Web Services and Information Delivery for Diverse Environments

Juliana Freire   Bharat Kumar

Bell Laboratories, 600 Mountain Ave., Murray Hill, NJ 07974, USA

{juliana,bharat}@research.bell-labs.com

Abstract

There is a growing need for techniques that provide alternative means to access Web content and services, be it the ability to browse the Web through a voice interface like the PhoneBrowser, or through a wireless PDA or smart phone. The Web was designed and works well for desktop computers, to be viewed in large screens and through *good* network connections. However, using the Web through a phone or a small wireless device poses a number of challenges. In this paper, we discuss the issues involved in making existing Web content and services available for diverse environments, and describe PersonalClipper, a system that allows casual users to easily create customized (and simplified) views of Web sites that are well-suited for different types of terminals.

1 Introduction

The ability to take information, entertainment and e-commerce on the go has a lot of promise. The wireless data market is expected to grow enormously in the next few years. In the US alone, Dataquest expects that the number of wireless data subscribers will explode from 3 million in 1998 to 36 million in 2003. Thus, very soon, millions of people will be able to access the Web, order services and goods from wireless Internet devices. However, the existing Web infrastructure and content were designed for desktop computers and are not well-suited for devices that have less processing power and memory, small screens, and limited input devices. In addition, wireless data networks provide less bandwidth, have high latency and are not as stable as traditional (wired) networks.

Consider for example accessing the Web from a personal digital assistant (PDA) such as the Palm Pilot. Current wireless data services such as Omnisky [14] run over CDPD[1], whose throughput rates vary from 5-6 kpbs up to 12-13 kbps. With a screen size of 160x160 pixels on a 6x6cm surface, it can be very hard to browse through large pages with rich graphics. In addition, input facilities are limited — even with Palm's Grafitti text input system, entering text can be very time consuming.

In order to address these limitations of bandwidth, screen real estate, and input facilities, there are three different approaches/models currently in use:

[1]CDPD [7] is a wireless IP network that overlays on the existing AMPS (analog) cellular infrastructure.

- *Re-engineering existing Web sites:* content providers create different versions of their Web sites that provide content formatted for specific devices. For example: The New York Times has a *palm-friendly* section at http://channel.nytimes.com/partners/palmpilot/index.html; amazon.com provides a specialized interface for Web-enabled phones, as well as for the Palm VII [2]; and various other Web sites now have mobile phone-friendly versions (see [22] for a list such sites).
- *Creating specialized wrappers that export a different view of a Web page or service:* third-party services such as byair.com and oraclemobile.com provide wrappers which export wireless-friendly clippings of a set of Web pages and services, such as stock quotes, traffic and weather information. These wrappers require no modifications to the underlying Web sites.
- *Using proxies that filter and reformat Web content:* proxies can be programmed to transform content according to client's display size and capabilities. For example, ProxiWeb [18] transforms HTML pages and embedded figures into a format that can be displayed on a Palm Pilot.

But these approaches have drawbacks (Table 1 summarizes the features of these approaches). From a content provider's perspective, having to create and maintain multiple versions of a Web site to support different devices is labor intensive and can be very expensive. The same is true of specialized Web clippings, as programs (or wrappers) have to be created for each Web site and these need to be updated every time the underlying site changes. From a user's point of view, both solutions are restrictive, as neither all Web sites support all kinds of devices, nor wrapper-based solutions offer clippings for all content or services a user may need.

Proxy transcoders, on the other hand, perform on-the-fly content translation and thus, are a good general solution for allowing users to browse virtually any Web site. The kind of translation done by these proxies include reduction of image resolution, modification of HTML constructs that can not be effectively viewed in smaller screens (*e.g.,* ProxiWeb rewrites pages that contain frames so that they display the links corresponding to the frames), and translation from HTML to other languages such as the wireless markup language (WML) [25]. But since Web pages must be presented as faithfully as possible, these general purpose proxies do not perform any personalization — Web pages are always displayed in their entirety. This is clearly not the ideal solution for somebody accessing the Web through a cellular phone with a 3-line display. Besides, some features are hard and sometimes impossible to translate. For example, existing browsers for the Palm Pilot do not support JavaScript and thus, it is not possible to guarantee that pages with JavaScript will behave correctly in these browsers. It is often the case that proxies are not able to transcode complex pages.

In this paper, we describe the PersonalClipper system. PersonalClipper provides a platform that allows end-users to easily create and maintain *personalized* clippings of Web sites. These Web clippings are shortcuts to content and services a user (or a group of users) is interested in, such as the CNN health headlines, weather information for a specific city, flight information from Travelocity, or one's bank balance. By allowing users to create their own Web clippings, a service can be offered that is *personalized* and not restricted to a set of supported Web sites, and users can easily customize such clippings for specific devices.

	multi-version sites	wrapped services	transcoding proxies	PersonalClipper
creation cost	high	high	n/a	low
maintenance	high	high	n/a	low
personalization	limited	limited	none	high
coverage	low	low	medium	high

Table 1: Summary of delivery techniques

The process of creating clippings is quite simple: it requires *no programming expertise*, and can be done by casual Web users. Furthermore, PersonalClipper generates clippings that are *robust* to certain changes to Web sites, and thus the need for maintenance is reduced. Unlike other systems for creating personal portals (*e.g.*, Portal-to-Go [15], ezlogin [9]), the PersonalClipper system offers *privacy*: clipping creation (and retrieval) can be done from the user's machine, without the intervention of a third-party server.

