Puppeteer: Component-based Adaptation for Mobile Computing

Eyal de Lara[†], Dan S. Wallach[‡], and Willy Zwaenepoel[‡]

[†] Department of Electrical and Computer Engineering
[‡] Department of Computer Science
Rice University

Abstract

Puppeteer is a system for adapting component-based applications in mobile environments. Puppeteer takes advantage of the component-based nature of the applications to perform adaptation *without* modifying the applications. We illustrate the power of Puppeteer by demonstrating adaptations that would otherwise require significant modifications to the application.

Our initial prototype supports Microsoft PowerPoint and Internet Explorer 5 without requiring any changes to the applications. It supports delayed data and image transmission as well as progressive image refinement. We measure our system's effectiveness with a large number of PowerPoint documents and Web pages. We measure user-perceived latencies for a variety of adaptation policies and over a variety of different network bandwidths. Our results show that Puppeteer can achieve average reductions in user latency of up to 84.22% for PowerPoint documents loaded over a 384 Kb/sec link and 76.42% for HTML documents loaded over 56 Kb/sec link.

1 Introduction

The need for application adaptation in mobile and wireless environment is well established [3, 10, 11, 18, 30, 29]. Users want to use the same tools in a mobile environment as they do in their well-connected office. Users want to access the same files, even in cases where the files are quite large and the bandwidth is limited, leading to long latencies for file upload and download.

Many approaches to adaptation have been proposed before. We divide them into roughly two categories: application-based [12, 13, 16] and system-based adaptation [19, 24, 25]. With application-based adaptation, the application is modified or rewritten, allowing any adaptation strategy to be implemented, but offering no benefit to users of a non-adapted application. With system-based adaptation, the system uses its interposition between the client and the remote data to perform the adaptation. System-based adaptation works with a wider range of applications but is limited in the adaptation strategies it can take. We have designed a system to adapt existing applications by taking advantage of their component-based nature: these applications already export a number of APIs allowing them to be used as components in a larger system, and we use these APIs to adapt them to low-bandwidth environments.

Puppeteer is a new system we have built to support a new form of adaptation we call *component-based adaptation*. Puppeteer takes advantage of the modular component-based structure, consistent file formats, and exposed APIs of modern productivity applications. By remotely "pulling strings" on these applications, Puppeteer can add a wide variety of adaptation policies to pre-existing applications without modifying them in any way.

Consider the example of loading a Microsoft PowerPoint file across a slow link with a protocol like FTP or HTTP. PowerPoint files can be several megabytes long, yet PowerPoint loads the entire file before it returns control to the user. Our goal is for PowerPoint to access a file over a slow link and return control to the user as fast as possible. One possible adaptation would be to load only the data associated with the first slide, return control to the user, and then load the remaining slides in the background. With component-based adaptation, we can fetch slides individually and 'paste' them into the document as the user is working. And, if the user prefers, we can transcode the images within the document, trading off image fidelity for compression, allowing the user to begin working immediately with low fidelity images in the document. While system-based adaptation could certainly perform image transcoding, component-based adaptation can dynamically respond to higher available bandwidth (which can vary over time in mobile environments) and refetch an image with higher fidelity.

Component-based adaptation is by nature restricted to adapt only component-based applications with exported APIs. While certainly a limitation, we observe that many desirable candidate applications for adaptation are already component-based: the Microsoft Office Suite, the KDE Office Suite, Microsoft Internet Explorer, etc. Recognizing the advantages of component-oriented software construction – independent of adaptation – we foresee an increasing number of such applications being developed as components with exported APIs. The more fundamental question about component-based adaptation is: to what extent can it support the adaptation mechanisms that a customized application-based approach can achieve? Furthermore, we wish to understand precisely how portable a component-based adaptation system can be. Clearly, the system will need "drivers" for each application it wishes to support. If these drivers could be small, that would demonstrate the portability of the adaptation system.

In this paper, we demonstrate that a large number of desirable policies can be implemented easily and efficiently with component-based adaptation. This paper describes the implementation of the Puppeteer system on Windows NT using Java, and our experiences using this implementation to adapt two applications – PowerPoint and Internet Explorer – for low bandwidths.

The rest of this document is organized as follows. Section 2 describes the requirements that application must meet to be adapted by Puppeteer. Section 3 presents the architecture of the system. Section 4 introduces the applications we use to evaluate the prototype. Section 5 describes the documents we use in our experiments. Section 6 describes the experiments platfrom. Section 7 presents our experimental results. Section 8 discusses future directions. Section 9 discusses related work. Finally, section 10 discusses our conclusions.

2 Background

In this section we examine the requirements that an application has to meet in order to be adapted by Puppeteer. This discussion is technology independent as the required features can be implemented in a variety of component systems.

Strictly speaking, Puppeteer only requires that an application exposes a run-time interface that allows the system to view and modify the data the application operates on. We will refer to this feature as the Data Manipulation Interface (DMI). Puppeteer, however, can benefit greatly from two extra features: a parsable file format and an event notification mechanism.

When the application only exposes a DMI, but has an opaque file format, Puppeteer adapts the application by running an instance of the application on the server and using the DMI to uncover the structure of the data; in some sense using the application as a parser. It also runs an instance of the application on the client and uses the client's DMI to update it with newly fetched components. This configuration allows for a high degree of flexibility and makes porting applications to Puppeteer more straightforward as programmers need not understand the application's file format. This configuration, however, has a larger overhead on the proxy server and results in lower overall system performance and scalability. Moreover, it requires both the client and server to run the environment of the application, which in most cases amounts to having to run the same operating system in both servers and clients.

