Highlight: A System for Creating and Deploying Mobile Web Applications

Jeffrey Nichols[*], *Zhigang Hua*[†], *John Barton*[*]

[*]IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120
{jwnichols, bartonjj}@us.ibm.com

[†]GVU Center & College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
hua@cc.gatech.edu

ABSTRACT

We present a new server-side architecture that enables rapid prototyping and deployment of mobile web applications created from existing web sites. Key to this architecture is a remote control metaphor in which the mobile device controls a fully functional browser that is embedded within a proxy server. Content is clipped from the proxy browser, transformed if necessary, and then sent to the mobile device as a typical web page. Users' interactions with that content on the mobile device control the next steps of the proxy browser. We have found this approach to work well for creating mobile sites from a variety of existing sites, including those that use dynamic HTML and AJAX technologies. We have conducted a small user study to evaluate our model and API with experienced web programmers.

ACM Classification: H.5.4 [Hypertext/Hypermedia]: Architectures – Dynamic HTML, AJAX, proxy servers, mobile clients.

General terms: Algorithms, Design, Human Factors

Keywords: Highlight, mobile web, dynamic HTML, JavaScript, proxy browser, proxy server

INTRODUCTION

Use of the web from mobile devices is becoming increasingly popular [15]. Many popular web sites offer mobile versions, and new devices, such as the iPhone, offer easy browsing of desktop-sized web pages on a small device. Unfortunately, recent survey results suggest that only about one-third of all mobile web users are satisfied with their experience [19]. We believe that two reasons for this low satisfaction are:

- existing mobile sites do not always offer streamlined access to the functionality each user wants;
- web site designs for desktop browsing are not always appropriate for scenarios when the user is on-the-go.

UIST'08, October 19–22, 2008, Monterey, California, USA.

A possible solution to both of these problems is to create technologies that allow users to create their own mobile web sites by re-authoring existing web sites. Using this approach, users choose the features that they want included in their mobile sites, avoiding the problem of missing functionality that exists in many of today's mobile sites. Re-authoring can also allow users to remove irrelevant content, skip over pages that are not needed, and break up complex pages into multiple simpler pages. These types of changes allow users to streamline the desktop browsing experience into something more appropriate for mobile situations. Figure 1 and Figure 3 show some examples of mobile applications that might be re-authored from existing sites.

As part of the *Highlight* project, we are building re-authoring tools and a deployment system to investigate the feasibility of re-authoring a wide variety of existing web sites. In previous work [17], we have explored how programming-by-demonstration techniques can be used by end users to re-author web pages with little or no embedded JavaScript. In this paper, we describe our server-side system that allows programmers to re-author existing pages that include dynamic JavaScript and AJAX (not possible with our PBD tool) and to deploy the re-authored applications to mobile devices.

This paper makes the following contributions:

- The use of a remote control metaphor and persistent state for easing the creation of mobile sites and making the re-authoring of sites with dynamic JavaScript and AJAX possible;
- The architectural design of placing a fully functional web browser in a proxy server to enable use of the remote control metaphor, and an implementation of this design.

Unlike previous systems that have used proxy servers that act as a pass-through filter for the HTML stream (e.g. [10, 11]), in Highlight's design the mobile device remotely controls the web browser embedded in the proxy server. Requests from the mobile device are translated into interactive operations on the current page in the proxy browser, such as filling in form fields and clicking links. Once the result of those interactive operations is available, relevant content from the new page is clipped from the proxy browser and returned to the mobile device. A mobile application description, which is uploaded to the server in advance and

chosen by the mobile user at the beginning of their session, guides the interaction between the mobile device and the proxy browser.

The remote control metaphor requires the proxy browser to maintain its state across each of the mobile device's requests. We have found the persistent state, along with the remote control metaphor, to be very useful in re-authoring existing sites for mobile devices. For example, a long form on an existing site can be broken up into multiple mobile pages. When the mobile returns the first part of the form, those fields can be filled in on the proxy browser and the next part of the form returned to the mobile device. When all of the fields have been filled in, the proxy browser can be instructed to submit the form.

The persistent state also enables the Highlight infrastructure to re-author pages with substantial amounts of dynamic JavaScript and AJAX features. These pages can alter their content without making new requests to their server or may make requests to the server through XmlHttpRequests that return data that cannot (or should not) be modified. These qualities make these pages impossible for previous proxy server-based systems to handle. Highlight is able to handle such features because it works at the interactive level. An interaction from the mobile device may trigger JavaScript to execute in the proxy browser. Once that code has completed and the current page has been updated, the result can be clipped and sent back to the mobile device.

We have evaluated our system from two different angles. First, we have used our system to implement and deploy mobile applications for a wide variety of existing web sites, including sites using modern AJAX toolkits such as Dojo [2]. This includes mobile applications programmed by hand and built with our end user tool. Second, we conducted a small user study with experienced web programmers to test the usability of creating new applications by hand.

RELATED WORK

The Highlight proxy server uses a remote control architecture in which the mobile device controls the proxy server's web browser through its requests. There has, of course, been other work in remote controlling one computer from another, such as VNC [20]. WinCuts [23] and Façades [22] are related systems that allow portions of an application to be remotely controlled, enabling users to re-author an existing application by choosing and re-arranging its on-screen rendering. These systems both use bitmaps to show the state of the remote system and replicate the exact mouse

