WebBee: An Architecture for Web Accessibility for Mobile Devices

Krian Upatkoon Wenjie Wang Sugih Jamin
Department of Electrical Engineering and Computer Science,
The University of Michigan, Ann Arbor, MI 48109-2122, USA
{*kupapatt, wenjiew, jamin*}*@eecs.umich.edu*

Abstract— **In this paper, we introduce *WebBee*, a client-proxy architecture that combines a web scraping proxy with a Java client to make a platform-independent gateway between small mobile devices, such as mobile phones, and the vast information available on the World Wide Web. The key technology behind WebBee is a web scraping engine that executes "scraping scripts," a domain-specific scripting language we designed for the purpose of fetching web pages and selectively extracting information from them. By transmitting to and displaying only this extracted information on the client device, WebBee gives mobile devices with small displays and low network bandwidth clean and quick access to web content customarily designed for desktop browsers. With WebBee, providers do not need to tailor their contents specifically for mobile devices.**

I. Introduction

Building bridges between the world of small mobile devices and the Internet remains an open area of research. While support for some applications such as e-mail have matured, the ability to serve world-wide-web content on mobile devices have been limited by screen size and network bandwidth. Evidence of small displays can be seen even in the latest devices. High-end phones such as the Nokia 3660 or Nokia N-Gage series [1], with a screen size of 176 x 208 pixels (35 x 41.5 mm), are considered larger than average. NTT Docomo's FOMA SH900i [2] (one of the newer phones on the Japanese market at the time of writing), hand-held PCs such as HP iPAQ pocket PC [3], and Dell Axim [4], all have screen sizes not exceeding 240 x 320 pixels. Given this small space, it may be considered unacceptable to retain all the information that would be shown on a desktop-sized display (generally 1024 x 768 pixels or larger).

Although the adoption of 3G [5] alleviates the network traffic problem in the case of mobile phones, the majority of mobile phone users in the world today use GSM [6], which is rated at 171.2 Kbps [7], but typically has a much slower download speed. For example, in Table I, we averaged the latency and bandwidth measured by wapSpeed.com [8] for three major wireless carriers in the United States, from wapSpeed's latter 25 tests. We found the average latency to be about 3 seconds, and the network bandwidth to be limited to around 20 to 30 Kbps, far less than the theoretical limit. Busse et al. reported an average latency of about 1 second for GPRS and about 275 ms for 3G (UMTS) on an European network [9]. In contrast, a computer user on a broadband connection may experience 3 Mbps downloads at a latency of less than 70 milliseconds.

TABLE I
LATENCY AND BANDWIDTH MEASUREMENT OF GPRS CONNECTIONS FROM WAPSPEED.COM [8] ON 05/29/2004

Carrier	Device	Latency	Bandwidth
Sprint PCS	SCP-8100	3.49 sec	29.7 Kbps
T-Mobile	Nokia N-Gage	3.07 sec	19.9 Kbps
AT&T Wireless	MOT-V600	2.85 sec	28.5 Kbps

Making the vast content and services of the world-wide-web accessible to small mobile devices has traditionally had two solutions: web browsers designed to render desktop-sized pages on small displays, and the use of mobile-friendly markup languages to create sites specifically targeted to the mobile audience.

Mobile web browser renders each web page to fit the limited display size of the target device, even if the web pages were not designed with small displays in mind. A good example of this technology is Opera [10], a web browser available for multiple platforms such as Symbian OS [11] and Palm OS [12]. Another product, *ePAGE* by Picsel [13], enables users to view HTML (and several other file formats) on various mobile devices. *ePAGE* features a "zoom" control system, where users pick a spot of interest on the screen to focus on and zoom into.

One big drawback to the generic browser solution is that the device must still download, at the very least, each web page's HTML file in its entirety. This burdens the limited wireless bandwidth of the device. Moreover, because many mobile phone plans charge consumers by total network traffic, users are unlikely to risk surfing bloated web sites. Another disadvantage of these technologies is that they require large amounts of computational power, and consequently, battery life, two of the most limited resources on mobile devices. For example, we tried viewing the amazon.com web page over GPRS using Opera on a Nokia N-Gage [1], considered to be a computationally high-end phone with a 104 MHz ARM processor. After the page was fully loaded, scrolling was still painfully slow, as the browser had to pause for almost one second after each click of the directional key.

One way to reduce the network demands and alleviate power-consuming processing on the mobile device is to perform the rendering and image shrinking in a proxy [14, 15]. Because many web sites tend to display links, images, and other information intended for desktop displays, the rendering algorithm (regardless of whether it is implemented on the phone or on a proxy) must be extremely sophisticated if it is to automatically eliminate extraneous information in order to shrink the page's rendering size. No matter how sophisticated the renderer, however, there is a limit to how much information a generic browser can remove without having any knowledge of what the user is interested in. For example, a weather web site made for desktop browsing may not only contain the current temperature and daily forecast, but also driving conditions, next week's forecasts, statistics, Doppler radar images, advertisements and so on. Without knowing exactly what the user is interested in, a generic browser would be forced to squeeze all this information into the portable device's display. This tight packing of information makes it hard for the user to find what she really wants to see on the page.