The structure of the paper is as follows. We start in Section 2 with a motivating example. In Section 3 we describe the PersonalClipper system, its methodology and architecture. Section 4 describes how the PersonalClipper can be used to create *views* of pages and services that are well-suited to different types of devices. Related work is discussed in Section 5. We conclude in Section 6 with some future directions.

2 Motivating Example

Consider the following scenario. Juliana plans to attend the VLDB conference and she is looking for flights from JFK to Cairo that leave from JFK on September 9th, and return from Cairo on September 16th. She must take the following steps:

- Go to http://www.travelocity.com
- Choose the *Find/Book a Flight* option,
- Enter the login information,
- Choose the *9 Best Itineraries* option,
- Specify details of itinerary.

This series of steps (depicted in Figure 1) produces a page with a list of alternative flights. Now, if she wants to do this from her Palm Pilot through a wireless modem, there are some problems:[2]

- *many interactions are needed to access the flights page:* Given the high latency and low-bandwidth of wireless data services, performing all these steps through a wireless modem or on a cellular phone can be hard (especially if a significant amount of information needs to be input), very time consuming, and sometimes impossible (*e.g.,* certain pages require JavaScript which is not supported by *micro* browsers, such as ProxiWeb [18] and AvantGo [5]).

[2] In fact, we were not able to access the Travelocity site using either the ProxiWeb [18] browser, which was not able to retrieve even the initial page, or AvantGo [5], which does not support secure connections.

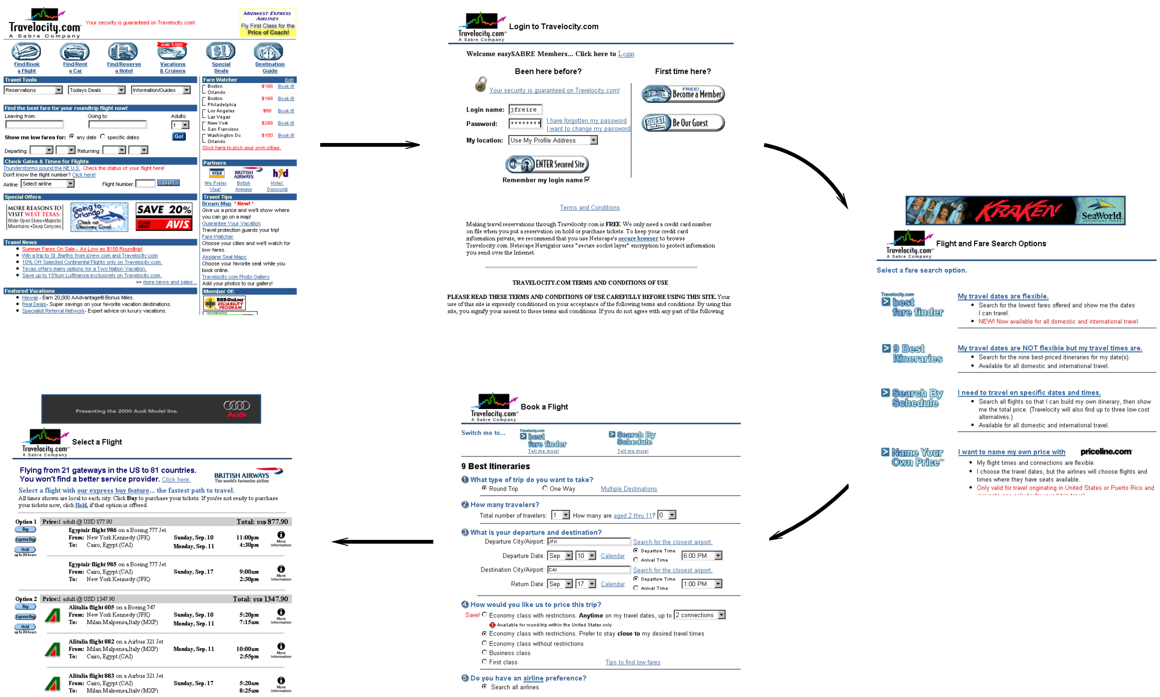

Figure 1: Sequence of steps to retrieve flight itineraries from travelocity.com

http://webclipserver.com/jfreire?clipping=travelcairo&mode=pull

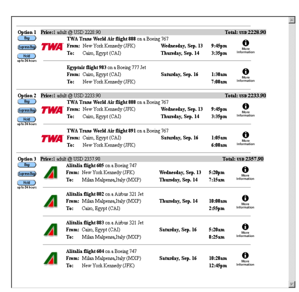

Figure 2: Retrieving flight itineraries from travelocity.com using the PersonalClipper server

- *irrelevant information is downloaded:* Most times, one is only interested in a subset of the information presented in a Web page. In this example, not only the user must download a series of intermediate pages, but she must also download the whole *Flights* page even though she might only need, say, the first three itineraries. Being able to access *only* the desired information is especially important in the wireless environment where bandwidth is scarce and expensive, and screen space is limited.

