the client's browser is not enough. However, in conjunction with knowledge about the HTML standards' release dates, we can determine acceptable content on a client-per-client basis in a reasonable way by mapping browser type and version ranges to HTML standards usage. This method requires the following understanding of HTML standards releases:

	IE	FF	Chr	Saf	Op	NS
HTML 2.0	1.0	X	X	X	2.1	1.0
HTML 4.01	5.5	1.0	1.0	1.0	4.0	6.01B
HTML 5.0	9.0	4	3	5	11.6	X

TABLE I
HTML STANDARDS AND MINIMUM BROWSER VERSIONS

Each HTML standard is paired with each browser in the above table, with the *minimum* version of the browser required for each standard noted in the corresponding table cell. This table can be mapped directly to a conditional structure in the tier detection code to take the browser version in the user agent string and determine which HTML standard it is best suited for.

For this to work, we need to determine the browser type and version from the user agent string. A library called *user-agent-utils* [15] is used to determine exactly this. The library is abstracted using a `Tier` class which takes the user agent string from the client's request and returns the tier value based on the conditions discussed above.

The user-agent-utils library works in the following way. The user agent string is passed to a `UserAgent` constructor, which has the methods `getBrowser()` and `getBrowserVersion()`, which return `Browser` and `BrowserVersion` objects, respectively. These objects work in the expected way, and provide an interface for acquiring information about the browser of the provided user agent's string value.

With the `Browser` and `BrowserVersion` objects in place, we can answer the two questions we are most interested in: the browser name and major version number. The browser name can remain a string for matching purposes, and we convert the major version to a double for simple range matching with the typical relational operators. The following Java code snippet illustrates both steps:

```
public static int determineTier(String userAgent){

//some preliminary processing on userAgent here

UserAgent ua = new UserAgent(userAgent);

Browser b = ua.getBrowser();
Version v = ua.getBrowserVersion();

String browserName = b.getName();
Double browserVersion =
Double.parseDouble(v.getMajorVersion());

//match name and version ranges here

}
```

D. Issues

There are a number of issues with this approach, one of which is that the Tier class must be updated for every new browser and HTML standard. Our current implementation takes into account all the major browsers and the versions of HTML that they support. Any new standards or browsers will need to be added manually when they are released.

The user agent string may be spoofed to show a different browser than what is actually making the request. This is often done for the browser to "request" content that it is equipped to handle but may not be reflected in the user agent string's value. We assume that clients will spoof for this reason and this reason alone, since clients that spoof to reflect capabilities that they do not have cannot be detected with this system, nor is there a reason to intentionally request data that cannot be interpreted.

The final problem with this system is that the client's user agent could be formatted in a completely unrecognizable way, and the tier defaults to 1 in this case. To combat this, we send a small piece of externally-linked JavaScript for the sake of detecting the capability of executing JavaScript. This small script simply upgrades the client to tier 2, the lowest tier that can execute JavaScript. With the tier upgraded, the script then reloads the application to reflect the new tier placement.

V. Incremental Application Loading via AJAX

The `Tab` widget adds a nice touch to Web applications by providing an interface users expect. Navigating an application by spotting a named clickable object that appears to be what one needs and exploring the application in an incremental fashion leads to more intuitive interfaces, an opportunity to add nice animations, and a more satisfying user experience in general. As discussed earlier in section 3, our initial design of the `Tab` widget for plus tiers was based on `Tab`s managing the content displayed by the application, similar to how word processors, image editors, and many applications utilize such a menu. This intuitive approach accomplishes exactly what we needed from *Tab*s: give users an opportunity to control the flow the their application usage by building all application components in the `writeResponse()` method and returning it to the client.

This design neglected one problem: the possible size of an application. Our initial approach consisted of hiding all but the selected content, and it accomplished what we wanted quite well. The unintended consequence of such an approach was the overhead of constructing the entire application and transferring it over the network. This is manageable for small responses, but as service capabilities improve, which increases both the construction time and response size, sending that much data at once reaches speeds that are unacceptable. This is especially true when one considers the typical user's reaction to application initialization time. Consider when one visits a Web site; if the site doesn't begin to respond within a few seconds of the request, frustration sets in and many users will more on to something that works faster.