In contrast, when the file format is parsable, either because it is human readable (*e.g.*, XML) or there is sufficient documentation to write a parser, Puppeteer can parse the file (or files) directly to uncover the structure of the data. In our experience this results in better throughput and enables clients and server to run on different platforms. (The Puppeteer client runs on Windows NT while the Puppeteer server runs on Linux.) In the course of constructing our prototype and in a previous study that evaluated Microsoft Office file formats [9] we have built several parsers for both text and binary formats. In our experience, writing parsers for text formats, like HTML and XML, is much simpler than creating parsers for binary formats, even when appropriate documentation is available.

Finally, Puppeteer tracks the users as she uses the application to tailor the adaptation to the user's behavior. We implement tracking by either polling the application for modifications (*e.g.*, checking where the user focus is several times per second), or when available, by registering for events with the application's event mechanism. Events are usually call-back functions that get executed as a side-effect of well-defined operations such as the user opening a new document, changing focus, or clicking the mouse. While Puppeteer requires DMI, it does not strictly need an event notification mechanism. Event notification, however, greatly simplifies the job of writing a Puppeteer driver for the application.

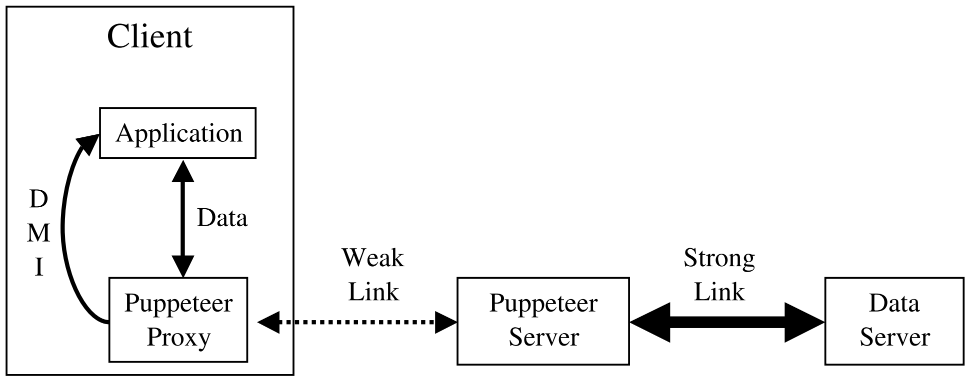

Figure 1: Puppeteer architecture.

3 Puppeteer

Figure 1 shows the four tier Puppeteer architecture. It consists of the application to be adapted, the Puppeteer client proxy, the Puppeteer server, and the data server, which stores the data that the user wants to work on. The application and data server are completely unmodified. The Puppeteer client proxy and server work together to adapt the system to the available bandwidth.

The client proxy is in charge of bandwidth adaptation and resource management. It acts as a single point of control for managing all communication between applications on the client and the server. It is also in charge of remotely controlling the applications and adapting them to the available bandwidth.

The Puppeteer server is assumed to have strong connectivity with the Internet and with the data server. The Puppeteer server is responsible for parsing documents, exposing their structure, and transcoding components as requested by the client proxy.

Data servers can be arbitrary repositories of data such as Web servers, file servers or data bases. In the design of our prototype we assume that the servers are not overloaded and that the latency between the Puppeteer server proxy and the data server is small compared to the latency between the Puppeteer client proxy and Puppeteer server.

In order for Puppeteer to support a new application, a client and server driver must be written for the application and its file format. The server driver understands the file format used (*e.g.*, XML, OLE archives) and can fetch specific components from a file. In the case of specific component types we wish to transcode, such as images, the server has support for this as well. The client drivers must support inserting new components into a running application, tracking which components are being viewed by the user, and must be able to intercept load and save requests. Table 1 summarizes the Server and Client driver interfaces.

Location	Functionality	Description
Client Driver	Close	Closes the application, performs cleanup.
	Update	Inserts new data into the running application.
	Monitor	Tracks the user and tells Puppeteer what to fetch.
Server Driver	Close	Performs cleanup.
	Parse	Parses the component and creates a skeleton.
	Transcode	Returns the component at a requested fidelity level.

Table 1: Client and Server driver interfaces.

In practice, the drivers are only a small part of the overall system. For instance, the PowerPoint and IE5 drivers require just 1412, and 611 lines of code, respectively. The remainder of the Puppeteer system is approximately 8000 lines of general-purpose code. Moreover, productivity suites like Microsoft Office use many of the same components across applications, and have similar file formats and DMIs, allowing reuse of code from one driver to the next.

3.1 Adaptation process

The adaptation process in Puppeteer is divided into three stages: parsing the document to uncover the structure of the data, fetching selected components at specific fidelity levels, and updating the application with the newly fetched data. We describe next each of these steps in more detail.

3.1.1 Parsing and skeleton creation

The first step in adapting an application is constructing a *skeleton* that captures the structure of the data used by the application. The skeleton has the form of a tree, with the root being the document, and the children being pages, slides or any other element in the main document. The skeleton is a multi-level data structure as components in any level can be themselves subdivided into sub-components. For instance a PowerPoint document is composed of slides that may contain images or OLE-based embedded objects, which can have further structure.