If we consider the example above, the ideal would be to create a *shortcut* that gives a one-click access to the first three itineraries (as shown in Figure 2). In general, it would be useful if one could *easily* create not only simple shortcuts, but also different *views* of Web sites that are better suited to be accessed from different terminals. In the Palm Pilot scenario, it would be useful to reduce the number of required interactions, and the amount of data input and transferred. For example, one could create a clipping template for Travelocity that would automatically login, and always fill in the departing city and preferred airline with default values, and require from the user just the travel dates and destination.

3 The PersonalClipper

In this section we describe the PersonalClipper system and its architecture, and discuss the main issues involved in creating and accessing clippings.

There are two steps involved in creating Web clippings: retrieving a Web page, and extracting elements from a retrieved page. Given the growing trend of interactive Web sites that publish data on demand, retrieving the information from the Web is becoming increasingly complicated. Many sites, from online classified ads to banks, require users to fill a sequence of forms and/or follow a sequence of links to access a page they need, and often, these hard-to-reach pages cannot be bookmarked using the bookmark facilities implemented in popular browsers. In order to create clippings of these pages, the process to access them must be automated. Also, as described in the example of Section 2, once the desired page is retrieved, a user may want to specify individual elements of the page she is interested in, so that irrelevant information is filtered out. A Web clipping thus must encapsulate the actions required to retrieve a particular page, and the specification of which elements should be extracted from that page.

It is possible to automate the retrieval of pages by writing programs in Java or in more specialized languages such as WebL [11]. One can also write Perl scripts to extract individual fragments of Web pages. However, this approach is not feasible for casual Web users that are not programmers. In addition, given the dynamic nature of the Web, maintaining these programs and scripts can be very costly, as they might require modifications every time Web sites change.

The PersonalClipper addresses these problems by providing a VCR-style interface similar to the WebVCR [3] to transparently record browsing steps; and a point-and-click interface to let users select page fragments. Furthermore, the system uses techniques that enhance the robustness of clippings, so that they *work* even if certain changes occur in the underlying Web sites.

After a clipping is created, it can be accessed through a PersonalClipper server, that may be located at

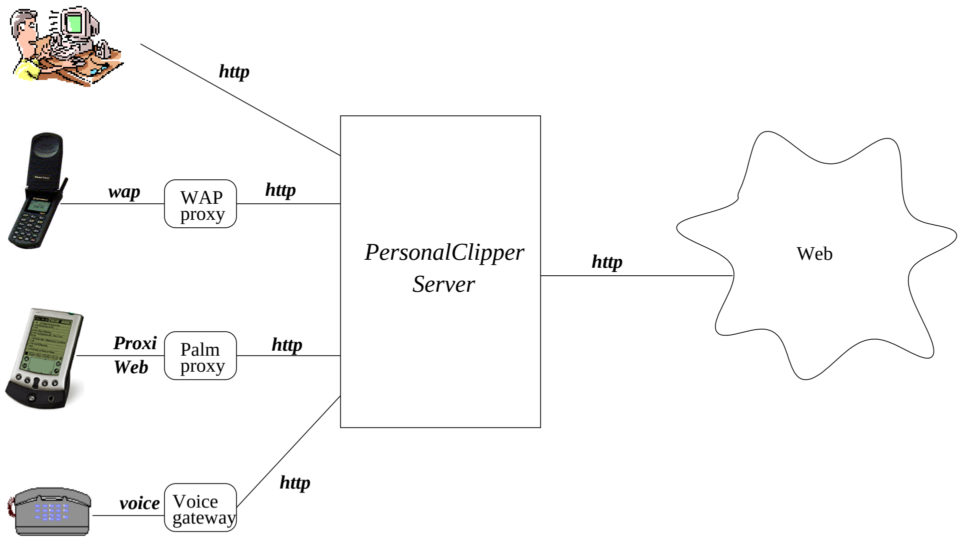

Figure 3: Accessing Web Clippings

a user's machine, at a service provider, or inside an Intranet. As Figure 3 shows, the PersonalClipper is a Web service that accepts requests from HTTP clients. A request to the PersonalClipper server contains an identifier for a particular clipping[3], which when executed, accesses a particular Web page, clips it, and returns the resulting (clipped) page to the requesting client.