Our solution is to carry the idea of incremental exploration of an app to the way the application is loaded. We wish to maintain the benefit of client control as well as minimize the requests made to the server, but to reduce the cost of building the whole application on initial request time and sending it over the network. We use the industry standard way of loading data into a Web page without loading a new page is a technique coined "asynchronous JavaScript and XML," known colloquially as AJAX. [16]

A. RFDE and AJAX

AJAX is used to asynchronously exchange data with a server in a client-side scripting language such as JavaScript. The technique involves sending a request to a server in a similar way to a browser, but the result is handled by the calling JavaScript code instead refreshing the page (this is where the asynchronous part comes in). We utilize the following advantages AJAX has over full page refreshing:

- Retrieving data with an AJAX call leaves the page unaffected unless an effect is implemented. This is advantageous when the majority of the page stays the same, such as when the layout and app control structure remains but the information displayed to the user is new. The unaffected portion of the page needn't be reserved by the server. This has the obvious benefit of reducing server load by reducing the amount of data prepared for the client in a given exchange to only the data the client does not already have, as well as the reduced network footprint of shrinking the exchanged data size.
- The AJAX exchange may be tied to actions performed

on the client. The typical method for this is to use an *event listener,* which waits for an action to be performed on a specific part of the page and performs the call when this event occurs. The call may utilize other information supplied to other parts of the application by the client, or simply fetch requested data. This means the information sent to the client is based on the client's needs, reducing extraneous processing on the server and network footprint.

The name AJAX is actually a bit misleading. The name does not mention any of the other data formats that AJAX calls can receive and use. Our primary focus wit `Tabs` is to receive portions of the application in HTML format; AJAX can do this without a hitch. We use the jQuery AJAX API [2] to request data and modify the application DOM accordingly. jQuery provides two functions that simplify AJAX exchanges, namely the `html()` [3] and `load()` [4] functions.

The two functions complement each other very well. The `html()` function acts on a typical selector call by augmenting the HTML inside to be what is provided as an argument to the `html()` function. For example:

```
$('p#myid').html('New␣content');
```

The $ selector works in the typical way; fetching the DOM element with the specified tag type and identifier (with the # symbol). The `html()` function replaces the old content inside of the paragraph tag (`p`) with the new content, namely the "New content." The `html()` function returns the selected element with the new content inside.

This is complemented by the `load()` function, which takes the selected object returned by the `html()` function and replaces the content *again* with whatever is returned by the specified resource. As an example call:

```
$('p#myid')
.html('loading...').load('site.com','r=rName');
```

This call replaces the content of the paragraph with the identifier "myid" with the string "loading..." and then replaces that new content with the representation of the resource selected from the service "site.com" via the parameter `r` set to "rName."

B. Tabs and jQuery

`Tabs` already select content based on the specified application `Panel` given to them at initialization time. Instead of transferring all of the application's content at application request time, we instead construct an *application skeleton.* The skeleton consists of placeholder `div` elements that are devoid of content, with the exception of one `div` that contains the application's initial view. When a `Tab` is clicked, the DOM element it refers to is requested using jQuery's `html()` (for placeholder purposes) and `load()` functions.

C. Fitting AJAX into RFDE

Two problems need to be addressed to have fully-functioning AJAX `Tab` widgets. The first is client-side content management. We do not want the client to make redundant requests for the same data, so we need the browser to cache the responses and have the client-side `TabPanel` keep track of what needs to be requested and what is already there.

During the `TabPanel`'s initialization phase on the server, we perform an inspection of the given `Tab` selection. During this inspection, a JavaScript dictionary is built that reflects the state of the application at initialization time. The following code illustrates the dictionary building procedure:

```
String initContent = ``{'';
for (int i = 0; i < this.countWidgets(); i++) {

 if (this.selected == i)
  this.getWidget(i).getDomRef().setActive(true);
 else
  this.getWidget(i).getDomRef().setActive(false);

 initContent = initContent.concat(i+``:''+``\`''+
 (this.selected == i ? ``1'':``0'') + ``\''' +
 (i != (this.countWidgets() - 1) ? ``,'' : ``}''));
}
```

The `initContent` string is used in the mapping of the the `TabPanel` to JavaScript. It creates an instance variable for the given `TabPanel` that maintains key-value relationships between each `Tab`'s integer identifier and the boolean, in this case a 0 or 1, for "not loaded" and "loaded," respectively. The client-side code that implements the `TabPanel` updates the dictionary values as content is requested, and detects that content is already available if the dictionary's value for the given `Tab` reflects such availability.

D. Client-Side Initialization

The second problem is that of client-side initialization. The client-side code refers to the DOM element that a `Tab` makes reference to during the initialization phase, causing the script to crash when the DOM element has not been loaded. What is needed is a way of differing widgets initialization to a later time when the required DOM portion is in place.