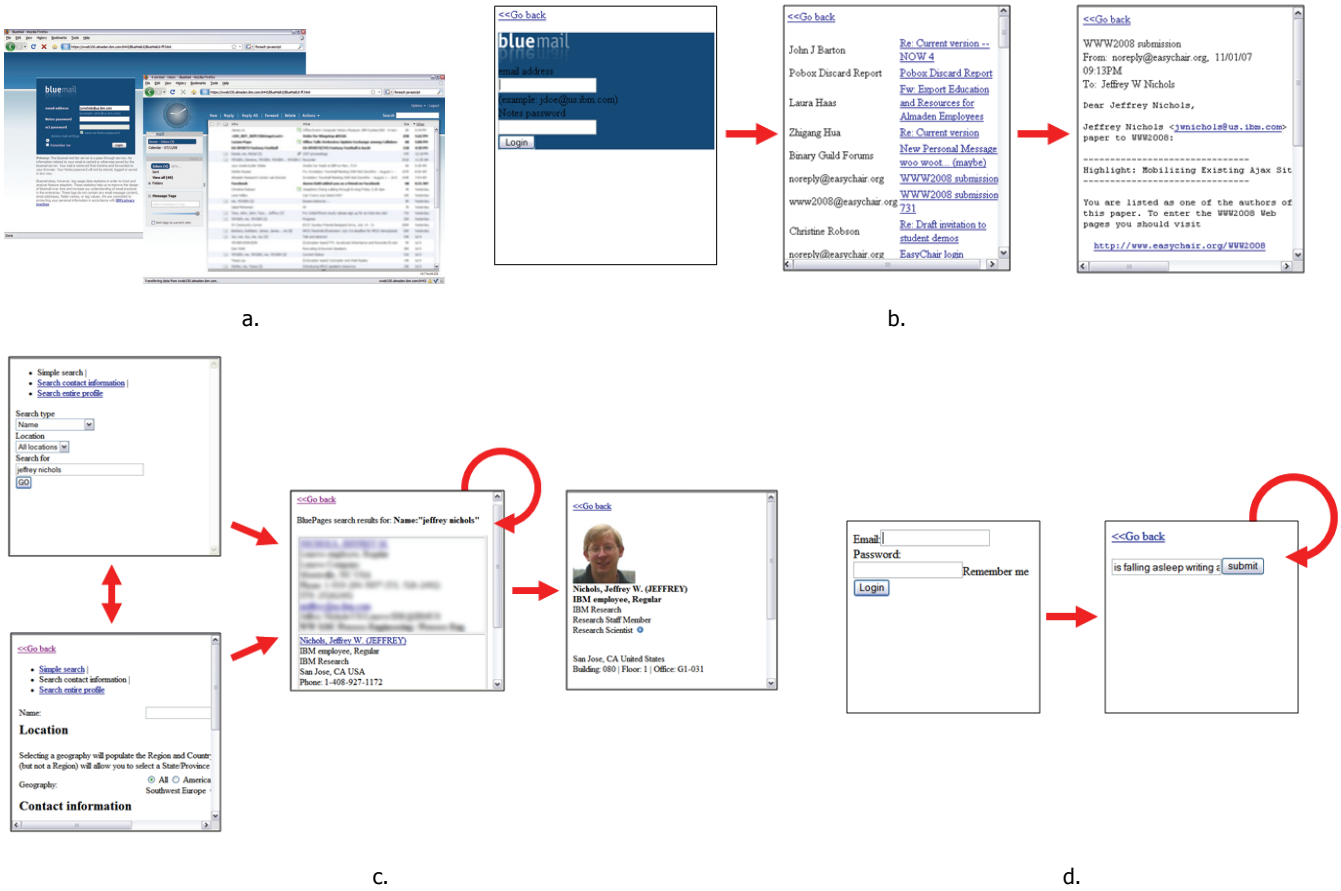

Figure 1. A variety of mobile applications created using Highlight. a) The existing interface for BlueMail, showing two different views served via the same web page. b) Our proof-of-concept mobile application for BlueMail. c) A mobile application for the IBM corporate directory (BluePages). d) A mobile application for changing your facebook status, built by one of our user study subjects.

and keyboard events from the controlling system on the controlled system. The Highlight proxy uses a more abstract approach, where the page seen by the proxy server is modified for mobile viewing and events from the mobile device must be translated into interactions with the proxy browser.

Another relevant system is Adeo [9], which allows mobile phone users to run previously recorded macros in a web browser on a remote machine and receive back the results on their phone. Inputs can be provided to the macros at the start, but otherwise macros run completely independent of the phone and only return when they have completed. Adeo's architecture has an element of the remote control metaphor used by Highlight, although in Adeo the browser is treated more like a function that returns a result instead of as an interactive entity. We believe that Adeo could be built using Highlight, however the reverse would not be possible.

A recent focus of mobile web work has been creating new browser interaction techniques that attempt to improve the mobile browsing experience. The iPhone's multi-touch interface is the best example of such a technique that is also having success in the marketplace. However, we believe there is still a place for content modification approaches such as that of Highlight. Modified interfaces will almost always be smaller and easier to navigate as compared to regular web pages. As long as the modified interface meets the needs of the user, then the modified pages should be faster and easier to use.

Substantial research has been conducted in the area of adapting web content [24] for mobile devices, and particularly in the area of using proxy servers to perform the adaptation. Much of this work has developed heuristics for automatically modifying HTML, CSS, and embedded images to suit the properties of the viewing device. For example, Digestor [5] modified web pages for mobile viewing with a set of heuristics that included splitting a single page into multiple pages corresponding to the outline structure of the original page. More recently, heuristics have been explored for modifying pages based on repeating structures [13]. This work has culminated in a variety of commercial services, such as Google Mobile Search [10] and Skweezer [11], which allow users to search the web and view result pages that are transformed for viewing on a mobile device.

A major difference between our work and the previous work is that most of these proxy server-based content adaptation systems seem to ignore Javascript. After adaptation, code on the pages may still function if its targets still exist and the code can still find the targets that have moved. A US patent [21] describes a possible method for partially addressing this problem, by automatically searching the JavaScript code in a page and changing any references to page elements that have moved. It is unclear how well this method would work in practice however. Furthermore, if JavaScript is used to generate new HTML content within the page, then these systems will be unable to alter the new content because this code is executed on the client after the proxy server's processing is complete. The Highlight proxy server is explicitly designed to address both of these issues.

d.mix [12] is a novel system that allows users to modify content from sites by selecting and combining page elements that are generated by calls to web services, potentially from different pages on multiple sites. A proxy server is used to store and serve applications created with d.mix and the creation of mobile interfaces is supported. d.mix seems to be limited to using content that can be extracted by a web service function call however, so it is not able to create mobile versions of pages that do not have a corresponding service.