The architecture of the PersonalClipper server is shown in Figure 4. The PersonalClipper server consists of the following modules: 1) the clipping DB, which stores clipping specifications; 2) the user profile manager, that performs user authentication for sensitive clippings stored on the server (*e.g.,* a clipping that retrieves a user's 401(k) balance); 3) the clipping scheduler, that periodically executes clippings (if so specified by the clipping creator); 4) the cache manager, that stores cached clippings; and 5) the clipping execution engine, that interacts with an HTML parser, Javascript interpretor, and HTML content extractor, to execute specified clippings.

In what follows we give a more detailed description of clipping creation and execution. For ease of presentation, we restrict our discussion to the scenario where the PersonalClipper server is hosted as a Web-based service that a user can access using a Java-enabled Web browser.

3.1 Creating Web Clippings

Web clippings have two components: retrieval and extraction. As depicted in Figure 4, the PersonalClipper provides applets for both tasks: the *recording applet* and the *extraction applet.*

In order to create a clipping, a user must first specify the page to be clipped. If the page requires multiple steps to be retrieved and does not have a well-defined URL, the user can use the recording

[3] As will be described in Section 3.2, requests may also include other parameters such as input values for the clipping.

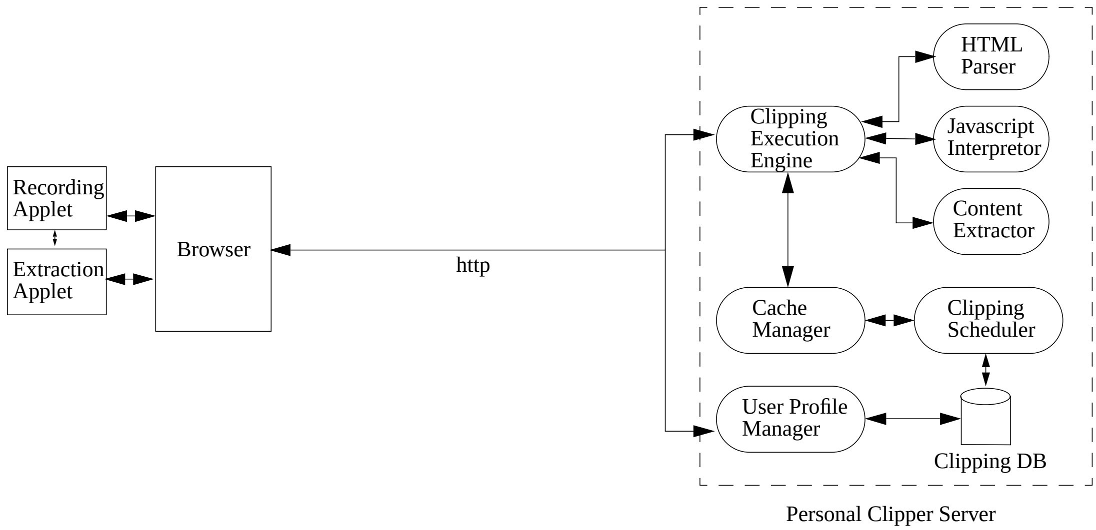

Figure 4: PersonalClipper Server Architecture

applet to create the script to access the page. The recoding applet is a variant of the WebVCR [3]. It has a VCR-style interface to record browsing actions. When the user clicks the *record* button on the applet, she is prompted to input the URL for starting page which is then loaded into a browser window. From this point on, the applet monitors all user actions in that browser window[4]. This monitoring is transparent to the user, who can simply navigate her way to the final page as usual. When the user reaches the final page, the sequence of recorded actions (*i.e.,* links traversed, forms filled along with the user inputs, and any other interactions with active content on the page[5]) is saved.

During the recording process, if the user is required to fill out forms, she can optionally specify which field values are to be stored in the clipping specification itself, and which ones are to be requested from the user every time the clipping is executed. This allows the user to create parameterized clippings. For example, a clipping to retrieve stock quote information from `quicken.com` can have as a parameter the ticker symbol, so that the user does not need to create a separate clipping for each stock. Also, for security reasons, a user may choose to not to save certain kinds of information such passwords inside a clipping, or to save it encrypted.

In contrast with the current practice of writing wrapper programs (*e.g.,* using languages such as WebL [11] or Java), the PersonalClipper system offers an alternative to quickly and easily create access wrappers/scripts that requires no programming — creating and updating these wrappers is a simple process involving only the usual browsing actions.

Once the desired page is retrieved, the extraction applet can be used to specify the fragments of the

[4]The applet adds Javascript event handlers to all *active* elements on the page, and when an event fires, it records the corresponding action. For more details on the monitoring process, the reader is referred to [3].

[5]Note however that this is currently restricted to handling Javascript, and not arbitrary active content such as applets and plugins on a page.

page that should be extracted. An interesting problem is how to identify these fragments. In general, any extraction specification chosen needs to provide the ability to 1) address individual or groups of arbitrary elements in a page, and 2) specify rules (that use the above addressing scheme) to extract the relevant content from the page. We wanted a solution that was standard, powerful, portable and efficient.