For most applications, we construct the skeleton by statically parsing the document at the Puppeteer server, which we assume has fast access to the whole document. However, static parsing at the server does not work well for pages that choose what data to fetch and display by executing a script, or by other dynamic mechanisms. Dynamic HTML (DHTML) [14] Web pages are a good example of this case. The browser fetches the initial HTML page, which can contain JavaScript. When the script is executed it instructs the browser to load images and other components. Because the structure of DHTML pages changes dynamically, the server cannot statically determine a document's skeleton. Even if the server interpreted the JavaScript as if it were a client, the document structure may still not become fully visible as certain events, such as moving a mouse over a button, will not necessarily be triggered. Instead, Puppeteer relays on a sniffing mechanism, implemented in the Puppeteer client proxy, that traces Web requests. The Puppeteer client proxy intercepts URL requests from within a page, allowing it to dynamically add new images and components to a Web page's skeleton. Regardless of whether the skeleton is built or modified in the proxy or the server, any changes to the skeleton are reflected in both ends. This is necessary because the skeleton is the main mechanism by which Puppeteer client proxies and servers communicate to exchange components.

3.1.2 Fetching components

Policies running on the client proxy control the fetching of components. These policies traverse the skeleton, choosing what components to fetch and with what fidelity level to fetch them.

Puppeteer provides supports for two types of policies: general purpose policies implemented by the system; and component specific policies implemented by the component driver.

Typical policies choose components based on available bandwidth and user specified preferences (*e.g.*, pre-fetch all text first), while other policies track the user as she runs the application and try to anticipate her needs (*e.g.*, fetch the PowerPoint slide that currently has the user's focus and pre-fetch subsequent slides in the presentation).

Regardless of whether the decision to fetch a component is made by a general purpose policy or by a component specific one, the actual transfer of component's data is performed by Puppeteer, relieving the component driver from the intricacies of communication.

3.1.3 Application update

When the user opens a document, the client proxy's policy selects an initial set of components from within the component to fetch. It then supplies this set of components to the application as though it had the full document at its highest level of fidelity. The application, believing that it has finished loading the document, returns control to the user. Meanwhile, Puppeteer knows that only a fraction of the document has been loaded and will use the techniques described above to fetch or upgrade the fidelity of the remaining components. Puppeteer also hooks into the application to track the user's movements through the document. If the user skips pages, the client proxy can drop components it was fetching and focus the available bandwidth on fetching components that will be visible to the user.

To update the application with new or higher fidelity components, the Puppeteer client proxy drivers in our system use the DMI interfaces exposed by the application. Puppeteer supports two update modalities that match the way most applications function. For applications that support a cut-and-paste mechanism (*e.g.*, PowerPoint, Word, Excel) the driver uses the ClipBoard to insert new versions of the components. On the other hand, for applications that must explicitly to read every item they display (*e.g.*, MS IE5), the proxy instructs the application to reload the component (*i.e.*, asking IE5 to refetch a URL).

4 Prototype applications

We chose to support Microsoft PowerPoint (a presentation graphics system) and Microsoft Internet Explorer 5 (a Web browser, hereafter "IE5") as the first two applications for our prototype because these two applications comply with all or most of the requirements for DMI, parsable file formats, and event notification stated in section 2. Moreover, these applications are widely popular and would likely be used in mobile environments. Most important, PowerPoint and IE5 have radically different DMIs. By supporting both, we are more likely to accurately design the interfaces between the portable portions of Puppeteer and its application-specific drivers. These drivers are discussed in more detail in section 3.

Next, we discuss how, and to what extent, PowerPoint and IE meet the requirements stated in section 2, and the techniques Puppeteer uses to adapt the applications.

4.1 Data manipulation interface

PowerPoint and IE5 DMIs are based on the Component Object Model (COM) [5] and the Object Linking and Embedding (OLE) [6] standards. The interfaces they provide are reasonably well documented [23, 28] and have traditionally been used to extend the functionality of third party applications.

COM enables software components to export well-defined interfaces and interact with one another. In COM, software components implement their services as one or more COM objects. Every object implements one or more interfaces, each of which exports a number of methods. COM components communicate by invoking these methods.

OLE is a set of standard COM interfaces. Among the many OLE technologies, we are most interested in its capacity to enable third party applications to remotely control instances of running applications; usually referred to as Automation. Puppeteer adapts applications by invoking their Automation interfaces to modify their runtime behavior.

The PowerPoint and IE5 Automation interfaces provide excellent access to compose and modify their internal data structures. Through calls to these interfaces Puppeteer can uncover the structure of the data or modify it by adding or removing elements (*e.g.*, cut and paste within PowerPoint), or by forcing an element to reload itself (*e.g.*, reloading within IE5).

4.2 File formats

PowerPoint

OLE also defines a structured binary file format [20, 21] (hereafter, an "OLE archive") for storing a hierarchy of components. The structure of an OLE archive resembles a directory tree with files at the leaves. OLE archives are used for all Microsoft's Office applications. However, Office 2000 introduced a new XML-based format [22], providing a more browser-friendly option for storing documents. While an OLE archive appears as a single file, an XML document appears as an entire directory of XML files, approximately one per component, image, or slide. The data for all embedded components is, however, stored in single compressed OLE archive. We choose to base the Puppeteer prototype on the XML representation because it is semantically comparable to the OLE archive and the human readable nature of XML makes it easier to parse and manipulate the document. Moreover, XML is also used by KDE's KOffice and many newer tools.

Another issue with PowerPoint is its pre-existing support for incremental document loading. When the file format is an OLE archive and it's on a filesystem that supports random access (*e.g.*, a local hard disk CIFS, or NFS), PowerPoint will delay image loading. However, if the filesystem is not random (e.g. HTML) access or the file format is XML, PowerPoint will load the entire document before returning control to the user.