In contrast to the mobile web browser approach, mobile-specific solution requires providers to create content specifically made for mobile devices. Examples are web sites written in mobile markup languages such as WML [16] or NTT Docomo's cHTML for i-mode [17]. While this is a clean solution, it relies completely on the content provider to create these sites; the additional set up and maintenance cost of these sites is multiplied if the provider is to cover the various markup language platforms available.

In short, current mobile browsing solutions face the following issues:

- Screen space is too small for a generic HTML browser
- Generic browser solutions consume much power
- Network bandwidth is slow and costly
- Mobile-specific solutions require effort on the content provider's part, for each platform to be supported

In this paper, we introduce *WebBee*, a client-proxy architecture that combines a web scraping proxy with a Java client to make a platform-independent gateway between small mobile devices and the vast information available on the World Wide Web. The key technology behind WebBee is a web scraping engine that executes "scraping scripts," a domain-specific scripting language we designed for the purpose of fetching web pages and selectively extracting information from them. Because scripts can be customized for each web site, the client will display only information of interest to the end user. In this way, WebBee gives mobile devices with small displays and low network bandwidth clean access to web content originally designed for desktop browsers.

Our prototype implementation of WebBee[1] includes commonly-used services such as directory services and weather forecasts. Trial users have found web access to these services on their mobile phones to take far less time using WebBee than it would using a generic browser like Opera. In addition to the convenience, network traffic is significantly reduced, resulting in cost savings for users with a pay-per-kilobyte Internet access plan. For example, a generic web browser needs to download a total of about 419KB for the full amazon.com web page and to retrieve the price of a book searched by its ISBN. With our WebBee prototype, the mobile client only needs to retrieve about 150 bytes of data. The WebBee mobile client program is compact and computationally inexpensive, expanding Internet accessibility to a large range of mobile devices.

[1]Sample WebBee applications are available for public download at http://webbee.eecs.umich.edu/index.wml (WML) or http://webbee.eecs.umich.edu/ (HTML)

We do not intend for WebBee to be a magic solution that automatically makes *all* of the World Wide Web's sites fit onto devices with small screens and bandwidth; in fact, we have argued that the general solution is not feasible. Some amount of human effort is required for making each site accessible through the WebBee system (namely, the creation of scraping scripts and client user interfaces). We will show later in this paper, however, that the effort needed per site is very little. As a result, we argue that WebBee can be adopted widely to cover almost all the useful web services that one might think to use while on-the-go with a mobile device.

In our design of WebBee we follow the following three design principles. We illustrate their use on the design of a WebBee-based weather lookup application.

1) **The mobile device is the portal**: instead of forcing users to go through a network portal to access each separate web site, we push the applications to the mobile device itself. With WebBee, users have a *Weather* application on their mobile device that they can start to immediately begin receiving weather information; the role of the network is transparent to the users.
2) **Minimize network use, maximize local interactions**: instead of requiring each user to browse a series of web pages just to submit a query for the weather of a city, the weather application residing on the mobile device interacts locally with the user to obtain the necessary information and makes a single query over the network on behalf of the user.
3) **Keep applications simple**: each WebBee application performs only a single purpose. This way, we minimize both user interaction and screen usage.

A faster network such as promised by 3G technologies will not obviate these design principles because screen real-estate on the mobile phone will continue to be limited, and network latency at 250 ms will continue to be an issue.

The rest of this paper is organized as follows. We first give a description of the WebBee architecture in Section II. In Section III, we present the specifics of our prototype implementation. We then discuss some challenges that we face in the design of WebBee in Section IV, and conclude in Section V.

II. WebBee Architecture

A. Overview

At the core of WebBee is a web scraping engine that executes mini-scripts, referred to as "scraping scripts." Web scraping is the process of extracting information from a web page; the scraping scripting language we designed allows for pattern matching to locate the desired information on each web page. Scraping is typically useful for extracting information from sites that have changing content, such as weather reports and news. Combined with user input, scraping can also be used for querying applications such as online dictionary or online auctions.

The scraping functionality of the WebBee architecture is closely related to that of *data-miners*. A data-miner is a