Our first choice was to use the DOM API [8] to specify extraction expressions. However this API is rather limited, *e.g.*, it does not allow the retrieval of tables from an HTML document. We found XPath [26] to be a better, more flexible addressing scheme than the DOM API. XPath views XML documents as a tree and provides a mechanism for addressing any node in this tree. One drawback of using XPath is that it requires pages to be well-formed. Since browsers are very forgiving in this respect, many Web sites generate pages that are ill-formed (*e.g.*, have overlapping tags, missing end tags, etc.). Consequently, the PersonalClipper system must first *clean up* HTML pages (*e.g.*, using tools such as HTML Tidy [10]) before users input XPath expressions over the document to specify the desired content.

The XPath expressions below extract the first three itineraries (each itinerary is represented by an HTML table) from the flight selection page of the Travelocity example of Section 2.

$$\begin{aligned} //html/body/center[2]/table[2] \mid \\ //html/body/center[2]/table[3] \mid \\ //html/body/center[2]/table[4] \end{aligned} \tag{1}$$

$$//html/body/center[2]/table[position() < 5 \; and \; position() > 1] \tag{2}$$

$$//table[contains(string(),'price') \; and \; position() < 5] \tag{3}$$

These expressions can be rather complicated, and writing them can be an involved task. In addition, there are multiple ways to specify a particular page element, and some may be preferable to others (as explained in Section 3.3). To address these problems, we are currently designing a point-and-click interface that lets users select portions of Web pages (as she sees them in the Web browser), and it automatically generates extraction expressions. The point-and-click interface will provide users with different levels of abstraction that correspond to a breadth-first search in the portion of the document tree that is visible in the browser. For example, if a user is interested in particular cells of a table, he must first select the table and then, zoom into the table to select the desired cells.

Figure 5 illustrates a Web clipping specification (simplified for exposition purposes) for the Travelocity example described in Section 2. The first part of a clipping specification corresponds to a sequence of browsing steps (*i.e.*, $< URL >$, $< LINK >$, and $< FORM >$). The $< EXTRACT >$ elements contain the extraction specifications. Note that multiple fragments can be specified, and users may choose to specify these fragments according to the terminal where they will be displayed. For example, if the clipping is to be displayed in a Palm Pilot, the user could choose to extract the first 3 itineraries (the extraction tag with $fragment\_name = 'first\_3\_itineraries'$), whereas if the clipping is to be displayed in a Web-enabled cellular phone with a 3-line display, a single itinerary may be preferable (*e.g.,* the extraction tag with $fragment\_name = 'first\_itinerary'$).

Given the unpredictable behavior of the Web (network delays, unreachable sites, etc.), caching plays an important role in a PersonalClipper server. Users can specify for each clipping, if and how often it should be executed (*e.g.,* weather information from for a user's home town should be refreshed every 6 hours) and cached.

3.2 Executing Clippings

After a clipping is specified, it can be saved, and uploaded to a PersonalClipper server. Users may then access clippings via URLs that uniquely identify them. Users may further specify additional parameters such as input values for clipping (*e.g.,* the password to access a bank account); the mode of operation (pull or push); whether the clipping should be cached; and how often it should be refreshed.

In the *pull mode*, the URL invokes a CGI script at the server, which in turn executes the clipping specification and immediately returns the clipped content to the requesting client. In the *push mode*, the execution and delivery of the clipping are asynchronous, *i.e.,* the clipping can be returned to the client later, possibly through protocols other than HTTP (*e.g.,* clippings could be emailed to users). The push mode is preferable when back-end Web sites are slow or temporarily unreachable, or when the end user cannot or does not want to keep a session open for too long[6].

The clipping execution is as follows. The page corresponding to the starting URL is fetched and parsed. The user actions are then executed in sequence, some of which might cause new Web pages to be fetched. For example, link traversals are executed by fetching the corresponding URL; form submissions are executed by first filling the form fields with the recorded user inputs, and then submitting the form; and if there are any Javascript event handlers on elements of the page the user has interacted with, such actions are filtered through the Javascript interpretor to ensure that the same handlers fire during replay.

After the final page has been retrieved (and cleaned), the extraction expressions are evaluated to extract the desired content. This can be done with an XSLT [27] interpretor such as XT[7]. The extracted content is then returned to the client.

Note that all processing (retrieval and extraction) is done at the PersonalClipper server. Only select portions of Web pages are returned to the requesting client, effectively giving users a one-click access to desired content, and considerably reducing the communication between the client and the PersonalClipper server. This feature is specially useful in wireless environments where users have to access the Web through high latency and low-bandwidth connections.

Since Web pages may change between record and replay, the PersonalClipper uses techniques to ensure that replaying a sequence of recorded actions will lead to the intended page, and that the correct

[6]Some wireless services, such as Sprint PCS, charge for usage time.