A popular method of modifying existing web pages is to inject custom JavaScript code directly into the page. The custom code then modifies the page by hiding or removing existing elements, adding its own new elements, and changing the event handlers that process user interactions. This approach was pioneered by Greasemonkey [3], an extension to the Firefox web browser, which is a client-side implementation that injects code into the browser once a page has been loaded. Monkeygrease [16] and Accessmonkey [6] perform code injection through a proxy server, which allows Greasemonkey-like functionality for browsers that do not support the Greasemonkey extension, such as most mobile browsers and Internet Explorer.

Like the Highlight proxy browser approach, code injection allows for modification of pages that include dynamic JavaScript and AJAX. There are some trade-offs between the two approaches however, and we will discuss these later after describing Highlight in more detail.

ARCHITECTURE

The architecture of the Highlight system uses a proxy server, which connects existing web servers and mobile devices. In our system, the web server believes that it is connecting to a normal desktop browser and is not aware that its content is being forwarded to a mobile device. Similarly, the mobile device is also not aware that it is connecting through a proxy. We chose this approach so that the connection could be seamless without requiring any configuration of proxy settings on the mobile device, which can sometimes be difficult and may only be desired for a few of the sites that the user visits. The only requirement Highlight places on the mobile device is that it has a web browser that can speak standard HTTP protocols and render at least simple HTML content. The Highlight system does not support converting HTTP/HTML to WAP/WML, though this could be added in the future. Assuming the mobile device has a compatible web browser, it is not necessary to modify or install any software.

The proxy server contains an HTTP server, a database of mobile application descriptions, and a proxy browser. When the user of a mobile device wishes to access an application, they point their mobile browser at the proxy server's main page where they will see a list of all the applications available in the proxy server's database (see the

first page in Figure 3). Choosing one of these applications causes the proxy browser to be opened, and the corresponding mobile application description specifies the steps that the proxy browser will follow to reach the page corresponding to the first page of mobile content. The application description also describes the content that should be clipped from that page and sent to the mobile client.

It is important to emphasize that the proxy browser is automated at the *interactive* level. Form fields are filled in and appropriate change events are generated, and links and buttons are "clicked" by generating click events with the appropriate targets. The advantage of this approach is that Highlight does not need to do anything special to invoke any client-side JavaScript code that may be included in the existing web page. For example, many web pages have links that reference a JavaScript method within the page rather than a URL on a web server. In other cases, event handlers for a link may be added dynamically by other JavaScript code on the page. By artificially clicking the link in the proxy browser, Highlight ensures that the original web page designer's assumptions about page operation will be maintained without needing to analyze JavaScript code or track the current state of the page. Moreover, this approach requires no knowledge of the server-side implementation of the web application beyond what can be inferred from its user interface. No knowledge of the application's client-side JavaScript implementation is needed either, unless the programmer wishes to use some portion of the existing JavaScript code in their mobile application.

Highlight's process of clipping content from a page is also performed without modifying the structure of the existing page. To accomplish this, nodes of the existing document are cloned into a new document and any modifications, such as removing unneeded elements, are performed in the new document. This approach is in contrast to code injection approaches, such as Greasemonkey, which modify the content of the page directly. Modifying the existing page may invalidate the assumptions of any JavaScript contained in that page, causing the page to no longer function.

Content clipped from the existing page will likely contain interactive elements, such as links and form fields. By default, Highlight rewrites links and forms to direct back to the proxy server and also provides the application description with a mapping between the elements on the mobile page and those in the proxy browser. Interacting with the mobile page will cause a new request to be sent back to the proxy server, where the request is interpreted by the mobile application description. In most cases, the mapping provided by the Highlight infrastructure is used to replicate a user's interactions on the mobile device within the proxy browser. For example, if the user clicked on a link in the mobile page, the corresponding link will be clicked on the proxy browser page. More advanced programmers can also create application descriptions that add extra behavior, such as setting values for form fields that exist in the original page on the proxy browser but were not included in the mobile page. In all cases, the application description automates navigating the proxy server to the next page that contains content for the next mobile page. This content will be clipped and modified according to the application description, and then sent to the mobile device. Thus, the nature of the interaction between the proxy browser and the mobile device is similar to that of a remote control. Interactions with the mobile device cause similar, though not necessarily identical, interactions with the proxy browser to retrieve the next set of content.

To make the creation of mobile application descriptions easier, Highlight provides a programming interface (API) that makes automating the proxy browser and clipping content much easier. This API is built on top of the standard DOM APIs already available in browsers, and augmented by a set of methods for automating interactive operations based on those used in the CoScripter system [14]. In our current prototype implementation, the API can also be extended by installing Firefox extensions, such as Chickenfoot [7], into the proxy browser.

The mobile application description provides one possible mapping between an existing web site and a mobile version, and multiple mobile application descriptions may be specified for a single web site. Users choose the application they wish to use when they begin using the proxy server. The advantage of this approach is that different users may specify different mobile versions of a single site and a single user may create multiple mobile versions of a single site that reflect different tasks that are performed using the site. For example, the same user might specify different applications for accessing the current weather and the 10-day forecast on weather.com. Another user might create a more complicated application that combines these two features.

Our current implementation of the Highlight architecture contains multiple proxy browser instances; one proxy browser instance for each mobile application that is being used on the proxy server. This design trades off scalability for improved security and privacy, and also allows for easier detection of side-effects. For example, if a page opens in a separate window we can assume that it is linked with interactions going on within its browser's session because the only inputs to a proxy browser instance are a single mobile client and the existing web site used by the mobile application.