It is useful to take a look at how widgets were originally initialized. Each widget maintains a string that is used to write its initial values to the JavaScript test. This test script is called by an event listener that waits for the window to load completely. The consequence of this is, as discussed earlier, that the script never begins to execute if the listener does not activate (*i.e.* the client does not support JavaScript). The listener calls the function `TierInit()` when the DOM loads, and this initialization function creates instances of each widget's JavaScript mapping and adds the mapping to the client-side widget manager.

Widgets with event listeners which wait for interactions with the browser require that their DOM elements be in place during the mapping function execution. This constraint exists because of the use of Prototype, which requires full knowledge of the DOM element in question to construct the event listener. If the DOM element is not in place, the initialization fails. Because of this added constraint, initialization must be differed to after the DOM portion is loaded.

Lucky for us, jQuery's `load()` supports callback functions, which provide exactly the semantics we are looking for. Call back functions are anonymous functions which are provided to functions such as `load()` that specify required behavior for a particular action. In our case, we provide a callback function to `load()` that calls the `InitTier()` function.

This method requires a slight modification to `InitTier()`. This will attempt to initialize all widgets regardless of whether or not their respective DOM elements are included. All that is needed to fix this problem is to use the widget's unique ID to try and select the DOM element before attempting to initialize the widget. The following code snippet illustrates the simplicity of the check:

```
add : function(wclass, id, initargs) {

if (this.getWidget(id) == null) {
  if (!($(id)))
   return -1; // no such element
 // init code
 } else {
  return null;
 }
}
```

This simple check performs a typical DOM selection for an element with the same ID as the widget. Recall that widgets prepared for JavaScript mappings are given the same HTML ID as their server-side ID, so this selection should return exactly the DOM element associated with the widget. If the widget is not found, the selector will return null, and performing the logical not on the null will evaluate to true, returning the integer -1 (acting as an error code).

VI. Widget Tree Minimization

As noted in the introduction to RFDE, widgets are arranged in a tree structure. Trees are a common structure on the Web, with data formats such as HTML, XML, and JSON being tree-like. The widgets' tree structure reflects the formats widgets are typically written to, simplifying the mapping from Java objects to Web-oriented representations.

Such a mapping must maintain the tree structure that is required of Web-oriented data. Arguably the simplest technique that preserves the nested relationships of the widgets is the *depth-first traversal* (DFT). The DFT algorithm takes the root of the tree and recursively calls itself on its children in the order they appear in the node. This traversal propagates down the tree until it reaches a *leaf* node (a node with no children), which is where it terminates. The traversal preserves the parent-child relationships of the widgets, and provides a sound mapping of widgets to their given representations.

As noted earlier, the mapping involves writing the widgets' representations, as Java strings on the server, to the required Web format (*e.g.* HTML, XML, etc) and appending the Web-formatted data to the servlet's response stream(`ServletOutputStream` [8]). This is done through multiple calls to the response stream's `append()` method for each widget. It is possible to observe groups of widgets which may be clustered into one widget, with one string representing the output, allowing the mapping function to make one call to `append()` rather than many.

A. Invariants

Some widgets never need to recompute their output, yet their `writeHtml()` methods must be called on every request, and their content must be recomputed and appended to the response's output stream. We observe that widgets that are siblings, leaves, and invariant needn't be recomputed on a request-per-request basis. These widgets may be *clustered* into single entities with single content strings. The clustering process yields the benefit of reducing the traversal space and thus reducing the number of function calls need to construct the widgets' collective representation.

To detect widget invariance, we first need to know exactly what causes a widget to vary. A widget changing is a result of one of two things: either it requires access to parameters passed from an HTTP verb which are stored in the state until all events are processed, or it is affected by the action of a widget (either itself or another widget), which is also reflected in a state variable.

In both cases, the widget variance may be detected by inspecting the application's state. Since all state variables exist at application initialization time, the state may be inspected during a *clustering phase* for variables associated with a given widget. If a widget is found to have no state variables, it is said to be an invariant widget.

Additionally, a container widget may hold widgets which do indeed vary, so a widget that is to be clustered must not be a container. Detecting whether or not a widget is a container is a straightforward process: it must implement the methods of the `WidgetContainer` interface. We may use the Java reflection API [14] to determine if a widget implements the `WidgetContainer` interface, and thus whether or not it is a container type.