HTML and Dynamic HTML

While HTML is straightforward to parse, the introduction of JavaScript has allowed for documents whose structure can change dynamically. This makes it impossible for Puppeteer to statically determine all the components that make up a given Web document. Even if the server interpreted the script as if it were a client, the document structure may still not become fully visible as certain events, such as moving a mouse over a button, will not necessarily be triggered.

Puppeteer implements a hybrid mechanism that that tries to uncover statically as much of the document's structure by parsing the HTML and falls back on a sniffing mechanism that traces Web requests and identifies any new elements added by scripts.

4.3 Event notification

PowerPoint's event notification mechanism is very primitive and encompass just a handful of large-granularity events like opening or closing of document, making it inappropriate for tracking the behavior of the user. PowerPoint DMI, however, provides functions for determining the location of the current focus. By polling on these interfaces Puppeteer can tracks the user.

In contrast, IE5 supports a rich event mechanism that allows third-party applications to register call-back functions for any event. Puppeteer uses this interface to detect when the user enters a URL and to track the location of the mouse. Puppeteer uses this information to drive image fetching and fidelity refinement.

5 Data sets

We selected the set of PowerPoint documents used in our experiments from a collection of Microsoft Office documents that we characterized earlier [9]. The full collection includes 2167 documents downloaded from 334 Web sites with sizes ranging from 20 KB to 21 MB.

We obtained our HTML documents by re-executing the traces of Web client accesses collected and characterized by Cunha *et. al.* [7]. These traces include access from 2 user groups made during a period of 7 months from November 1994 through May 1995. These traces have 46,830 unique URLs corresponding to 3026 Web sites. For every URL that we were able to access (many pages had either disappeared or were corrupted), we downloaded the HTML file and any images referenced by them. We did not download any documents linked from these pages. In this manner we acquired 3796 HTML files and 15,329 images, comprising 89 MB of data downloaded from 1009 sites. Documents ranged in size from a few bytes to 773 KB, including images.

Because these data sets are so large, transmitting them at low bandwidth without transcoding would take a prohibitive amount of experimental time. We chose to run our experiments on just 92 PowerPoint documents and 182 HTML documents to limit the running time of tests. In the slowest network configuration the selected sets requires 138 minutes for PowerPoint and 55 minutes for HTML to complete the longest test. For completeness, however, we ran one test over the full sets of both document types over a high bandwidth network. The full sets and the selected documents produce similar plots.

For our PowerPoint experiments, we selected 92 documents by sorting all documents larger that 32 KB into buckets with sizes increasing by powers of 2. We then randomly selected 10 documents from each bucket. The largest bucket, consisting of documents with sizes greater than 16 MB had only 2 documents. Thus, our experimental set has $9 \times 10 + 2 = 92$ members.

For our Web experiments, we selected 182 HTML documents from the downloaded set by sorting all documents larger that 4 KB into buckets with sizes increasing by powers of 2. We then randomly selected 25 documents from each bucket. The largest bucket, consisting of documents with sizes greater than 512 KB had only 7 documents. Thus, our experimental set has $7 \times 25 + 7 = 182$ members.

6 Experiment environment

Our experimental platform consists of two Pentium III 500 MHz machines running Windows NT 4.0 that communicate via a third PC running the DummyNet network simulator [27]. This setup allows us to control the bandwidth between client and server to emulate various network technologies.

All our experiments access data stored by an Apache 1.3 Web server. For the experiments where we meassure the latency of loading the documents using the native application, Apache is the only process running on the server. For the Puppeteer experiments, the Apache server and Puppeteer server run on the same machine.

7 Experimental results

In this section we experimentally explore two questions: how much benefit can be derived from using Puppeteer to load documents over various network speeds; and what overhead is incurred by using Puppeteer in terms of both latency and bytes transmitted.

For each of the prototype applications, we consider three network configurations that shift the operational bottleneck from the client's CPU to the network. In all experiments we compare downloading the documents using only the native application to downloading them with Puppeteer support.

7.1 Latency reduction

Puppeteer reduces user perceived latency, the time users wait before they can start work on a document, by changing the application's policies in two basic ways: (1) changing the order and time when components are fetched; and (2) changing the fidelity of fetched components. In this section we explore the impact of these two adaptation strategies. First we explore the use of fetching components on demand for PowerPoint documents. Second, we explore the use of progressive JPEG compression and rendering for Web pages.

7.1.1 Fetch order

Loading the first slide

In this experiment we measure the latency for loading the first slide in a PowerPoint presentation. Figures 2, 3, and 4 show the latency measurements for loading the documents over 384 Kb/sec, 4 Mb/sec, and 100 Mb/sec network links. Figure 5 shows the data transfered to load the documents. The figures show results for native PowerPoint (*PPT*), and Puppeteer runs that load all the components of a presentation (*Full*), just the components of the first slide (*Slide*), and the components of the first slide plus the text for the entire presentation (*Prefetch*). For each document, the plots contain four vertically aligned points representing the latency to transmit the document in each system configuration.

Full is included to measure the overhead of using Puppeteer. This configuration represents the worst possible case as it incurs the overhead of uncovering the document's skeleton but does not benefit from any adaptation. *Full* shows that the Puppeteer overhead is not significant in cases with low bandwidth but becomes noticeable as bandwidth increases; documents experience median overheads of 5.02%, 17.05%, and 49.13% for the 384 Kb/sec, 4 Mb/sec, and 100 Mb/sec respectively. The larger differences in latency only occur in smaller documents. The amount of data transfered by *Full* is comparable to *PPT* with most documents experiencing overheads smaller than 2.72%.