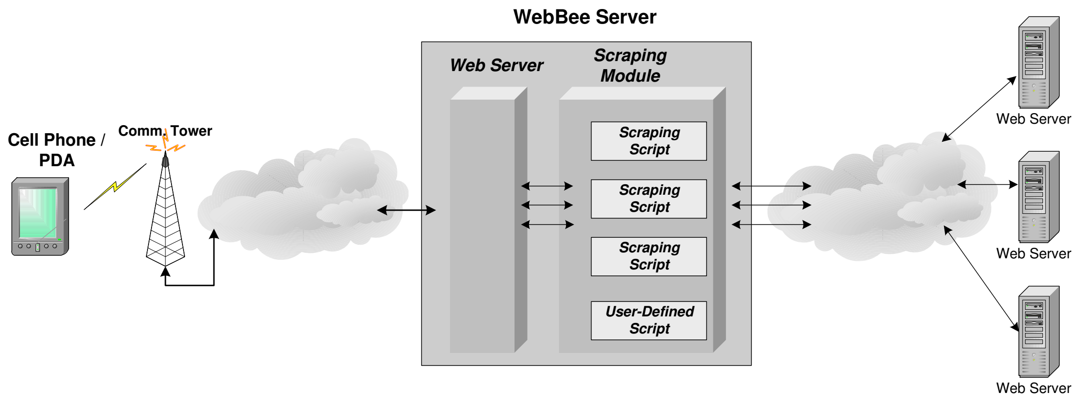


Fig. 1. WebBee Architecture

program that automatically extracts specified information of interest (e.g., text) from one or more web sites. A basic data-miner can be thought of as having two tasks: first, to comprehensively explore a web site by following links; second, for each page it comes across, the data-miner will scrape the desired information and add it to its database. The difference with WebBee is mainly in the first task; rather than automatically crawling through a web site, WebBee downloads a specific page to scrape in response to a user's request.

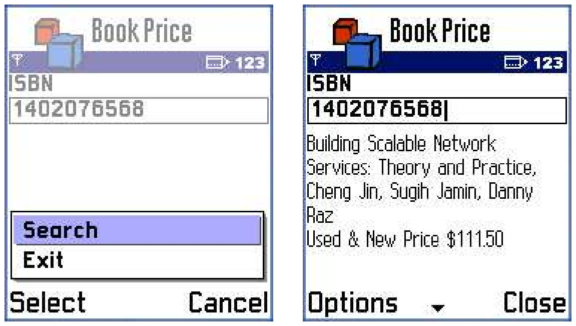
The structure of the WebBee architecture is illustrated in Fig. 1. The target client device can be any small mobile device with some form of Internet access, such as a mobile phone with an Internet plan. Another common client may be a PDA with wireless access like GPRS or WiFi (802.11b, etc.).

The WebBee server runs on a machine with a high-speed Internet connection. This server contains a web server as well as the scraping engine itself. The web server allows the WebBee server to act as a proxy between the mobile device and the web sites the user wants to view. The scraping engine is a web server module that is executed to process incoming requests from clients.

When a client on the mobile device communicates with the WebBee server via an HTTP POST request, the scraping engine module is invoked to service the request. The client indicates which script the user wants to run (or supplies its own script, if necessary), and the scraping engine begins to execute the script. The script instructs the WebBee server to download and scrape the appropriate web page(s). The scraped information is then returned to the mobile device via an HTTP reply.

The advantage of the proxy approach is that all the unnecessary information is eliminated before any data is transferred to the mobile device, reducing transfer time and network traffic costs. Also, because the work of scraping is done off the device, the client program can be kept small and simple, saving memory and processing power (which can have a significant impact on battery life) on the device.

We illustrate how our system works with a simple example. When a mobile phone user wants to look up the price of a book on amazon.com, she starts a Java client on her phone and enters the ISBN of the book (Fig. 2). The client contacts a WebBee server, which fetches the appropriate script from



Building Scalable Network Services: Theory and Practice, Cheng Jin, Sugih Jamin, Danny Raz
Used & New Price $111.50

Fig. 3. Screenshots for searching book price from amazon.com.

its cache and begins to execute it. According to the script, the WebBee server then posts a search request to amazon.com for that particular book by ISBN, and downloads the page with detailed information regarding the book. The title and price are scraped from the page, formatted, and sent back to the phone to display. On the surface, the user sees only the end result: a screen showing the price of the book (Fig. 3).

Using HTTP as the method of client-server communication brings several advantages. First, in the case of mobile phones, some service providers block access to all remote ports except port 80 (HTTP). Additionally, HTTP connection classes are provided with Java MIDP [18], which handles most communication error handling, simplifying the Java client and keeping its footprint small. Another important consequence of using HTTP is that the WebBee server appears as if it were just another web server to the clients; the fact that external web sites are being downloaded for scraping is hidden from the perspective of the client.

B. Scraping Scripts

In our prototype implementation, WebBee scraping scripts are manually written for each web site to scrape. The scripting language is simple enough that a user with some HTML experience can pick it up in an hour. Creating a simple script to scrape a page may typically take around fifteen minutes or less. The scripting language includes functionality such as fetching

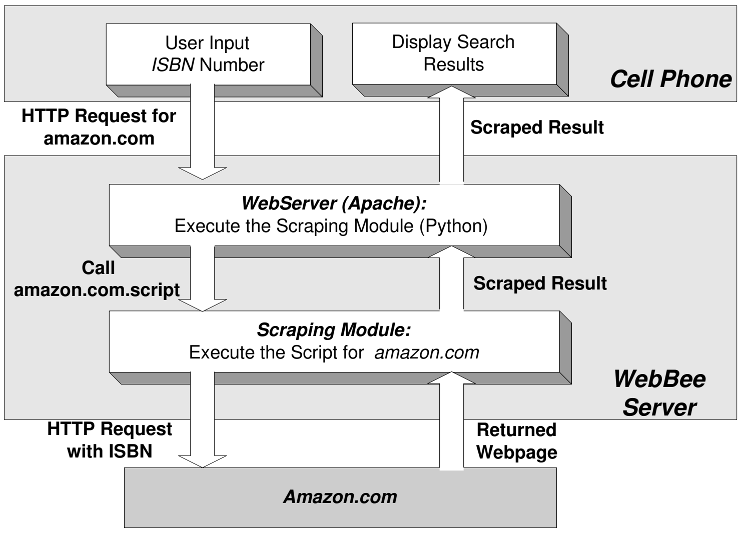


Fig. 2. Searching for a book's price from amazon.com.

a web page, forwarding form data, and pattern matching for locating information.