The following pseudo code describes the clustering algorithm:

Algorithm 1 pseudocode for clustering algorithm

```
1:  Function Cluster($Widgets$, $State$):
2:  wc ← new WidgetCluster();
3:  for widget $w$ in $Widgets$ do
4:    if $State.get(w) = \oslash$ then
5:      if isLeaf($w$) then
6:        wc.add($w$); $w \leftarrow \oslash$;
7:      end if
8:      if ¬isLeaf($w$) then
9:        Cluster(w.getContainer(), State);
10:       wc ← new WidgetCluster();
11:     end if
12:   end if
13: end for
```

The function `Cluster()` takes a list of widgets called

`Widgets` and the application's state (referred to as `State`). First, it creates a cluster to be placed in the first position that is found to be acceptable. Then, for each widget in the collection, we check for an empty state variable set, then whether or not it is a leaf node. If so, we add it to the cluster, and if not, we perform a recursive call of `Cluster` and create a new cluster for the next set of widgets to be potentially clustered. An example of the algorithm follows:

$$\begin{array}{|c|c|c|c|c|c|}\hline w_1 & w_2 & w_v & w_3 & w_4 & w_5 \\ \hline \end{array}$$

$\Downarrow$

$$\begin{array}{|c|c|c|c|c|c|}\hline wc_1\{w_1, w_2\} & \oslash & w_v & wc_2\{w_3, w_4, w_5\} & \oslash & \oslash \\ \hline \end{array}$$

The first list contains two invariant widgets in the two leftmost positions (denoted by w_1 and w_2), followed by a variable widget (w_v), and then three more invariant widgets (w_3, w_4, and w_5). After the widget clustering algorithm is performed on the list, the result is the array (below $\Downarrow$) contains a cluster of w_1 and w_2 in the first position, a null ($\oslash$) in the second, the variable widget in its original position, a second cluster of the remaining widgets in the fourth position, and nulls in the remaining positions.

B. Dealing With Sparsely Populated Arrays

Unfortunately, we are not finished. The array is sparsely populated in a way that is unpredictable to iterate over without running into a number of null sections. The number of operations involved in iterating over the sparse list is the same as without the clustering; resulting in less benefit than we would like. What remains to be done is the removal of nulls from the result array, finishing with a fully populated array containing nothing but widgets and clusters we must visit.

The intuitive approach is to use the available tools provided by Java's `List` interface. The `List` interface [6] provides a `remove(`i`)` function which removes the i^{th} element from the list and shifts everything to the right of the i^{th} element to the left by one, effectively shrinking the array and decrementing the index of each shifted element by one. This results in a simple algorithm for removing the nulls (iterate over the list and call `remove()` on each null), but the worst case performance is less than attractive. Let us consider a case where the result array contains n clusters, with each cluster being a combination of at most m widgets. The removal process just described will have m calls to `remove()` for each of the $n-1$ clusters (omitting the first cluster which is never preceded by nulls). Each call to remove will have fewer and fewer operations to perform; specifically the first call will perform $(n-1)m$ index operations, the second call will perform $(n-2)m$ index operations, and so on.

The runtime expression (denoted S) is as follows:

$$S = (n-1)m + (n-2)m + (n-3)m + ... + 2m + m$$

$$= m \sum_{i=1}^{n-1} i$$

$$= \frac{mn(n-1)}{2}$$

The resulting worst case running time is $O(mn^2)$.

C. Condensing the Arrays

Instead of relying on the Java `List`'s remove function, we can benefit by creating our own algorithm. The algorithm can make use of the `List` interface's `set(`i,e`)` function, which sets the i^{th} element of the list to the value e in constant time (for an `ArrayList` [1] implementation of the `List` interface). The algorithm makes use of the observation that in a single pass over the array we can detect pockets for placing later widgets and clusters, as well as place the widgets and clusters in those pockets. The algorithm is as follows:

Algorithm 2 pseudocode for cleaning algorithm

```
Function Cleanup($W$):
baseNull ← -1;
**for** $i \leftarrow 0$ to $W.size() - 1$ **do**
   **if** $W.get(i) = \oslash \wedge$ baseNull = -1 **then**
      baseNull ← i;
   **end if**
   **if** isWidget($W.get(i)$) $\wedge$ baseNull $\neq$ -1 **then**
      $W.set(baseNull, W.get(i))$;
      $W.set(i, \oslash)$;
      baseNull++;
   **end if**
**end for**
**for** $i \leftarrow W.size() - 1$ **down** to baseNull **do**
   $W.remove(i)$
**end for**
```