Slide shows the large potential of Puppeteer to reduce latency. While we expected to save latency with the slower 384 Kb/sec and 4 Mb/sec networks, the savings over the 100 Mb/sec network came as a surprise. While Puppeteer achieves most of its savings in the 384 Kb/sec and 4 Mb/sec networks by reducing network traffic, the transmission times over the 100 Mb/sec are too small to account for the savings. The savings result instead, from reducing the initial parsing/rendering time. On average, *Slide* achieves latency reductions of 84.22%, 66.97%, and 56.71% for documents larger than 1 MB on 384 Kb/sec, 4 Mb/sec, and 100 Mb/sec, respectively.

Also, the data in figure 5 shows that, for large documents, it is possible to return control to the user after loading just a small fraction of the total document's data (about 4.5% for documents larger than 3 MB).

An interesting characteristic of the figures is the large variation in user-perceived latency at high network speeds and the alignment of data points into lines as the network speed decreases. The high variability over high network speeds results from the experiment being CPU-bound. With high bandwidth, user-perceived latency is mostly dependent on the time that it takes for PowerPoint to parse and render the presentation. This time is not only dependent on the size of the presentation, but is also a function of the number of components (such as slides, images, or embedded object) in the presentation. In contrast, as we slow the network speed, the process becomes network bound and user-perceived latency becomes dependent on the overall size of the document. This observation is supported by the similar shapes of figures 2 and 5.

Prefetching

We explore the effects of prefetching by extending the *Slide* experiment of the last section to prefetch the text elements of the rest of slides of the presentation. The *Prefetch* runs in figures 2, 3, 4, and 5 show the results of this experiment.

When comparing the data point of the *Prefetch* run to *Small*, we see that the latency has moved up slightly. The latency is still significantly lower lower than the latency for *PPT*, achieving savings in averages of 73.37%, 54.94%, and 45.18% over 384 Kb/sec, 4 Mb/sec, and 100 Mb/sec, respectively. Moreover, the increase in the amount of data transfered, especially for documents

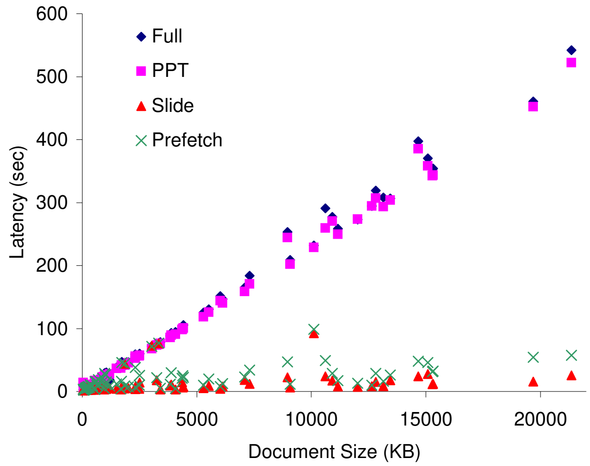

Figure 2: Load latency for PowerPoint documents over 384 Kb/sec. Shown are latencies for native PowerPoint (*PPT*), and Puppeteer runs for loading all components (*Full*), loading just the components of the first slide (*Slide*), and loading the text of all slides in addition to all the components of the first slide (*Prefetch*).

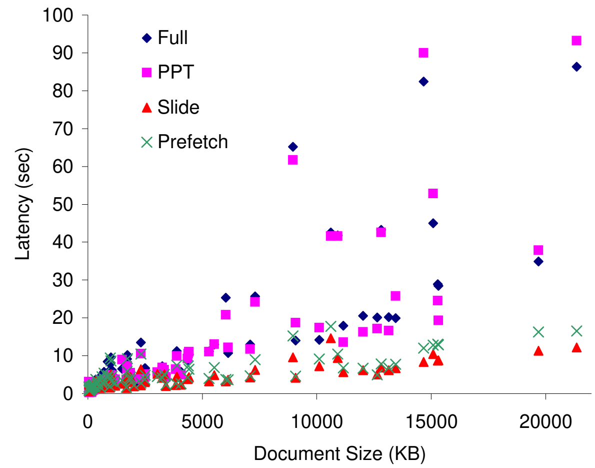

Figure 4: Load latency for PowerPoint documents over 100 Mb/sec. Shown are latencies for native PowerPoint (*PPT*), and Puppeteer runs for loading all components (*Full*), loading just the components of the first slide (*Slide*), and loading the text of all slides in addition to all the components of the first slide (*Prefetch*).

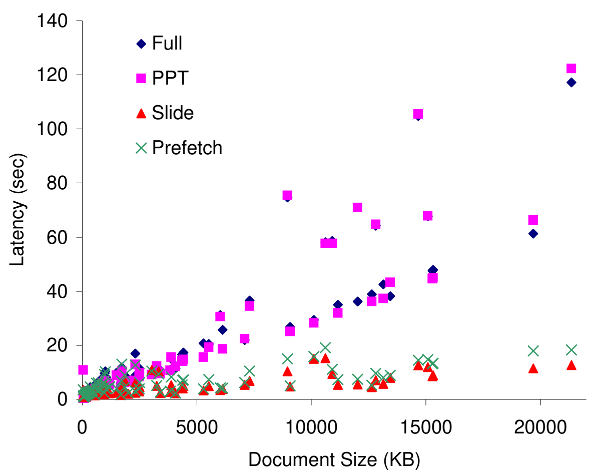

Figure 3: Load latency for PowerPoint documents over 4 Mb/sec. Shown are latencies for native PowerPoint (*PPT*), and Puppeteer runs for loading all components (*Full*), loading just the components of the first slide (*Slide*), and loading the text of all slides in addition to all the components of the first slide (*Prefetch*).