Similar to sites with mobile-friendly pages, there is a need for creation (and possibly maintenance) of scripts for each site. However, we argue that the use of scraping scripts requires no more work than the creation of these specialized mobile content pages by the content author, and has additional advantages as well.

The first is portability: any mobile device capable of running Java MIDP applications will have access to all the content scraped by any script, regardless of whether the device's built-in browser understands HTML, WML or cHTML. The content author need only produce a normal HTML site and a scraping script for it, rather than creating multiple versions of each page. If the content author includes "scrape-friendly" tags (described later in this paper) in their pages, no changes to the scraping script will be necessary when the author updates the site's layout; thus, maintenance of scripts will be kept to a minimum.

The second advantage is ease of implementation. In the traditional solution of creating separate mobile-friendly versions of a site, it is the content provider's responsibility to author the mobile-friendly pages, and to tie in their functionality to their server back-end. Using the scraping approach, any savvy user (not just the content author) can create a mobile-friendly interface to an existing web site without requiring any additional access to the server back-end functionality. By opening the development of such scripts to all users, we are confident that WebBee will speed the expansion of mobile-accessible content.

Simply scraping information for display with scripts, however, is just the most basic use of WebBee. Most forms of interaction that a desktop browser can perform with a site, can also be mirrored in WebBee using appropriate scripts and clients. For example, specialized clients can be written to make more sophisticated applications, such as a secure bidding interface for an online auction. Other example interactive sites suitable for WebBee include web loggers (or "blogs"), and phone and address directory services.

C. Image Transcoding

Besides the extraction of text, we have designed WebBee to transcode [19] images as well. Image transcoding is especially useful for scraping web sites with content such as map services. A scraping script for getting driving directions off MapQuest [20], for example, extracts not only the text of the directions, but also the associated images illustrating the route to follow.

For each image indicated by the script, the WebBee server will download the image, then resize and re-encode it as necessary to make it fit on a small display. The allowed limit of resizing can be specified in the scraping script to ensure that the image is not made so small as to be illegible. Other possible operations to reduce the image size include cropping (the removal of sides of the image) and color reduction (a 256-color image may be reduced to 8 colors, or to greyscale, for example). The transcoded image binaries are tagged onto the end of the HTTP replies that are sent back by the server to the mobile devices.

We chose PNG (Portable Network Graphics) [21] as the file format for the transcoded images, as PNG image decoding is natively supported in all Java MIDP devices; support for other formats such as GIF or JPEG, on the other hand, are not always guaranteed. The use of PNG considerably simplifies the client code needed to load and display the received images. One issue with the use of PNG, however, is that it is a lossless format; while it achieves a good compression ratio on line drawings (e.g., maps), it performs poorly compared to the lossy JPEG format on images such as photographs.

D. Location-Aware Services

In the United States, the Enhanced 911 mandate passed by the U.S. Federal Communications Commission requires, by the end of 2005, that all wireless carriers be able to locate any wireless phone calling 911 to an accuracy of 50 to 100 meters [22]. If this location information were available to client software running on the phones, the user's current position can be used with WebBee to provide location-dependent services.

Given a user's position from the client, for example, the WebBee server can look up the user's current address or zip code via a database, facilitating the automated retrieval of a map of the user's vicinity from a map services site. A quick lookup on directory services can locate nearby shops and restaurants. Getting the local weather forecast would be trivial. We have experimented with Bluetooth Global Positioning System (GPS) units. At the time of writing, however, there are very few mobile phones that support the JSR 82 Java API [23] for Bluetooth accessibility from within a Java application.

III. Implementation Details

A. The WebBee Server

We have implemented a working prototype of our architecture. The scraping engine is written in Python [24], and runs on any machine that has the Apache web server [25] with the `mod_python` module. `Mod_python` is an Apache module that embeds the Python interpreter within the server. We chose `mod_python` because it is proven capable of efficiently handling large volumes of concurrent requests [26].

The Apache web server listens for incoming client requests on the standard HTTP port 80. When a connection from a client is established, Apache forwards the request to `mod_python` for handling. Values of the form fields submitted by the client are extracted and sent as parameters to an instance of the WebBee scraping engine. The most significant parameter specifies which scraping script to run. As shown in Fig. 2, the WebBee engine loads the appropriate script from disk and begins to execute it. The script will instruct the WebBee engine to download one or more web pages into memory. The downloaded web page is then scraped according to the instructions in the script, and the formatted output is sent through Apache to the waiting client. Once the client has received all the output data, the connection is closed.