Consider the following example. The `Cleanup()` function takes an array W as input, say, the output array from our clustering example earlier:

$$\begin{array}{|c|c|c|c|c|c|}\hline wc_1\{w_1, w_2\} & \oslash & w_v & wc_2\{w_3, w_4, w_5\} & \oslash & \oslash \\ \hline \end{array}$$

We iterate over the array to detect the first pocket, then subsequent iterations place detected widgets in the pocket and increment the pocket index. The result array looks like this:

$$\begin{array}{|c|c|c|c|c|c|}\hline wc_1\{w_1, w_2\} & w_v & wc_2\{w_3, w_4, w_5\} & \oslash & \oslash & \oslash \\ \hline \end{array}$$

The second loop begins at the end of the array and truncates it null by null, resulting in zero shifts and an output array that looks like the following:

$$\begin{array}{|c|c|c|}\hline wc_1\{w_1, w_2\} & w_v & wc_2\{w_3, w_4, w_5\} \\ \hline \end{array}$$

The worst case running time can be characterized as the following. Consider an input array with n clusters and at most m nulls per cluster. A total of $n-1$ clusters must be indexed and relocated (omitting the first cluster, which is always in its correct place), and at most m nulls must be detected for each of the n clusters. Additionally, mn nulls must be truncated from the end of the array in the second loop, m for each cluster. The sum (S) of the operations is:

$$S = (n-1) + 2mn$$

resulting in a worst case running time of $O(mn)$; reducing worst case performance of the cleanup code by a factor of n.

D. Combining Clustering and Cleanup

After considering what must be done during cleanup, we can devise an algorithm that combines the efforts of the two algorithms in the previous sections into one procedure. The new algorithm keeps track of the contiguous pockets and places unclusterable widgets and clusters in the null pocket as they are detected rather than waiting until all widgets have been clustered. The only portion of the previous cleanup algorithm that must be postponed is the removal of trailing nulls. While the complexity of the algorithm is asymptotically the same, we still gain a benefit equal to a constant factor of two by removing the first of two passes over the array from the cleaning algorithm. The resulting clustering algorithm is the natural combination of the two algorithms.

E. WidgetCluster Data Structure

The `WidgetCluster` data structure is used to maintain the output of each widget contained in the cluster as a single string to be appended to the servlet's output stream. This can easily be achieved by concatenating additional strings representing adjacent widgets to the cluster's current representation. This process must be done for each of the three tiers, and additionally for each subsequent widget that is addable to the cluster. The resulting structure contains three strings, one for each tier, each representing all of the widgets added to the cluster.

Two methods were implemented for this: one that combines widgets by performing concatenations on the representation strings, and another that utilizes a string buffer for each representation string. The first implementation was optimal in terms of space; since Java strings are immutable, concatenating strings together involves allocating enough space for both strings, placing both strings in the new space, and flagging the old strings for garbage collection. This is optimal in terms of memory, since it immediately gets rid of unused space after the new string is generated from the concatenation. The issue here is the time it takes to allocate the new memory to contain both strings. We will explore the running time of this implementation in the performance evaluation section.

The second method utilizes Java's `StringBuffer` [9] instead of direct string concatenations. Using buffer space instead of reallocating memory for every concatenation reduces computational overhead by reducing the number of allocations that must be performed. This method takes a penalty in terms of memory overhead by allocating more space than is needed by the string representations stored in the `WidgetCluster` data structure. We show in the performance section that this space overhead is not completely unreasonable, and the speed gained (50% decrease in time taken) out-weight the additional memory overhead.

VII. Performance Evaluation

We take a three-pronged approach to evaluating the performance improvements by measuring both the time it takes for the site to initially load on the client browser, and the time the server takes to initialize for future requests, and the time the server takes during the execution of one request-response cycle. All experiments were performed a machine with a single core 1.7GHz x86 processor and 2GiB of RAM, running Ubuntu Linux 10.04 LTS and Tomcat 6.0. [20] The JVM's maximum heap size was increased to 1GB to accommodate the largest tests. We utilize the WebPageTest [10] testing framework for the AJAX measurements, which is freely available under the BSD license.