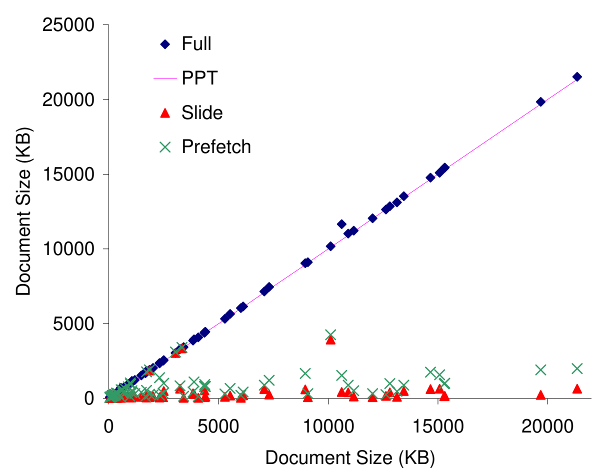

Figure 5: Data transfered to load PowerPoint documents. Shown are totals for native PowerPoint (*PPT*), and Puppeteer runs for loading all components (*Full*), loading just the components of the first slide (*Slide*), and loading the text of all slides in addition to all the components of the first slide (*Prefetch*).

larger than 4 MB, is small, amounting to only an extra 6.42% above original document size. These results are consistent with the findings of [9] where text accounted for a small fraction of the total data in large PowerPoint documents. These results suggest that text should be prefetched in almost all situation and that the lazy fetching of components is more appropriate for the larger image and OLE objects that appear in the documents.

7.1.2 Fidelity

JPEG compression

In this experiment we explore the use of lossy JPEG compression and progressive JPEG technology to reduce user perceived latency for HTML pages. Our goal is to reduce the time required to render a full version of the page by lowering the fidelity of some of the page's elements.

Our prototype converts, at run time, GIF and JPEG images embedded in the HTML document into progressive JPEG format[1] using the PBMPLUS [26] and Independent JPEG Group [1] libraries. We then transfer only the first $1/7$ bytes of the resulting image. In the client we convert the low fidelity progressive JPEG back into JPEG format and supply it to the browser as though it comprised the image at its highest fidelity. Finally, the prototype only transcodes images that are greater than a user specified threshold. The results reported in this paper reflect a threshold of 8 KB, below which it becomes cheaper to simply transmit an image rather than run the transcoder.

Figures 6, 7, and 8 show the latency for loading the HTML documents over 56 Kb/sec, 384 Kb/sec, and 10 Mb/sec network links. Figure 9 shows the data transfered to load the documents. The figures show latencies for native IE5 (*IE*), and for Puppeteer runs that load all the images at their original fidelity (*Original*), load only the first $1/7$ bytes of transcoded images (*ImagTrans*), and load transcoded images and gzip-compressed text (*FullTrans*).

Original is included to measure the overhead of Puppeteer. This configuration represents the worst possible case as it incurs the overhead of uncovering the document's skeleton but does not benefit from any adaptation. *Original* shows that the Puppeteer overhead is not significant for slower networks, with documents larger than 128 KB experiencing median overheads of 3.38%, 13.77%, and 125.81% in the 56 Kb/sec, 384 Kb/sec, and 10 Mb/sec respectively. Moreover the amount of data transfered by *Original* is comparable to *IE* with most documents larger than 128 KB experiencing overheads smaller than 2.53%.

ImagTrans shows that on 10 Mb/sec networks, transcoding is always detrimental to performance. In contrast, on 56 KB/sec and 384 KB/sec networks, roughly $2/3$ of the documents larger that 128 KB experience an average reduction in user-perceived latency of 73.81% and 49.58% for 56 KB/sec and 384 KB/sec, respectively. A closer examination revealed that the remaining $1/3$ of the documents, those seeing little improvement from transcoding, are composed mostly of formated HTML text and have little or no image content. To reduce the latency of these documents we added gzip text compression to the prototype. The *FullTrans* run shows that with image and text transcoding, Puppeteer achieves average reductions in latency for all documents larger than 128 KB, at 56 KB/sec and 384 KB/sec, of 76.42% and 50.13%, respectively.

The *IE* figures are similar to the plots of the previous section in that they show large variations in user-perceived latency at high network speeds and an alignment of data points into lines as the network speed decreases. This is again a result of the transition from experiments that are CPU bound (10 Mb/sec) to network bound (56 Kb/sec). As in PowerPoint, in the 10 Mb/sec case, user-perceived latency is highly dependent on parsing and rendering time. The large variations in latency result from the differences in content types in HTML documents. We observe that IE5 takes considerably longer to format large text sections than it takes to load images of comparable size. Consider the two document in figure 8 that have the labels A and B. While A is smaller than B (539 KB vs. 757 KB) it takes considerably longer to render. A closer examination revealed that A it is composed of only HTML text, while B has little text (5 KB) and its size is accounted for by several images.

Incremental rendering

In this section we extend the *FullTrans* method explained earlier by incrementally increasing the fidelity of the images displayed by the browser. This experiment comprises a significant departure from the original browser policy that loads single images incrementally, but fetches images in order, with at most four images being loaded at one time. Instead, Puppeteer loads all

[1] A useful property of a progressive image format, such as progressive JPEG, is that any prefix of the file for an image results in a complete, albeit lower quality, rendering of the image. As the prefix increases in length and approaches the full image file, the image quality approaches its maximum.