Scalability is an important concern however, and a separate proxy browser instance is a significant resource to allocate for each session. We believe that to make this system work at a significant scale, a headless browser will be needed that is designed explicitly to work in an unattended environment. Given that there are several full featured browsers that can run on mobile devices, such as Opera and Safari Mobile, it seems likely that a headless browser could be created with a small enough footprint that many instances could run on a single machine. Aptana's Jaxer product [4], which embeds a Mozilla browser on the server-side for DOM parsing but not remote control, suggests that such a

solution may become available. An alternate approach may be to use a distribution model more akin to that of the Slingbox, where users host their own Highlight servers at home for use by their own mobile devices.

Prototype Implementation

We have implemented a prototype version of our system using the Apache Tomcat server and a custom Java servlet as the HTTP server component and the Mozilla Firefox web browser as a proxy browser instance. A difficulty with the current prototype is that the Firefox browser is not designed to be used in an entirely automated environment. The browser will often display user interface elements outside of a web page, such as modal dialog boxes, that must be dealt with appropriately by the automation system and, in some cases, communicated to the end user. An important example of this situation occurs when the browser visits a page that uses HTTP authentication. In this case, the browser will display an authentication dialog box that must be filled in. Our prototype proxy server will automatically identify this situation and send an authentication web page to the mobile user. When the results of this page are submitted, they are filled into the dialog box and execution of the application continues. Similar strategies are used to deal with other dialog boxes. This example introduces issues of security, which will be discussed later.

DESCRIBING AN APPLICATION

Conceptually, Highlight mobile application descriptions can be thought of as a storyboard of multiple mobile pages, or *pagelets*, that are connected by links and forms. Some example mobile application storyboards can be seen in Figure 1 and Figure 3. The structure of a mobile application storyboard overlays the structure of the existing web site, but does not need to match it. For example, the mobile application may skip pages in the existing site or divide a single existing page into multiple pagelets. When a mobile application is running, its current location within the storyboard is tracked in order to determine the possible next pages for the application.

Highlight mobile application descriptions are contained in a single JavaScript file that defines an object for each pagelet, each of which contains two specific methods:

- The **clip method** generates the content for the current pagelet from the content of a page from the existing site. This method identifies content from within the current page, extracts the appropriate pieces, and transforms them into the HTML content that will be sent to the mobile device. This method may also add new content, including JavaScript, to give mobile pages extra functionality or support improved interactivity.
- The **event method** for a pagelet is called when the mobile client is viewing that pagelet and then initiates a connection to the server, typically by clicking a link or submitting form data. The event method examines the web request, determines the interaction taken by the user, and performs interactions within the proxy browser to reach the next page of content. This may involve stepping through multiple pages on the way to the page that contains the content for the next pagelet. The event method also determines which pagelet is next and when the proxy browser is ready for the next clip method to be executed.

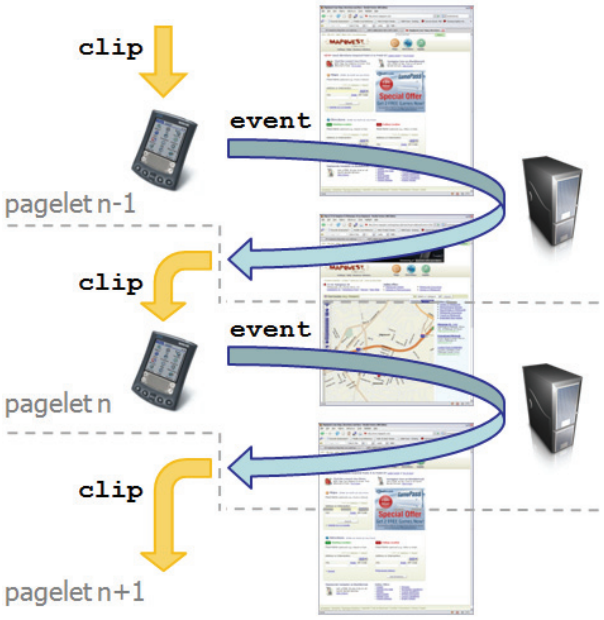

Figure 2. A timeline view of a mobile application being executed on the proxy server showing the order in which the methods from the mobile application description are called.

Figure 2 shows a graphical depiction of the interactions between the mobile device, the pagelet methods, and the proxy browser.

A key decision in the design of Highlight was choosing a level of abstraction for the mobile application description language; we considered using either a relatively abstract declarative language of our own design or an existing programming language. In making our decision we considered a few criteria:

- ease of creating applications for humans;
- ease of creating applications for computers;
- complexity of applications that could be created.

The advantage of using a declarative language would have been increased ease of creating applications for humans and computers, and a reduced need for debugging. The disadvantage, however, was that applications would be limited to the complexity that our language allowed. Exploring a new idea would have required extending the declarative language and then implementing code to handle the language changes within the proxy server system. Instead, we chose to use an existing programming language to describe our applications. This decreased the ease of creating applications by humans, did not affect the ease of creating applications by computers, and greatly increased the complexity of applications that could be created. This

design also gave us increased ability to explore the design space for applications by first writing the entirety of our applications within the mobile application description, and then over time moving the commonly used pieces of code into the proxy server's API (an ongoing process).

To offset the increased difficulty of creating applications for humans, we have been investigating authoring environments that use programming-by-demonstration techniques to allow users to create mobile application descriptions without writing any code [17]. Once an application is specified within the authoring environment, the corresponding JavaScript code can be generated and uploaded to the proxy server. Note that this end-user environment is restricted in the types of pages that it can re-author, such as those that use lots of dynamic JavaScript or AJAX. Programming an application by hand also gives the author substantial extra power, including the ability to add new features that were not part of the existing page, to re-author sites that use AJAX, and to include JavaScript and AJAX in the pages sent to the mobile device.