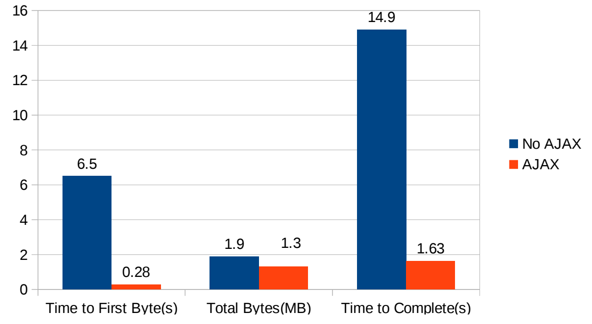
A. AJAX

As discussed earlier, the original rendition of RFDE's `Tab` widget involved hiding content that was not selected as active and displaying that content once the associated tab was selected. This resulted in very fast response time to user interaction, but very drawn-out initialization time. To combat this, we introduced AJAX support, as discussed in section 5. For our benchmark application, the initial measurements are the following:

1. Total bytes loaded: 1.9MB
2. Average time to first byte: 6.5s
3. Average time to completely load and render: 14.9s

Our improvements from the AJAX introduction are reflected in Figure 1:

Fig. 1. Performance Introduced via AJAX



Note that the units of measurement are in seconds, MB, and seconds for the first, second and third columns, respectively.

The measurement of the time to first received byte is the most representative of the AJAX performance improvement. This value mainly considers the time it takes for the server to construct the application's representation and send the markup to the client. This measurement does not reflect the time taken to download certain parts of the benchmark application, specifically the images that the map is made up of and necessary JavaScript files. The time to receive the first byte is the best measurement of the performance increase since the time spend waiting for the first byte is exclusively time taken on the server to generate the markup and network latency, and is therefore not masking any benchmark-specific problems.

The decrease in the total number of bytes loaded upon initialization is marginal. Again, to understand where this decrease is coming from, it is important to consider the benchmark application itself. The application contains a map, and the map is made up of many small images. The majority of the reduced size can be attributed to the loading of the map being differed to a later time. The remaining size is mostly attributed to JavaScript files, which are necessary for representing the application in the plus tier.

Our measurements of this improvement come full circle with the inspection of the total time to load and render the application. Again, the initial measurement's is partially attributed to server involvement ($\approx 50\%$), and loading the map tiles. While this improvement seems to be over-dramatized by the map tiles, by considering the time taken to receive the first byte pre-AJAX, and the time to completely load the application post-AJAX, we see that using AJAX clearly improves the overall initialization performance.

B. Widget Tree Clustering

We must consider two factors when measuring the performance increase introduced by clustering the widget tree: the initialization time and the request-response cycle time. We expect the initialization time to increase since we are increasing the amount of computation being performed during the initialization phase, and we expect the request-response cycle time to decrease since we are reducing the number of function calls and traversal space. The measurements were taken with a randomly generated test application. The application formed a tree of depth 8, branching factor of 4, with S content widgets per branching location and at each leaf ($S = 5$, 30, 60). Our measurements show that exactly this happens, and are reflected in Figure 2 (NC = no clustering; C = clustering):

We see the dramatic increase in initialization time. These measurements are for the clustering implementation that utilizes string concatenation. Figure 3 compares the concatenation and buffer implementations' average initialization times. The buffer implementation is much faster because of the large reduction in number of memory allocations for the representation strings.

Fig. 2. Initialization Time

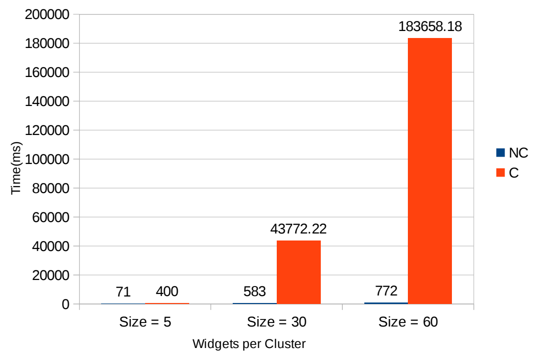
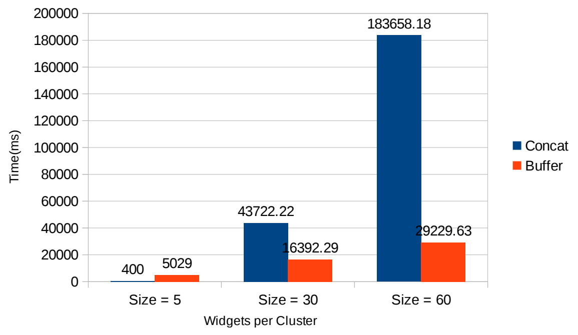
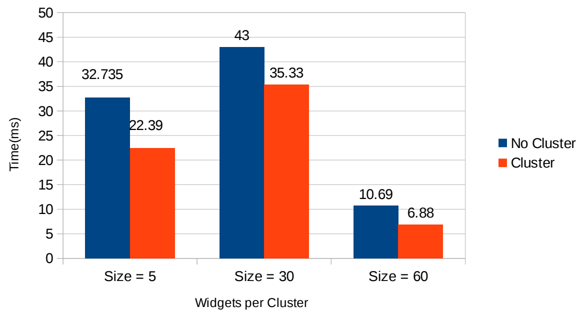