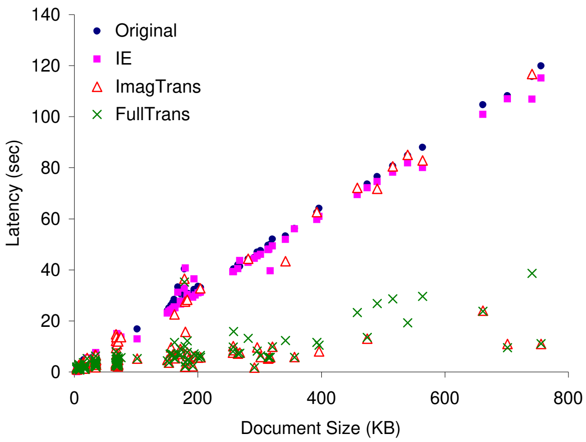

Figure 6: Load latency for HTML documents over 56 Kb/sec. Shown are latencies for native IE5 (*IE*), and Puppeteer runs that load all the images at their original fidelity (*Original*), load only the first 1/7 bytes of transcoded images (*ImagTrans*), load transcoded images and text (*FullTrans*).

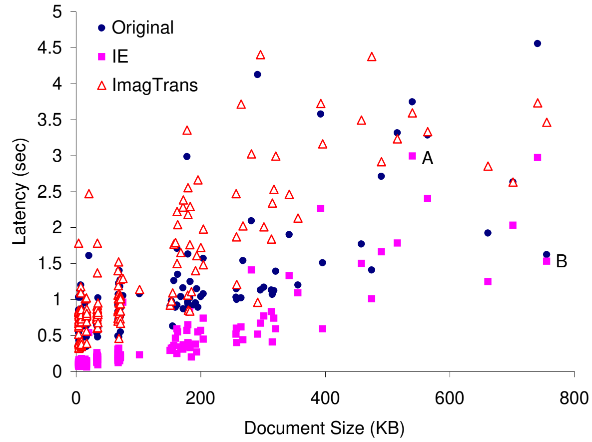

Figure 8: Load latency for HTML documents over 10 Mb/sec. Shown are latencies for native IE5 (*IE*), and Puppeteer runs that load all the images at their original fidelity (*Original*), load only the first 1/7 bytes of transcoded images (*ImagTrans*), load transcoded images and text (*FullTrans*).

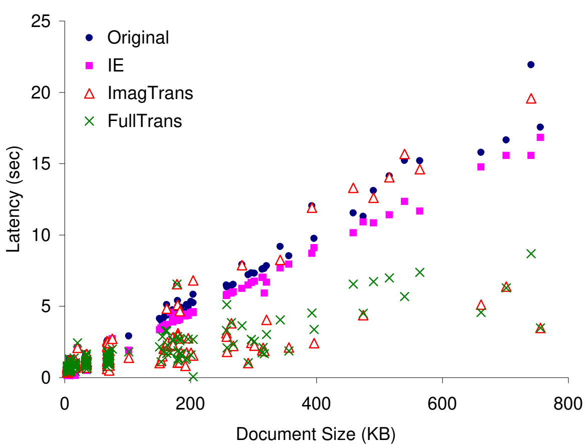

Figure 7: Load latency for HTML documents over 384 Kb/sec. Shown are latencies for native IE5 (*IE*), and Puppeteer runs that load all the images at their original fidelity (*Original*), load only the first 1/7 bytes of transcoded images (*ImagTrans*), load transcoded images and text (*FullTrans*).

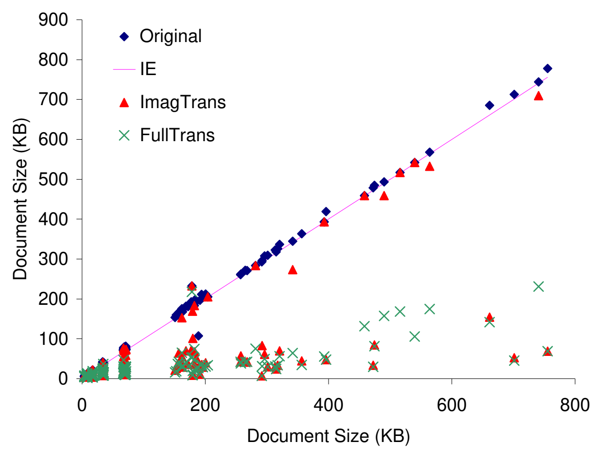

Figure 9: Data transfered to load HTML documents. Shown are data totals for native IE5 (*IE*), and Puppeteer runs that load all the images at their original fidelity (*Original*), load only the first 1/7 bytes of transcoded images (*ImagTrans*), load transcoded images and text (*FullTrans*).

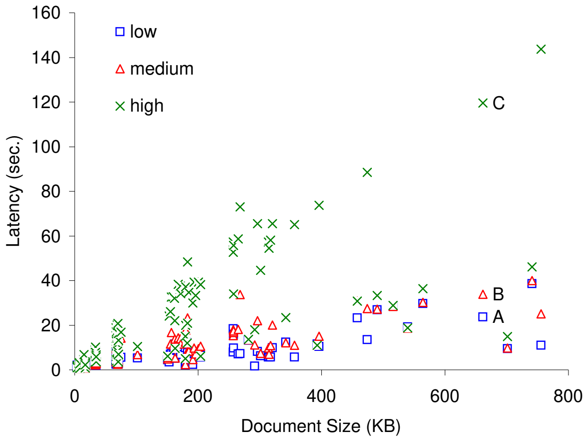

Figure 10: Incremental rendering of HTML documents over 56Kb/sec.

images at their lowest fidelity first and only then begins to increase image fidelity. fidelity level.