Whenever a new user request is received while all existing WebBee engine instances are busy serving other requests, `mod_python` creates a new *subinterpreter* to service that request. Subinterpreters ensure that different instances of the WebBee engine do not interfere with each other. Once created, a subinterpreter will be reused for subsequent requests. Thus, the multiple WebBee engine instances remain persistent in memory, keeping overhead minimal for future requests.

B. Web Scraping Scripting Language

We have designed a preliminary version of our scraping script language. The script is interpreted line-by-line, and each line contains a command to be executed. When the last command has run to completion, the script terminates and the resulting output is sent back to the client. Some important script commands are listed in Fig. 4.

The way information is extracted is based on manipulating a cursor through the HTML file (or files) downloaded from the web site to be scraped. The movement of the cursor is influenced by search commands that attempt to locate the specified patterns in the HTML file that encase the information to be extracted. As the cursor is moved to its desired positions, the script allows for the setting of starting and ending markers at the current cursor position. These markers serve to indicate blocks of text to be extracted. An additional command copies the indicated block of text to an output buffer, removing HTML tags if desired. The contents of the output buffer is returned to the client at the end of the script. A simplified example of how to extract the title from an HTML document is shown in Fig. 5.

Description 1 (Script commands)

1. **fetch** {**get**|**post**} <**url**>
   Fetches a web site into an internal buffer.
2. **seek** {**start**|**current**|**end**} <**offset**>
   Moves the cursor.
3. **search** {**fwd**|**back**} {**start**|**end**} <**exp**>
   Searches for the first exact instance of a string (exp) within the file.
4. **searchre** {**fwd**|**back**} {**start**|**end**} <**reg_exp**>
   Similar to "search" command, but does a regular expression search instead.
5. **select** {**start**|**end**}
   Sets the start/end selection markers to the current cursor position.
6. **select** {**print**|**cleanprint**}
   Prints the current selection to the output stream ("cleanprint" also removes HTML tags).
7. **print** <**string**>
   Prints a string to the output stream.
8. **if** {**pass**|**fail**}
   else
   endif
   These three commands are used to test if a command (e.g., search) succeeded, and branch accordingly.

Fig. 4. Basic WebBee script commands.

In Fig. 5, line 1 specifies the web page that the script should fetch. The second line searches for the first instance of "<title>". If there is a hit, as indicated in line 3, the script continues to scrape the page. Otherwise, the script stops and prints an error message (lines 9-11). Line 4 marks the start of selection to current cursor position, which is right after the first instance of "<title>"; line 5 search for a following instance of "</title>"; line 6 marks the end of selection to the new cursor position, which is before "</title>".[2] Lines 7 and 8 then print out the result.

The scripting language is designed to be as simple as possible for the jobs that it performs. By keeping the script's complexity down, possible security concerns can be more easily addressed. Additionally, the automatic generation of such scripts can be made simpler.

In our current implementation, scraping scripts are stored as text files on the scraping proxy server. A future version is planned to allow the client to send the server additional scripts,

[2]To simplify the example, we do not include the script commands to handle the failure case for line 5, which occurs if the web page has a "<title>" tag but no "</title>".

Description 2 (Script Example)

```
fetch get http://www.some-url.info
search fwd end <title>
if pass
    select start
    search fwd start </title>
    select end
    print The title is:
    select print
else
    print No title found!
endif
```

Fig. 5. An example scraping script that extracts the title of a page.

or instruct the server to download scripts from an external source on the Internet.

The client is implemented in Java MIDP, a platform chosen with portability in mind. The purpose of the client is to provide a user interface such that the user can post requests to the server, and to display returned results. Because all the scraping is done on the proxy, the client program can be kept small and simple.

C. Output Formats

In its simplest operating mode, WebBee scraping scripts output scraped information as plain text, which is sent by the server for verbatim display on the mobile client. While this is very suitable for small tasks such as retrieving weather reports or news, the application designer might sometimes desire to obtain more information in a structured manner, such as a list of search results, where each result is a name along with its associated contact information and ID number. In this case, plain text output would require specialized parsing in each client, which places an additional burden on the application developer.

To handle these cases, it is natural to consider using XML for data transport. XML gives the convenience of representing data structures in a standardized format that is easy to parse, thought it adds a cost of requiring tags around data elements and groups.

The WebBee server can be set to use XML in its output mode with a script command. To the script writer, this operating mode appears fundamentally the same as plain text mode, in the sense that it supports the same commands for searching and extraction. However, there are additional commands for delimiting extracted pieces of text by name and group, for which the WebBee server will generate the appropriate XML tags. There are several compact XML parsers for Java MIDP (such as kXML [27], which we are currently using) that can be incorporated into the client. The use of XML makes the transmission of structured data simple. An important point, however, is that the script writer need not be aware that XML is even used for transport; he only needs to keep in mind the concept of using names and groups to structure his extracted data. The WebBee server and the communications library in WebBee clients hide the details of transport from the user.

Transmission of binary data is handled by first setting the output mode to XML. Whenever a binary block of data is "attached," a descriptor for its content and length is added as XML, and the binary data itself is appended at the end of the output stream. The communications library in the client automatically handles binary data based on the descriptors it sees.