Fig. 3. Concatenation vs. Buffer Implementations



Our final measurement is of the request-response cycle time on the server. To measure this, timing code was placed around the template's `execute()` method call. Please recall that nearly all computation performed on the server is done in this method (other than the servlet's `doGet()` method, which only calls `execute()` and nothing else). We see a reduction of approximately 7% in the average request-response cycle time, reflected in Figure 4.

Fig. 4. Request-Response Cycle Times



An anomaly occurs in figure 4; specifically the reduction in response time when the widgets per cluster reaches 60. This anomaly may be explained by the response stream better utilizing buffer space. The response stream's buffer starts at a capacity of 8 kilobytes, and the stream ignores requests for less buffer space under the conditions of our experiments. The buffer ignoring the request for a reduced capacity shows that some optimization may be occurring behind the scenes, which may be responsible for the performance increase as the volume of content increases. This question deserves further investigation, but at this point the reason for the performance is unknown.

VIII. Related Work

A. JavaScript Support

A common goal for JavaScript libraries is to work towards aiding developer productivity or implement cross-browser compatibility in some way. A number of tools do a fairly good job of this, but are typically focused on niche portions of application development. An example of a tool that aids developer productivity is the JavaScript library Prototype [7], which accomplishes the goal of aiding developer productivity by simplifying object-oriented design of JavaScript classes by introducing a rigorous inheritance pattern to a language devoid of such a pattern. The library's main goal is to assist in class design, and does not focus as much on cross-browser compatibility. To fill this gap, many developers use the JavaScript library jQuery to handle cross-browser compatibility issues. The jQuery library provides a DOM element selector function that abstracts browser-specific details from the developer, providing a seamless interface to DOM manipulation that considers browsers as old as Internet Explorer 6.6. [5] Consequently, both of these libraries provide AJAX support; a common theme in JavaScript libraries. These tools do what they were designed for very well, and are used in RFDE's class design and cross-browser support for JavaScript features.

B. Representational State Transfer

As its name suggests, RFDE is based on the Representational State Transfer architecture design pattern of Web software development. Software systems that adhere to the REST paradigm must follow a number of principles:

- A uniform interface between clients and servers. This improves portability of interface and server code between platforms. In the case of the Web (or main focus), this interface is the set of HTTP verbs (GET, PUT, POST, etc).
- Servers do not maintain explicit client states, reducing the overhead of a server taking on a new client. Instead, clients maintain their own state, either explicitly though some client-side state mechanism, or implicitly through the universal resource identifier (URI).
- Data offered by servers is explicitly or implicitly marked as cacheable or not cacheable. This opens up the possibility of reducing the application's network footprint, but also allowing the server to mark data as short-lived (such as rapidly changing financial information, which should not be cached for an extended period of time).
- Clients cannot tell what layer of a service they are communicating with, improving the flexibility of service design. Services can then use HTTP verbs to delegate work to other services that implement specific functionality, increasing overall system modularization.
- Clients may receive code on demand from servers. This is an optional constraint, which is a bit counter intuitive. The constraint is optional, since services may be able to extend the capabilities of a client by sending things such as Java applets or JavaScript code. There are cases where a server will not send code, or simply cannot since the client cannot utilize it.

C. RFDE-base Applications

So far we have dealt with Web services made with RFDE that act as full-fledged applications. This is a useful application of RFDE, but it is important to note that it is not the only one. An RFDE-based service may not contain any user interface elements, and consist of only widgets that provide some desired function or service. An example of this would be a map tile projection, which takes latitude, longitude, and zoom parameters and returns the floating point numbers that comprise the screen coordinates of the map's tiles. Such a service does not need a user interface, and instead of writing the service representation out as HTML and JavaScript, such a service may instead use JSON or XML to represent the output.

Additionally, services written with RFDE can write the necessary representation based on a representation request made by the client. Using the map projection as an example, a client may be implemented to use JSON-formatted projection output. Such a client may request that the projection service respond with a JSON representation of the output, while at the same time another client may require the XML representation of the same exact resource.