[7]XT is available at http://www.jclark.com/xml/xt.html.

```
 < URL > http : //travelocity.com < /URL >
< LINK >
   < text > null < /text >
   < href > http : //dps1.travelocity.com/lognlogin.ctl?tr_module =
   AIRG&SEQ = 1 < /href >
   < target > null < /target >
   < loc > opener.document.links[8] < /loc >
< /LINK >
< FORM ><! − −Loginform − − >< /FORM >
< LINK ><! − −9BestItinerarieslink − − >< /LINK >
< FORM >
   < name >< /name >
   < method > post < method >
   < target >< /target >
   < action > https : //dps1.travelocity.com : 443/lognmain.ctl?
   SEQ = 1 < /action >
   < loc > opener.document.forms[0] < /loc >
   < ATTRS >
      ...
      < ATTR >< name > trip_option < /name >< loc > 5 < /loc >
      < type > radio < /type >< prop > stored < /prop >
      < val > roundtrp < /val >< /ATTR >
      ...
      < ATTR >< name > depart_airport < /name >< loc > 10 < /loc >
      < type > text < /type >< prop > stored < /prop >
      < val > JFK < /val >< /ATTR >
      < ATTR >< name > depart_month < /name >< loc > 11 < /loc >
      < type > select − one < /type >< prop > stored < /prop >
      < selected_index > 8 < / >< text > Sep < /text >< /ATTR >
      < ATTR >< name > depart_day < /name >< loc > 12 < /loc >
      < type > select − one < /type >< prop > stored < /prop >
      < selected_index > 12 < / >< text > 13 < /text >< /ATTR >
      ...
   < /ATTRS >
< /FORM >
< EXTRACT fragment_name =′ first_3_itineraries′ >
   //table[contains(string(),′price′) and position() < 5]
< /EXTRACT >
< EXTRACT fragment_name =′ first_itinerary′ >
   //table[contains(string(),′price′) and position() < 2]
< /EXTRACT >
```

Figure 5: Web Clipping for retrieving the itineraries from http://www.travelocity.com

fragments are extracted — even when the underlying pages are modified.

3.3 Robustness Issues

Usually, changes to Web pages do not pose problems to a user browsing the Web, but they do present a challenge to a system that performs automatic navigation. In a sequence of recorded browing actions, some links may contain embedded session ids, and forms may contain hidden elements that change from one interaction to the next. Thus, for each user action during replay, the PersonalClipper system must locate the correct object (link, form or button) to be operated on, and this can be challenging in the presence of changes to Web pages (*e.g.*, such as addition/removal of banner ads).

To ensure that clippings execute properly and retrieve the intended page, *enough* information must be saved for each action. For example, for a link traversal the PersonalClipper saves: the DOM address of the link, its text and URL. During replay, if an exact match for the link cannot be found in a page, heuristics are used that try and find the *closest match* for it. For a more detailed discussion on the heuristics used, see [3]. Note however that if the page structure changes radically, these heuristics may fail, in which case the clipping will need to be re-recorded.

Extraction expressions can also be made robust to changes to Web pages. For example, in the XPath expression (1) above, if a new `center` tag is added to the document, the expression will no longer retrieve the correct tables. Besides the index of the particular node to be extracted, the specification may include extra information that helps the system identify certain elements if the indices happen to change. For instance, the XPath expression (3) specifies tables with an index less than 5 and that contain the "price" string — this expression would still retrieve the correct itineraries even if new `center` tags are added. Robustness can also be improved by adding redundancy in the specification, for example, the path from the root of the document to the element, and some contextual information (such as surrounding text) [16].

4 Delivering Clippings to Diverse Terminals

The PersonalClipper functions as a Web service, and as Figure 3 shows, the destination for the clipped content can be any user-agent that understands HTTP (*e.g.*, a browser on a user's desktop). The PersonalClipper platform can thus be used to create personal portals like `mynetscape.com` that puts together Web clippings with information from various Web sites, and that users can access from their Web browsers with a single-click. The PersonalClipper can also be used in conjunction with other gateways and transcoding proxies to provide content to devices that do not handle HTTP/HTML, for example, it can be used together with a WAP gateway.

There are many benefits to using the PersonalClipper for delivering information to diverse terminals. By offloading all processing and most network communication to a server, it fits well the thin-client architecture used for wireless devices. In addition, by customizing and filtering content, it can significantly simplify Web pages, making the job of transcoding proxies a lot easier. In this section, we examine in more detail some of the issues involved in using the PersonalClipper in conjunction with transcoding proxies. For simplicity, we focus on WAP proxies.

The Wireless Application Protocol (WAP) is based on a 3-tier architecture where the central component, the *gateway*, is responsible for encoding and decoding requests from wireless devices to Web servers and vice-versa. As Figure 3 illustrates, as a user browses the Web through a Web-enabled cellular phone, requests are sent to a WAP gateway. The WAP gateway decodes and executes the requests (*e.g.*, a URL fetch). When the requested document is retrieved from the Web, it is translated into WML (Wireless Markup Language), appropriately encoded, and returned to the phone. Since WAP gateways *talk* HTTP and HTML, it is straightforward to use any existing WAP gateway together with a PersonalClipper server.