D. Applications

We have prepared a variety of sample applications for WebBee. Some of these include:

- Amazon: Search Amazon.com for books and other items, see prices and reviews, etc.
- Local: Search local.google.com for nearby businesses, obtain maps and directions.
- Maps: Get driving directions, route maps, area maps, etc. through MapQuest.
- Movie: Get local movie listings and show times.
- UM Directory: Search the University of Michigan's database for affiliated people.
- Weather: Obtain the local weather forecast from weather.gov.

Some of these applications are already available as subscription services from several major cellular operators in the United States, e.g., the Verizon SuperPages service [28]. WebBee applications are service-independent, however, and the number of applications can grow quickly from public contribution, rather than waiting on each provider to create new services. Given this public contribution of scripts and applications, we believe that it will be possible, in a reasonable amount of time, to cover almost every *useful* Internet function that one may wish to perform on a mobile device.

We reassert that WebBee is not meant to be a general browsing solution. There are a number of functions on the Web that users would rather use a desktop browser instead; for example, applying for a home loan, purchasing stocks or airline tickets. Choosing what Internet functions one may or may not want to perform on a mobile device is subjective; due to the open nature of WebBee, if a user finds an application inadequate, she can easily alter the existing script or write her own.

E. User Profiles

To select the appropriate script to execute on the server, the client refers to each script's unique identifier, which in our current implementation is the script's filename, and flag indicating whether or not to look in the global or private pool of scripts. The global pool of scripts consists of user-contributed scripts which have been made available to the public. Other scripts stored on the server may be private to each particular user.

In order to maintain this database of scripts, the WebBee server maintains a database of profiles linked to each user. Within these profiles, an authenticated user can store her private scripts. She can also choose to make publicly available any of the scripts she has uploaded. Guest users may execute any of the globally-available scripts without having to authenticate to the server.

IV. Discussions and Future Works

A. Server Scalability

As mentioned earlier, the WebBee server appears from the outside simply as a web server; that is because it is in fact a web server, albeit with an intelligent back-end. WebBee experiences similar CPU and network loads to an ordinary web server. One difference is that the bulk of the network data transferred is downstream: from web sites to the WebBee server. The scraped data returned to the client is a tiny fraction of this downstream data, around 0.5% to 2.5% for a typical application. Because the images that appear on each web page are seldom downloaded (only when the script specifies certain images to be scraped), it can be inferred that WebBee actually generates less network traffic, per page per user, than the web site that serves the original content would.

The WebBee server code maintains the desirable properties of any back-end service for a web site. It executes quickly, adding little CPU overhead to serving each request. The code is reentrant and remains in memory as a single instance for all requests, and consumes a reasonably light amount of additional data memory per request. To illustrate, our unoptimized prototype implementation, running on a 1.7 GHz Intel Pentium M machine, takes a total of just 85 microseconds of CPU time to parse and execute a script that scrapes a sample 24 KB page (disk and network time excluded). There is generally very little disk access; the only data to reside on disk are the scraping scripts, and because these are small and frequently accessed, they tend to remain cached in memory.

Scaling up WebBee is as simple as scaling up a web server. The common solutions of using server farms to distribute CPU load, as well as mirroring and multi-homing to spread network traffic [29], can be suitably adopted to WebBee. Additionally, caching the data obtained from active crawling of popular sites can reduce network traffic.

B. Resilience to Failure

WebBee's architecture requires all web traffic to pass through a proxy server; thus, the server becomes a single point of failure, whether by malicious attack, bugs or network congestion. We will briefly address these issues here.

The simplest way to deal with single points of failure would be to maintain several proxy servers in different locations, and have clients maintain a list of known servers to choose from. Clients may initially choose a server at random, and adjust their probabilities of using each server based on evaluations of their performance. If the server does not respond, the client will lower its probability of being selected the next time. If one working server offers better average response times for executing the same scraping script over another server, the probability that it will be chosen next time is increased.

To address the possibility of server degradation through buggy or malicious scripts, we put effort into keeping the scripting language as simple as possible, while maintaining desired functionality. Fortunately, pattern-based scraping is a relatively straightforward task, and a small set of commands and basic flow control is sufficient for most web sites of interest.

C. Script Creation

Critical to widespread adoption of WebBee would be the ease of creation of WebBee scraping scripts. While manual creation of such scripts is almost trivial to users familiar with HTML, we believe that the script creation process must be simplified to the point that any users who can use a web browser can create their own scripts. To meet this end, we are developing a tool named WebBee ScriptHelper to aid in the generation of WebBee scripts. ScriptHelper is a desktop tool, but the resulting scripts are to be used by WebBee applications.

In the design of this tool, we postulate that users are generally interested in receiving *dynamic* content on their mobile devices. We define *dynamic* to mean any content that changes over time, either through updates outside the user's control (e.g., news), through variations in user interaction (e.g., results from looking up different words in an online dictionary), or by a combination of both factors (e.g., weather reports that change over time, or as the user searches new locations). *Static* content refers to content that does not change over time. Separating dynamic from static content on a page is the basic function of ScriptHelper.