IX. Concluding Remarks

This project has been a fascinating and illuminating introduction to Web systems. The project combines multiple facets of Web services, specifically server- and client-side implementations of services and interfaces, RESTful systems, and designing with baseline functionality in mind to improve robustness. This project has introduced me to technologies and techniques that are currently used in Web system development and will likely continue to be used for some time.

The experience of receiving a working code base and taking the time to carefully examine and understand the system as a whole is extremely valuable. Knowing when to ask for assistance and when to buckle down and chug through problems is a useful ability to have. Taking the time to understand a system and devise ways to improve it, then subsequently evaluating the changes and quantifying the improvements is very satisfying. Being exposed to this cycle of examine-improve-evaluate will likely prove to be the most valuable part of this project for me.

The results of the project are promising. RFDE offers a way to create robust Web applications that utilize a widely-used architecture and a number of useful technologies. Using HTTP metadata offers finer capability granularity than feature testing alone, and the two complement each other quite well. The introduction of `Tabs` offers a way to create multi-faceted applications and interfaces that are intuitive to users. Using AJAX to load the application as needed has proved to be significantly faster than the original bulk loading method, and is a testament to why AJAX is so widely used. Finally, something as simple as combining strings in buffers to reduce the number of function calls can increase performance on a request-per-request basis; an improvement that adds up over the course of many thousands of transactions a Web service may handle in a given time frame.

RFDE is not without fault, however. Creating new widgets and defining ways that widgets can interact with one another is not always the most straightforward exercise. Future work with RFDE may include defining a language for creating new widgets. Such a language would provide a set of widget traits and ways of interaction, and specify a syntax for defining the structure and semantics of widgets. Additionally, a tool that converts existing applications to RFDE-based implementations would be useful.

References

[1] ArrayList API. http://docs.oracle.com/javase/6/docs/api/java/util/ArrayList.html.

[2] jQuery AJAX API. http://api.jquery.com/category/ajax/.

[3] jQuery AJAX API. http://api.jquery.com/html/.

[4] jQuery AJAX API. http://api.jquery.com/load/.

[5] jQuery Browser Support. http://docs.jquery.com/Browser_Compatibility.

[6] List Interface API. http://docs.oracle.com/javase/6/docs/api/java/util/List.html.

[7] Prototype JavaScript Framework. http://www.prototypejs.org/.

[8] ServletOutputStream API. http://docs.oracle.com/javaee/6/api/javax/servlet/ServletOutputStream.html.

[9] StringBuffer API. http://docs.oracle.com/javase/6/docs/api/java/lang/StringBuffer.html.

[10] WebPageTest. http://www.webpagetest.org/.

[11] World Wide Web Consortium. http://www.w3.org/MarkUp/.

[12] Bob Clary. Browser Detection and Cross Browser Support. Mozilla Developer Network, 2003. https://developer.mozilla.org/en/Browser_Detection_and_Cross_Browser_Support.

[13] E. Albert, S. Chawathe. REST Framework for Dynamic Client Environments.

[14] Glen McCluskey. Using Java Reflection. http://java.sun.com/developer/technicalArticles/ALT/Reflection/.

[15] Harald Walker. user-agent-utils. http://code.google.com/p/user-agent-utils/.

[16] Jesse James Garret. AJAX: A New Approach to Web Applications, 1995. http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications.

[17] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter. Extensible Markup Language (XML) 1.0, 1999. http://www.w3.org/TR/REC-xml/.

[18] Roy Fielding. *Architectural Styles and the Design of Network-based Software Architectures.* PhD thesis, University of California, Irvine, 2000. http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.

[19] Steven Champeon. Inclusive Web Design for the Future. http://www.hesketh.com/thought-leadership/our-publications/inclusive-web-design-future.

[20] The Apache Software Foundation. Apache Tomcat 6.0.

[21] Web Hypertext Application Technology Working Group. HTML Living Standard. http://www.whatwg.org/specs/web-apps/current-work/multipage/the-xhtml-syntax.html.

[22] World Wide Web Consortium. Hypertext Markup Language - 2.0. http://www.w3.org/MarkUp/html-spec/html-spec_toc.html.

[23] World Wide Web Consortium. RFC 2616. http://tools.ietf.org/html/rfc2616.

[24] World Wide Web Consortium. HTML 3.2 Reference Specification, 1997. http://www.w3.org/TR/REC-html32.

[25] World Wide Web Consortium. HTML 4.01 Specification, 1999. http://www.w3.org/TR/html4/.