WAP provides a push framework [23] that can be used in conjunction with the PersonalClipper push mode to provide batch/asynchronous content retrieval. The usage scenario is as follows. An end user requests a clipping from a PersonalClipper server by specifying its URL and optionally a set of parameters (*e.g.*, the frequency of push). The PersonalClipper server would then act as a push initiator, periodically retrieving and filtering the specified content, and pushing it to the user's device via a push proxy gateway. Notification services could also be built using the push mechanism. For example, rules could be added to the clipping specification that dictate under what conditions the clipping should be pushed into the device. For devices that do not support a push framework, different mechanisms may be used: specialized servers/gateways could be layered on top of the PersonalClipper to send information to pagers, email addresses, or convert content to speech and send it to a voice mailbox.

To enable secure e-commerce services, and to allow end users to access sensitive information (*e.g.*, 401(k) balance), there needs to be a mechanism to provide security between the device and the back-end Web sites. Since the PersonalClipper server executes requests on user's behalf, it is not possible to establish an end-to-end secure connection. The next best scenario is to have two secure connections: one between the device and the PersonalClipper server, and another between PersonalClipper and the back-end Web site. For WAP devices, this requires WTLS (Wireless Transport Level Security) [24] to provide application-level security, rather than secure connections between the user-agent and the WAP gateway only. In this scenario, for devices that cannot handle HTML, the task of transcoding the request/response must be performed at the PersonalClipper server, since a separate WAP gateway would not be able to access the encrypted data flowing from the PersonalClipper server to the WAP device.

Note that tighter coupling between the PersonalClipper and transcoding proxies is possible. In this scenario, the PersonalClipper could be used as a universal server that accepts requests from various devices and returns content formatted according with the type of the device — multi-device clippings could be created that specify how the clipping should be displayed in different devices. A $DEVICE$ tag could be added to clipping specifications (see Figure 5), for example:

$$< \texttt{DEVICE}\ \texttt{type} =' \texttt{cellular\_phone}'\ \texttt{model} =' \texttt{nokia9000}' >$$
$$\quad < \texttt{DISPLAY}\ \texttt{fragment} =' \texttt{first\_itinerary}' / >$$
$$< /\texttt{DEVICE} >$$

The $DEVICE$ tag may also contain information about general capabilities and characteristics of the device, as suggested in the W3C CC/PP note [6].

5 Related Work

The area of information delivery to heterogeneous devices has attracted a lot of attention recently. In the domain of wireless devices such as PDAs and cellular phones, the Wireless Application Protocol (WAP) initiative [25] is working on standard solutions to enable wireless users access to secure, reliable, stateful transaction services via resource constrained portable terminals [12]. The main objectives of the WAP Forum are: to bring Internet content and services to digital cellular phones and other wireless terminals; create a protocol that will work across differing wireless network technologies; enable the creation of content and applications that scale across a very wide range of bearer networks and device types. WAP thus has the potential to enable transport-independent client/server communications sessions from portable devices over wireless links. However, WAP also faces important challenges. WAP is based on a 3-tier architecture where the central component, the *gateway*, is responsible for encoding and decoding requests from wireless devices to Web servers and vice-versa. Given the growing complexity of Web sites (*e.g.,* the presence of scripting languages, dynamic content, malformed content), transcoding can be very hard, and in practice, many pages and services are just not amenable to transcoding and cannot be accessed through WAP. By allowing users to easily customize services and filter out irrelevant content and complex features, the PersonalClipper system greatly simplifies the transcoding process, increasing the Web coverage for WAP devices.

The simplification of transcoding applies to domains other than WAP. The PhoneBrowser [17] provides a programmable platform that gives the general population of Web page authors the means for building Interactive Voice Response (IVR) systems without having to own any IVR equipment. Web IVR applications are currently built using transcoders such as Spyglass' Prism [20]: content providers write Prism scripts to transcode existing pages into versions that more amenable to being read out. The PersonalClipper system can be used as an alternative to automatically generate these scripts without requiring users to write programs.

In the area of information integration, many systems and techniques have been proposed to *wrap* Web sites. Most of the work in this area, though, focuses on extracting structure from semi-structured data (*e.g.,* [4, 1]). The extractor component of the PersonalClipper system is not concerned with *understanding* the structure or discovering the schema of the underlying data, but in providing robust mechanisms to identifying high-level HTML or XML syntactic components (*e.g.,* the first table after a specific string).

The first version of PersonalClipper uses XPath to address specific components to be extracted from Web pages. Other languages could also be used for this purpose, for example WebL [11] or the scheme used by W4F [19]. These languages provide good mechanisms to extract fragments from documents – in some cases, they are easier to use than XPath. However, XPath is a widely accepted standard and there are freely available tools to process XPath expressions.