BUILDING A MOBILE APPLICATION

To demonstrate more concretely how a Highlight application is built, we will describe how a simple mobile to-do list application can be constructed from the Backpack web site [1]. This site uses an AJAX design to provide users with a flexible interface for creating and aggregating lists of content. The Highlight mobile to-do list application uses a small subset of Backpack's features, and converts the original AJAX interface into a simple HTML form-based interface for access on a wide-variety of mobile devices. The full version of this application description can be viewed at:

`http://www.jeffreynichols.com/highlight/BackPackTodo.js`

This example illustrates the two biggest challenges for constructing a mobile application from web, and particularly AJAX, applications:

- Identifying the content to clip/manipulate on the existing page;
- Determining when content is ready to clip.

Note that both of these issues are shared by any tool that re-authors existing web content, including code injection approaches like Greasemonkey.

For clipping content we support several different methods: traversing the pages' DOM structure with the standard JavaScript API, an XPath expression [8], a heuristic representation known as "slop" that was pioneered by Chickenfoot [7] and CoScripter [14], and a Dojo toolkit-specific [2] method that traverses the toolkit hierarchy to find the "widgets" included in the page. The particular method used depends on the web site being mobilized and the author of the mobile application description. For AJAX applications that are based on a publicly available toolkit, the best approach may be to make use of the toolkit's methods for traversing its widget-level description of the page. In this example, we

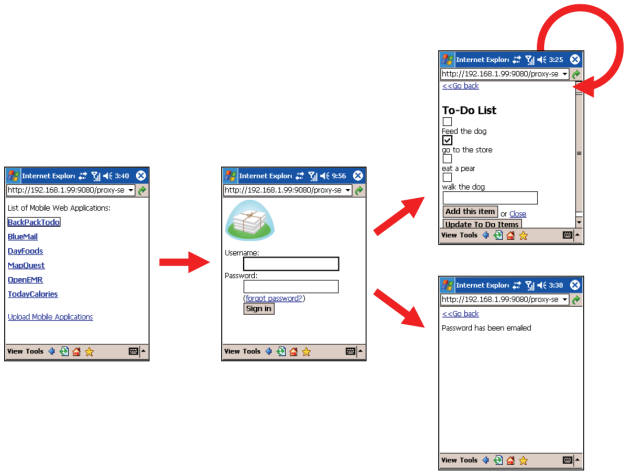

Figure 3. A storyboard diagram of the Backpack To-Do List application, showing screenshots of the actual application being used through Mobile Internet Explorer on a Windows Mobile device. Also shown is the application list view from the Highlight proxy server.

choose to use a combination of DOM API methods and XPath expressions for clarity and brevity.

Determining when content is ready to clip is most difficult for web sites that use client-side JavaScript and AJAX. In these cases, interacting with the page may invoke code that modifies the DOM in some way. The delay between interacting and the DOM change will take an arbitrary amount of time, as AJAX applications may request data from the server before performing the modification. DOM level 2 events standardize support for eventing based on structural changes within the DOM, however most current web browsers provide inadequate support for these features. As a result, Highlight provides two API methods to help application authors determine when content is available. The first, used in our example here, is the ability to delay for a specific period of time before continuing. Authors can use this feature to manually poll for the existence of an element. Highlight also has a `pollForItemExist` method that polls internally and calls a callback when an element with a particular XPath expression exists and is visible.

BackPack Example Walkthrough

The first step in building a Highlight application is to decide on the structure of the mobile application. This includes determining what pagelets the mobile application will have and how they will be connected. A storyboard of our to-do list application is shown in Figure 3. We chose to have three pagelets in our application: a login pagelet, a confirmation pagelet for when the user clicks the "forgot password" link, and the main "todolist" pagelet.

The next step is to implement the special *start* event method. This method is not associated with any pagelet and is intended to take the proxy browser from its initial blank state to the point at which the first page of content can be generated. In this case, our event method opens a URL within the proxy browser using a standard JavaScript API call, specifies that the "login" pagelet is next and that the

clip method for that pagelet should be called from the browser's onload event using the Highlight API. The special clipCallback method is provided to every event method to give it control over when the clip method should be called. Note that the event method does not need to check for easily identifiable failures, such as the web site being unavailable or returning a standard HTTP error code. These errors are checked for by the Highlight API's `callOnceOnLoad` method. A more robust version of this event method might have checked for a web-site-specific "server down" message by including additional code to be called during the onload event before calling the clipCallback. We chose not to do so here for brevity.

Start event method:

```
doEvent:function(clipCallback,browser,document,
                 evt,formData)
{
  browser.loadURI('http://jwnichls.backpackit.com/');
  HighlightProxy.setNextPagelet('login');
  HighlightProxy.callOnceOnLoad(browser, clipCallback);
}
```

The next step is to implement the pagelet methods. To show how pagelet content is generated, we will first examine the clip method for the login pagelet. This method assumes that the proxy browser is now viewing Backpack's login page, and uses XPath expressions to find the appropriate content within the existing page and replicate it for the mobile page. This method also modifies the content slightly by removing a "Remember me" checkbox, which is a web site feature that the Highlight proxy server cannot support (cookies are removed at the end of every session).