The idea behind ScriptHelper is simple: the user chooses a web page to scrape, and opens it using the embedded web browser in the ScriptHelper application. Using the ScriptHelper browser, she views multiple versions of the same page with varying content. ScriptHelper then compares the all the different versions of the page, and from its analysis detects the blocks of dynamic content sandwiched between static content. The user then selects which particular blocks of dynamic content she is interested in, and ScriptHelper generates the appropriate scraping script for her.

Using the familiar example of a weather web site, the user would begin by querying the forecast for her current location. She then goes back and runs a query for a different location. Next, she asks ScriptHelper to perform its analysis. Based on the differences between the two pages, ScriptHelper tells the user it has detected three dynamic blocks: one containing the current day's forecast, another containing travel advisories, and the last a text advertisement. The user is interested only in scraping the forecast, so she selects only the first block and hits the "generate" button. ScriptHelper detects unique text patterns encapsulating the selected block, that are common to both versions of the page. It uses these text patterns, as well as the URL of the weather web site that the user viewed, to create WebBee script commands for searching, selecting, and extracting that particular block. From this simple interaction with the user, ScriptHelper is able to gain all the information it needs to generate a simple WebBee script.

There are some challenges involved in developing such a tool. First of all, the identification of patterns around a piece of data is not foolproof, nor are the patterns necessarily unique. To improve the accuracy of the search patterns, the user is encouraged to view more than just two versions of each page. Also, when the resulting pages can differ significantly based on the query, ScriptHelper will fail. An example of this case would be when the user searches for driving directions. If she supplies exact addresses, the resulting page shows the driving directions she expected. If she supplies incorrect or vague locations, the resulting page may instead display a selectable list of approximate results. An advanced version of ScriptHelper would have to be able to generate scripts that branch conditionally to handle both cases.

D. Making Pages "Scrape-Friendly"

An important issue with the use of scraping scripts is that once the layout of a site changes, the patterns that surround each piece of information of interest are likely to change too. This will likely cause scripts to fail.

A content author who creates a scraping script for his own site (or wants to aid script writers) can include "scrape-friendly" tags in their site content. Such tags preserve the validity of scraping scripts throughout any site revisions, and require almost no effort to add and maintain.

The syntax for scrape-friendly tags is simple:

```
<scrape id="data_element_name">
  interesting information goes here
</scrape>
```

We illustrate the use of these tags with an example of a weather report page:

```
Your current weather:
<scrape id="temperature">18</scrape>
degrees centigrade.
<scrape id="forecast">
Expect strong winds tonight,
with a chance of rain.
</scrape>
A message from our sponsor:
<scrape id="advertisement">
For roof repairs,
call Biff & Buff Builders Inc.
at 222-555-1234
</scrape>
```

Desktop browsers that do not recognize the "scrape" tags will simply ignore them, rendering the pages without any visible difference. To the WebBee script author however, pattern matching has been made trivial, and they can also be assured that their scripts continue to work correctly after any page layout changes. Content authors who are interested in allowing mobile-friendly access to their content can include these tags at almost no additional effort. In contrast, the current practice of making explicit mobile-accessible content would require the content author to create multiple versions of each page in each mobile markup language (WML, etc.).

Scrape-friendly tags are not required for WebBee to function normally; they are just an optional means of ensuring scripts remain robust in the face of site layout changes.

E. WebBee as Client Middleware

Our current implementation of the client entails the creation of individual Java binaries for each WebBee application, each with a hard-coded user interface (UI). A cleaner solution is to implement the WebBee client as a middleware for mobile devices, where a WebBee "application" would consist of a UI Descriptor and additional information tying the UI Descriptor to a particular WebBee script. The UI Descriptor of an application describes the user interface for that application and is written in a UI markup language. Such a markup language will have descriptions for the data entry screen, including widget layout and text entry boxes. It will also describe how the returned information is to be formatted for display.

This middleware version of the WebBee client would be a Java program that allows the user to select an existing WebBee "application" to execute, or to download and save new ones from the Internet. Once a WebBee "application" is launched, the middleware interprets the UI Descriptors for display, and handles the network communication between the mobile device and the WebBee server. The communications layer relays requests submitted from the rendered UI to the server, and forwards the server's response back to the UI layer for display as specified by the UI Descriptors.

One advantage of a middleware client is that the user needs only one Java application for all the functionality WebBee offers; each WebBee "application" is simply a UI Descriptor and some associated metadata saved in the mobile device's persistent storage. The rendering and communications code would be shared, ensuring that any bug fixes and updates to the middleware immediately affect all the "applications." However it is essential that the UI Descriptors must be as easy to create as the scrapping scripts in order for WebBee to gain widespread use.

Another benefit of this approach is that common parameters, such as user name or address, can be shared among all WebBee applications. For example, the zip code in the target address in a driving directions application may also be the default zip code in a weather application, and the same address may be used for querying local cinema showtimes. Java MIDP currently does not allow for data sharing between Java applets.

F. Limitations

WebBee uses web forms to pass user input to web sites, and scrapes the resulting HTML pages for information to send back to the user. Because WebBee relies on web forms for user interaction, it is not a suitable solution for accessing sites using client-side technologies like Java, Javascript or Macromedia Flash to process and display queries. The amount of work required to duplicate the functionality of these client-side technologies on the WebBee server would be enormous, and to extract meaningful results (e.g., from graphical rendering of text in a Flash application) may also be difficult. We do not feel this is a major issue, however, as sites that provide their information in this manner are still by far the minority.