Recently, there has been a proliferation of personalization systems which offer services that range from notifications about changes to certain Web pages (*e.g.,* Mindit [13]) to the creation of personal portals (*e.g.*, Portal-to-Go [15], ezlogin [9], Yodlee [28]). These systems have some drawbacks, most notably:

- *Limited coverage:* services offer *clippings* for a limited number of sites. For example, Yodlee2Go [28] allows users to check flight info on Expedia, but it does not allow users to access Expedia's rental car services.[8]
- *Lack of privacy:* in order to use these services, users are forced to go through third-party servers that can observe all user interaction (passwords input as well as content retrieved).

The PersonalClipper system addresses both of these problems: it lets users create clippings for virtually any Web site/page; and by placing a PersonalClipper server at a user's machine, it offers complete privacy — clippings can be created and accessed without the need to go through a third-party server. It is worth pointing out that even though the main motivation for PersonalClipper is to provide personalized Web clippings to end-users, the system can also be used by portal services to simplify the creation and maintenance of specialized wrappers.

6 Discussion

The PersonalClipper provides a platform that lets end-users as well as content providers easily create and maintain customized (and simplified) views of Web pages and services. These customized clippings are easy to create (clipping creation requires no programming expertise); require low maintenance (they are robust to certain changes in the underlying Web sites); they are highly customizable; and clippings may be created for virtually any Web site.

When used as a Web service, the PersonalClipper server performs all processing (page retrieval and content extraction) required to construct a clipping, and returns to the requesting client just the final clipping. By reducing the number of interactions required to retrieve hard-to-reach pages, and allowing users to customize clippings so that need for data input is diminished and *only* the desired content is retrieved, the PersonalClipper can be an integral part of a thin-client architecture for content delivery. It can be specially useful in environments where users have to access the Web through high latency and low-bandwidth connections.

Furthermore, in conjunction with transcoding gateways, customized clippings can be used to provide content to devices that do not handle HTTP/HTML, for example, it can be used together with a WAP gateway. An important advantage of using the PersonalClipper in this scenario comes from the customization and content filtering, which can significantly simplify Web pages, making the job of transcoding proxies a lot easier.

Finally, it is worth pointing out that Web clippings contain useful information about the capabilities of Web services, such as for instance the attributes needed to retrieve a certain Web page. It would be interesting to investigate if and how this information can be used to facilitate not only the discovery and selection of specific services, but also the process of combining different services.

[8] Note that even though ezlogin.com [9] offers the option for users to create their own Web clippings, they are not able to create clippings of certain hard-to-reach pages, that for example involve JavaScript actions.

Acknowledgements: The authors thank Jayant Haritsa for useful comments on the first draft of this paper.

References

[1] B. Adelberg. NoDoSe - a tool for semi-automatically extracting structured and semi-structured data from text documents. In *Proc. SIGMOD*, pages 283–294, 1998.

[2] Amazon anywhere. http://www.amazon.com/anywhere.

[3] V. Anupam, J. Freire, B. Kumar, and D. Lieuwen. Automating Web navigation with the WebVCR. In *Proc. of WWW*, pages 503–517, 2000.

[4] N. Ashish and C.A. Knoblock. Wrapper generation for semi-structured internet sources. *SIGMOD Record*, 26(4):8–15, 1997.

[5] Avantgo. http://www.avantgo.com.

[6] HTML Tidy. http://www.w3.org/TR/NOTE-CCPP.

[7] CDPD. http://www.wirelessdata.org/develop/cdpdspec.

[8] DOM. http://www.w3.org/TR/REC-DOM-Level-1.

[9] ezlogin. http://www.ezlogin.com.

[10] HTML Tidy. http://www.w3.org/People/Raggett/tidy.

[11] T. Kistlera and H. Marais. WebL: a programming language for the Web. In *Proc. of WWW*, 1998. http://www.research.digital.com/SRC/WebL/index.html.

[12] James Kobielus. Wireless application protocol. Technical Report v1, The Burton Group, April 2000.

[13] Mind-it.http://www.netmind.com/.

[14] Omnisky. http://www.omnisky.com.

[15] Portal-to-go. http://www.oraclemobile.com.

[16] T. Phelps and R. Wilenski. Robust intra-document locations. In *Proc. of WWW*, pages 105–118, 2000.

[17] Phonebrowser. http://phonebrowser.research.bell-labs.com/.

[18] ProxiWeb. http://www.proxinet.com.

[19] Arnaud Sahuguet and Fabien Azavant. Building light-weight wrappers for legacy web data-sources using W4F. In *Proc. of VLDB*, pages 738–741, 1999.

[20] Spyglass prism. http://www.spyglass.com.

[21] Voicexml. http://www.voicexml.org.

[22] http://www.sprintpcs.com/wireless/wwbrowsing_providers.html.

[23] Wap push architectural overview. http://www.wapforum.org/, November 1999.

[24] Wireless transport layer security specification version 1.1. http://www.wapforum.org, November 1999.

[25] Wireless Application Protocol Forum. *Wireless Application Protocol: The Complete Standard*. Wiley, 1999.

[26] XPath. http://www.w3.org/TR/xpath.

[27] XSLT. http://www.w3.org/TR/xslt.

[28] Yodlee2go. http://www.yodlee.com.