Login clip method:

```
doClip:function(document, proxyhost)
{
  // Clip content into a document
  var clipDoc = HighlightProxy.createClipDocument();
  var bodyElem = clipDoc.getElementsByTagName('body')[0];
  var loginPanel =
          HighlightProxy.clipAsDOM(document,'/HTML[1]…');
  bodyElem.appendChild(loginPanel);

  // Remove the "Remember me" checkbox
  HighlightProxy.remove(clipDoc,'/HTML[1]…');

  // Give the forgot password link a custom targetName
  var link = clipDoc.getElementsByTagName('a')[0];
  HighlightProxy.setTargetName(link, 'forgotPassword');

  return loginPanel;
}
```

At the end of the method, you will note that a *targetName* is assigned to the "forgot password" link. The target name property is a mechanism that Highlight provides to authors to help them determine what interaction the mobile user just performed with the mobile page. The target name can be used to influence the subsequent logic of the application and also to identify the corresponding element in the existing web page on the proxy browser. Highlight automatically assigns target names to all interactive elements in the mobile page, but the author can provide custom names to identify particular elements and make their code more readable.

The login pagelet event method is shown below. This method identifies the action the user performed on the mobile device: in this case, either clicking the "forgot password" link or the submit button. Because we assigned a custom target name to this link in the clip script, we can use the target name to determine the action. Afterward, the method mimics interactions within the proxy browser to advance the browser to the next page. For the "forgot password" link, this involves simulating a click on the link and specifying that the next clip method should be called from the onload event of the next page.

Login event method:

```
if (evt.targetName == "forgotPassword") {
  var link = HighlightProxy
               .getLinkNodeForTarget(evt.targetName);
  HighlightProxy.clickElement(link);
  HighlightProxy.setNextPagelet('emailedPassword');
  HighlightProxy.callOnceOnLoad(browser, clipCallback);

} else { // the login form was submitted
  HighlightProxy.autoFillInFormFields(
                     browser.contentDocument, formData);
  var submitButton = browser.contentDocument
                       .getElementsByTagName('input')[4];
  HighlightProxy.clickElement(submitButton);

  HighlightProxy.callOnceOnLoad(browser, function() {

    if (browser.contentDocument.location.toString() ==
        "http://jwnichls.backpackit.com/login") {
      // login failed
      HighlightProxy.setNextPagelet('login');
      clipCallback();
    } else {
      // login succeeded
      var addItemLink = browser.contentDocument
        .getElementById('link_to_add_child_list_1327418')
        .getElementsByTagName('a')[0];
      HighlightProxy.clickElement(addItemLink);
      HighlightProxy.setNextPagelet('todolist');
      HighlightProxy.callAfterDelay(1000,
                                    clipCallback);
    }
  });
}
```

The most interesting aspect of the method occurs when the form is submitted. To advance to the next page, we fill in the form fields based on the submitted information and simulate a click on the submit button. After loading the next page, the event method must check to see if login was successful. In this case, a simple method is checking the URL of the current page. If we are still on the login page, then we stay on the login pagelet and re-clip. Otherwise, we assume that we are on the application's main page. The initial version of the main page does not have a text box for adding a new item to our list. To ensure that this text box will appear in the clipped version of our page, the event method simulates a click on the "Add an item" link for our to-do list. Note that the number in the id of the click element does not change across multiple sessions. After a short delay that allows the text box to appear, we proceed with clipping the "todolist" pagelet.

The "todolist" clip method is shown below. The list, its "add item" text box, and the "add item" submit button are conveniently located within a element that has a constant id across all sessions, so this element can be clipped and sent to the mobile device as content. Two modifications must be made however. Most importantly, clicking checkboxes in Backpack is handled by JavaScript code, so there is no submit button for these checkboxes. This approach will not work for our simple mobile application, so we must add a

submit button to update the checkbox values with the proxy server. To differentiate from the "add item" button, we give each a different target name through the Highlight API. The final call to `cleanUpForms` ensures that Highlight has a mapping between all of the input fields in the pagelet and the page in the proxy browser, so that the appropriate information can be updated when the mobile form is submitted.

Todolist clip method:

```
var clipNode = HighlightProxy.clipAsDOMByID(document,
                                            'list_1327418');

// update the targetName on the add button
var addItemButton = HighlightProxy
                    .findSubmitButtonWithLabel(clipNode,
                                               "Add this item");
HighlightProxy.setTargetName(addItemButton, "addItem");

// add a submit button to update the items
var updateButton = HighlightProxy.createSubmitButton(
          document,"Update To Do Items", "updateItems");
clipNode.appendChild(updateButton);

// clean up any form mess
clipNode = HighlightProxy.cleanUpForms(document,
                                       clipNode);
return clipNode;
```

The event method for the to-do list pagelet checks the target name parameter to determine which button was pressed, and then takes the appropriate actions within the browser. Note that the `autoFillInFormFields` method dispatches the appropriate events when form fields are entered, which ensures that the appropriate AJAX calls are made when the checkbox values are changed in the proxy browser. We allow a certain amount of time to elapse before clipping so that the AJAX methods can be processed.

Todolist event method:

```
HighlightProxy.autoFillInFormFields(
                     browser.contentDocument, formData);
if (evt.targetName == "addItem") {
  var addButton = HighlightProxy
      .findSubmitButtonWithLabel(browser.contentDocument,
                                 "Add this item");
  HighlightProxy.clickElement(addButton);
}
HighlightProxy.setNextPagelet('todolist');
HighlightProxy.callAfterDelay(1000, clipCallback);
```

EVALUATION

We have evaluated the Highlight system in two ways: 1) to show breadth we created a number of mobile applications for a variety of different existing web sites, and 2) to show usability we conducted a small user study with web programmers.

Breadth

To date, we have created more than 20 mobile applications from existing sites such as aa.com, amazon.com, ebay.com, fitday.com, google.com, homedepot.com, mapquest.com, sfgate.com, weather.com, several internal IBM intranet sites, and many others. A large portion of these applications were built with the Highlight Designer [17], our end-user authoring environment. A few applications, such as one for the internal IBM corporate directory (called BluePages, see Figure 1c), were created initially in the authoring environment and then modified by hand to support additional features.