G. Extending Device Compatibility

For the few mobile devices that do not support Java, but have a built-in browser for a particular markup language, an alternate interface to WebBee is required. We propose that front-ends for each script can be created automatically on the WebBee server in HTML, WML, or cHTML. As each scraping script specifies the arguments expected from a client, it would be a simple matter to automatically create an input widget for each argument in a script in the desired markup language. The output generated from the scraping script, whether plain text or XML, can be converted according to device-specific markup languages.

Once these pages are generated, the user can access each script's interface using the device's built-in browser by entering each script's URL, which may simply be its filename concatenated with ".html". While this easy solution works for the majority of simple scripts, it is insufficient for more sophisticated applications, especially those which process or retain data on the client. For such applications, the Java client would work best.

V. Conclusion

In this paper, we have presented WebBee, a client-proxy architecture that describes a platform-independent gateway between small mobile devices and the World Wide Web. The design of WebBee encourages each application to specialize in a single task, allowing the user interfaces to be made as simple as possible; simple UIs are very suitable for the small displays of mobile devices. Each WebBee application minimizes the use of the network, requesting as much information from users as possible before connecting to the network to resolve their queries. Web scraping scripts, which are an important part of WebBee, tell the server how to extract information of interest from each web site. Since only extracted information is returned to each mobile client, less data are sent over the mobile network, and the information can be fitted to the small displays of the mobile device. Finally, instead of requiring users to go through a network portal, WebBee applications reside on the users' mobile devices, further reducing network usage. The WebBee application repository is open to public contribution, requiring no support from each web site's administrators in order to grow.

References

[1] "Nokia Phones," http://www.nokia.com/phones/.

[2] "FOMA SH900i," http://www.3g.co.uk/PR/March2004/6812.htm.

[3] "HP iPAQ Pocket PC," http://welcome.hp.com/country/us/en/prodserv/handheld.html.

[4] "Dell Axim Series," http://www1.us.dell.com/content/ products/compare.aspx/handhelds.

[5] Federal Communications Commission, "Third Generation "3G" Wireless," 2002, http://www.fcc.gov/3G/3gfinalreport.pdf.

[6] Michel Mouly and Marie-Bernadette Pautet, *The GSM System for Mobile Communications*, Telecom Publishing, 1992.

[7] R. Chakravorty and I. Pratt, "Performance Issues with General Packet Radio Service," *Journal of Communications and Networks (JCN), Special Issue on Evolving from 3G deployment to 4G definition*, vol. 4, no. 2, Dec. 2002.

[8] "wapSpeed.com," http://wapspeed.com.

[9] M. Busse, B. Lamparter, M. Mauve, and W. Effelsberg, "Lightweight qos-support for networked mobile gaming," 2004.

[10] "Opera Web Browser," http://www.opera.com/.

[11] "Symbian OS," http://www.symbian.com/.

[12] "Palm OS," http://www.palmsource.com/palmos/.

[13] "Picsel," http://www.picsel.com/.

[14] Y. Hwang, J. Kim, and E. Seo, "Structure-Aware Web Transcoding for Mobile Devices," *IEEE Internet Computing,Septemer/October 2003 (Vol. 7, No. 5)*, 2003.

[15] A. Fox, S. D. Gribble, and E. A. Brewer Y. Chawathe, "Adapting to Network and Client Variation Using Active Proxies: Lessons and Perspectives," *IEEE Personal Communications*, Sep. 1998.

[16] "Wmlscript tutorial from wireless developer network," http://www.wirelessdevnet.com/channels/wap/training/wmlscript.html.

[17] "NTT Docomo I-Mode," http://www.nttdocomo.com/corebiz/imode/index.html.

[18] "Java Mobile Information Device Profile (MIDP)," http://java.sun.com/products/midp/.

[19] R. Han, P. Bhagwat, R. LaMaire, T. Mummert, V. Peret, and J. Rubas, "Dynamic adaptation in an image transcoding proxy for mobile web browsing," *IEEE Personal Communications*, vol. 5, no. 6, pp. 30–44, Dec. 1998.

[20] "MapQuest," http://www.mapquest.com/.

[21] "Portable Network Graphics (PNG): Functional specification," ISO/IEC 15948:2004.

[22] J. Warrior, E. McHenry, and Kenneth McGee, "They Know Where You Are," *IEEE Spectrum*, Jul. 2003.

[23] "JSR-000082 Java APIs for Bluetooth," http://jcp.org/aboutJava/communityprocess/final/jsr082/index.html.

[24] "Python Programming Language," http://www.python.org/.

[25] "Apache Web Server," http://www.apache.org.

[26] "MoinMoin Performance Proposals," http://moinmoin.wikiwikiweb.de/PerformanceProposals.

[27] "kXML," http://www.kxml.org/.

[28] "Verizon SuperPages On the Go," http://verizon.superpages.com/.

[29] "Akamai," http://www.akamai.com/.