Application Name	# Ps	Lines	Ave. Lines/ Pagelet	Ave. Clip Lines/ Pagelet
AA Flight Tracker*	3	112	35.33	22.33
Traffic #1	3	58	18.33	15.67
Traffic #2*	1	33	26.00	26.00
Facebook Status	4	59	14.75	7.50
BackPack To-Do	3	46	14.33	5.67
BlueMail	3	127	36.33	22.00
AEO Store Locator*	2	81	37.00	27.50
BluePages*	5	174	69.40	44.80
Google Image Search*	3	130	41.00	27.00
Home Depot Store Finder*	2	73	33.00	24.00

Table 1. Breakdown of lines of code for a selection of mobile applications. # Ps is the number of pagelets in the mobile app. An * following the name of an application indicates that some portion of the code was automatically generated by our end-user authoring environment. The start pagelet code is not included in the average lines/pagelet calculation.

Table 1 shows the lines of code needed to build a selection of these mobile applications and a breakdown of the average number of lines to define each pagelet and the number of lines used in just the clip method of each pagelet. An important thing to note from this table is that for most applications clipping requires the most code, and the code needed to do clipping has few differences from the code needed to do clipping using other re-authoring approaches. It is also notable that re-authoring AJAX applications requires somewhat more event method code than traditional apps, which reflects the higher complexity of the interactions that the application needs to automate.

One of the most interesting applications that we have created so far is for BlueMail [18], an AJAX web mail client implemented by a team at our research lab. This web site is implemented using the Dojo toolkit [2], comprising at least 198 Dojo widgets. All interaction with the BlueMail client takes place inside of a single web page at one URL, even though some views of the application look substantially different (see Figure 1a). The interesting aspect of this application is that it makes use of the Dojo toolkit's programming interface, which is available through JavaScript on the web page, to find content within the page to clip and events that notify the system when it is time to clip. Unlike the example Backpack application, our BlueMail application does not have to use arbitrary delays to wait for content to be updated. The BlueMail application so far is still just a proof-of-concept for using AJAX toolkits and supports just a small subset of BlueMail's features: logging in, viewing the inbox, and viewing a message (see Figure 1b).

Usability

In order to get some initial feedback on the design of our system and the usability of the API, we invited two experienced web programmers from within our research lab to sit down with the system and attempt to build a simple mobile application for an AJAX web site of their choosing. Ses-

sions with both programmers lasted approximately 3 hours, with the first hour being an introduction to the system through a written developer guide and a verbal presentation. Following the introduction, the programmers were both allowed to starting writing their application and use whatever tools they found helpful in order to build their application. Beyond a text editor, the only other tool either of these programmers chose to use was a Firefox browser with the Firebug extension to extract XPath formulas for nodes in various web pages. One programmer chose to build a traffic report application that would be useful for his ride home, and the other chose to implement a status modification tool for facebook (the latter can be seen in Figure 1d).

The programmers each experienced the typical problems associated with learning any new API, such as finding the correct method to use in a particular situation. They also had some trouble adapting to the idea of automating the user interface of the web site, as opposed to using more common approaches. Both spent more time than necessary to understand the inner workings of the two web sites, and most of this knowledge was not used in the final solutions.

Each of the programmers had a different initial conceptual problem with the Highlight system. One expected that we would use a more traditional page-based model for specifying the transformations between the existing site and the mobile site, and he did not initially understand the benefit of the remote control metaphor. The other programmer noted that our model of grouping a clip method and an event method within a pagelet was the opposite of the way he typically thought about developing web pages. We are considering incorporating his suggested improvements into a future version of the Highlight system.

COMPARING CODE INJECTION AND HIGHLIGHT

Another means of re-authoring web sites is to inject JavaScript code into a web page, either on the browser or through a proxy server, and use this code to perform the modifications. Code injection, like Highlight, can re-author sites that include dynamic JavaScript and AJAX.

Code injection modifies the existing page directly, so the author of the injected code must be careful to not make modifications that will break the existing page. Changing the structure of the document, either by adding, removing or moving elements, can easily break any JavaScript in the existing page. Thus, code injection techniques seem to work best for making small modifications to an existing page, such as adding an extra piece of information or providing a different interaction for an existing feature. An informal survey of the Greasemonkey scripts on userscripts.org seems to provide some evidence for this claim. Highlight, in contrast, seems to work best for making large modifications to a web page.

It is important to note that writing Highlight apps and Greasemonkey scripts share many of the same challenges. Both systems are built on top of the same standard web APIs for manipulating the DOM and receiving events from the user. Finding the appropriate content to use and identifying the correct set of events to listen to are generally difficult, and both approaches use the same techniques to solve these problems.

DISCUSSION AND FUTURE WORK

We have shown that Highlight can be used by programmers to create mobile applications from existing web sites and deploy those applications to mobile devices. The example applications that we have shown here demonstrate a common use of Highlight, which is to modify sites that use JavaScript heavily into simpler sites that no longer have any dynamic features. It is possible to create mobile applications that include JavaScript however, and even to have those mobile applications make AJAX requests back to the Highlight server. The Highlight programmer can add this JavaScript either by copying it from the existing page, in which case they must ensure that the code will continue to work on the modified mobile page, or they can write their own new code and include it in the page. Adding such code is not easy, however it is not any more difficult than adding JavaScript and/or AJAX to an existing page. In the future, we hope to explore mechanisms that can automate the transfer of code from the existing page to the mobile page without programmer intervention. The dynamic nature of JavaScript and the lack of a type system will make this quite challenging.

Building a Highlight application requires the programmer to face several challenges: robustness to errors occurring in the existing site, and brittleness of the application to web site changes. These problems also exist for most other systems that re-author or extract web page content, and we do not provide any new support for addressing them. A generic solution that may work for Highlight could be to show the rendered page to the user when an error occurs and allow the user to indicate how to proceed, perhaps by indicating to where the desired element has moved. Highlight mobile application descriptions are written in JavaScript, however. Work will be needed to understand how to translate corrections identified by the user into changes in the underlying code.

Security and privacy are also important issues for Highlight, because all of the user's interactions with a Highlight application are being replicated in a browser on a proxy server. This creates an additional security hole for potentially sensitive information. Some of these issues are addressed by explicitly separating the browser instances for every session. To the extent that the browser itself is secure, this prevents a malicious application from crossing the web page boundary and snooping on others. Another means to ensure privacy is to wipe a browser instance from the proxy server once its session is complete, including any cached HTML, cookies, and images. Note that removing cookies after a session does not break any web sites, though it does require users to log in every time they use a mobile application. This problem could be addressed by having users log in to the Highlight server and maintaining

cookies for each user, however we have not implemented this. Security between the proxy server and the mobile device is also an issue. We currently do not use SSL on this channel, though this could be added in the future.

CONCLUSION

We have explored a new approach to "mobilizing" the web through re-authoring existing sites using the Highlight system. An important innovation in Highlight is that it allows web sites that make use of dynamic JavaScript and AJAX to be re-authored to support smaller, task-specific interactions. This is accomplished by embedding a full-featured web browser inside of a proxy server and using a remote-control metaphor wherein the mobile browser controls the proxy browser. We have shown that useful mobile applications can be constructed using this approach.

ACKNOWLEDGMENTS

Our thanks to the USER group at the IBM Almaden Research Center, our user study subjects, and especially the CoScripter and BlueMail project members, for their feedback on this work.

REFERENCES

1. "Backpack," 2007. http://www.backpackit.com/.
2. "Dojo: The JavaScript Toolkit," 2007. http://dojotoolkit.org/.
3. "Greasemonkey Firefox Extension," 2007. http://www.greasespot.net/.
4. Aptana, "Aptana Jaxer," 2008. http://www.aptana.com/jaxer.
5. Bickmore, T.W. and Schilit, B.N. "Digestor: Device-independent Access to the World Wide Web," in *Selected papers from the sixth international conference on World Wide Web.* 1997. Santa Clara, CA: pp. 1075-1082.
6. Bigham, J.P. and Ladner, R.E. "Accessmonkey: A Collaborative Scripting Framework for Web Users and Developers," in *Proceedings of W4A.* 2007. Banff, Canada: pp. 25-34.
7. Bolin, M., Webber, M., Rha, P., Wilson, T., and Miller, R.C. "Automation and Customization of Rendered Web Pages," in *Proceedings of UIST.* 2005. Seattle, WA: pp. 163-172.
8. Clark, J. and DeRose, S., "XML Path Language (XPath), Version 1.0," 1999. http://www.w3.org/TR/xpath.
9. Faaborg, A., *A Goal-Oriented User Interface for Personalized Semantic Search.* Media Arts and Sciences Massachusetts Institute of Technology, 2006, Boston. http://alumni.media.mit.edu/~faaborg/files/thesis/draft/complete/faaborg_thesis.pdf.
10. Google, I., "Google Mobile Search," 2007. http://www.google.com/mobile/.
11. Greenlight, "Skweezer," 2007. http://www.skweezer.net/.
12. Hartmann, B., Wu, L., Collins, K., and Klemmer, S.R. "Programming by a Sample: Rapidly Creating Web Applications with d.mix," in *Proceedings of the 20th annual ACM symposium on User interface software and technology.* 2007. Newport, RI: pp. 241-250.
13. Hwang, Y., Kim, J., and Seo, E., "Structure-Aware Web Transcoding for Mobile Devices." *IEEE Internet Computing*, 2003. **7**(5): pp. 14-21.
14. Little, G., Lau, T., Cypher, A., Lin, J., Haber, E.M., and Kandogan, E. "Koala: Capture, Share, Automate, Personalize Business Processes on the Web," in *Proceedings of CHI.* 2007. San Jose, CA: pp. 943-946.
15. MDA, "Q3 2006 – Mobile Internet Figures Continue To Grow," 2007. http://www.themda.org/PressReleases/Page_Press_PressReleases_LatestStats.asp.
16. Monkeygrease, "Monkeygrease: The Server-side Greasemonkey," 2007. http://www.monkeygrease.org/.
17. Nichols, J. and Lau, T. "Mobilization by Demonstration: Using Traces to Re-author Existing Web Sites," in *Proceedings of IUI.* 2008: pp. 149-158
18. Nusser, S., Cerruti, J., Wilcox, E., Cousins, S., Schoudt, J., and Sancho, S. "Enabling efficient orienteering behavior in webmail clients " in *Proceedings of the 20th annual ACM symposium on User interface software and technology* 2007. Newport, Rhode Island, USA pp. 139-148.
19. OPA, "Going Mobile: An International Study of Content Use and Advertising on the Mobile Web," 2007. http://www.online-publishers.org./media/176_W_opa_going_mobile_report_mar07.pdf.
20. Richardson, T., Stafford-Fraser, Q., Wood, K.R., and Hopper, A., "Virtual Network Computing." *IEEE Internet Computing*, 1998. **2**(1): pp. 33-38.
21. Schwerdtfeger, R.S., Weiss, L.F., and Dutta, R., "Electronic document delivery system employing distributed document object model (DOM) based transcoding and providing interactive javascript support," 2006. International Business Machines Corp.: USA.
22. Stuerzlinger, W., Chapuis, O., Phillips, D., and Roussel, N. "User interface façades: towards fully adaptable user interfaces," in *Proceedings of UIST.* 2006. Montreux, Switzerland: pp. 309-318.
23. Tan, D.S., Meyers, B., and Czerwinski, M. "WinCuts: Manipulating Arbitrary Window Regions for More Effective Use of Screen Space," in *CHI Extended Abstracts.* 2004. pp. 1525-1528.
24. Zhang, D., "Web Content Adaptation for Mobile Handheld Devices." *Communications of the ACM*, 2007. **50**(2): pp. 75-79.