In this experiment we assume that images have 3 fidelity levels, which we will refer to as *low*, *medium*, and *high*. We chose these levels to correspond to the first $1/7$, next $2/7$, and last $4/7$ of the bytes in the progressive JPEG representation. For GIF images, we first convert them to JPEG. *Low* and *medium* are then the same as avoe. For *high*, however, we transmit the original GIF image. This will support GIF features not present in JPEG, such as animations and transparency.

Figure 10 shows the results of this experiment. For every document, it displays three aligned points with the latency required to load the document at its lowest fidelity (*e.g.*, point A in the plot) and then refined twice (*e.g.*, point B and C).

7.2 Puppeteer characterization

In this section we characterize the execution time and data transfers of Puppeteer. We first explore where Puppeteer spends its time. We then determine how much protocol data is transfered between the server and the client to adapt documents.

7.2.1 Time

Figures 11 and 12 show the breakdowns of execution time for PowerPoint and IE5. The PowerPoint plot shows results for 100 Mb/sec and 384 Kb/sec for loading all components in the documents (*Full*), and just the components of the first slide (*Slide*). The IE plot shows results for 384 Kb/sec and 56 Kb/sec for loading with (*Trans*) and without (*Orig*) image and text transcoding.

Each figure plots for various document' sizes the time spent transferring data (*Network*), parsing (*Parse*), and transferring the skeleton (*Skeleton*). The PowerPoint plot also shows the time spent rendering the document (*Render*), while the IE plot displays the time spent transcoding (*Transcode*) images and text.

The data for the *Full* and *Orig* runs show a clear trend where as we increase the size of the documents and decrease the speed of the network, the time spent transferring data becomes the prevalent contributor to user perceived latency. Overall the time spend by Puppeteer parsing, transmitting the skeleton, and transcoding varied for PowerPoint documents from 54.60% of total execution time on small documents on 100 Mb/sec to 2.53% on 384 Kb/sec for large documents; and for HTML documents from 37.38% over 384 Kb/sec for small document to 1.63% over 56 Kb/sec for large documents.

Figure 11 shows that the latency reduction achieved by the *Slide* run over 384 Kb/sec results mainly from a large reduction in network time. In contrast, over 100 Mb/sec the reduction in rendering time is the dominant factor in reducing overall latency.

Figure 12 shows that performing image and text transcoding on the fly is a good technique for reducing network latency. Overall transcoding time 11.5% to less than 1% of execution time. Moreover since Puppeteer overlaps image transcoding with data transmission, the overall effect on execution time diminishes as network speed decreases.

7.2.2 Data

Figure 13 plots the data breakdown for PowerPoint and HTML documents. We divide the data into application data and Puppeteer overhead, which we further decomposed into data transmitted to fetch the skeleton (*skeleton*) and data transmitted to request components (*fetch*). For this experiment we only consider the worst case scenario, which corresponds to requesting each component separately at its highest fidelity.

The data shows that the Puppeteer data overhead becomes less significant as document size increases. Overall the overhead varied for PowerPoint documents from 20.28% on small documents to just 2.94% on large docu-

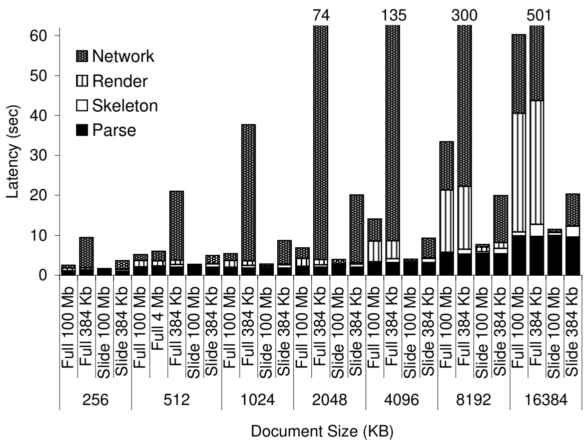

Figure 11: Breakdowns of execution time for PowerPoint with Puppeteer support for 100 Mb/sec and 384 Kb/sec for loading all components in the documents (*Full*), and just the components of the first slide (*Slide*) for local and remote documents.

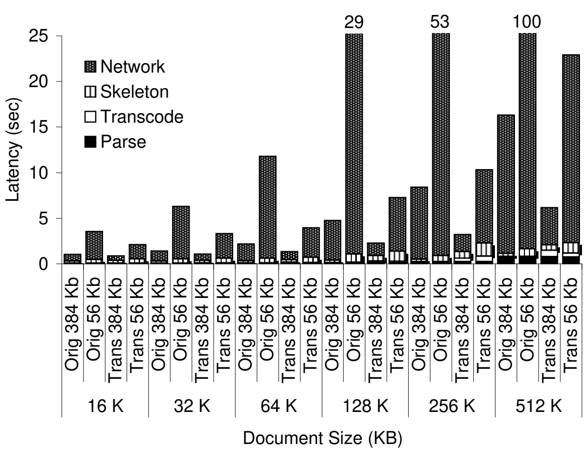

Figure 12: Breakdowns of execution time for IE with Puppeteer support for 384 Kb/sec and 56 Kb/sec for loading with (*Trans*) and without (*Orig*) image and text transcoding.

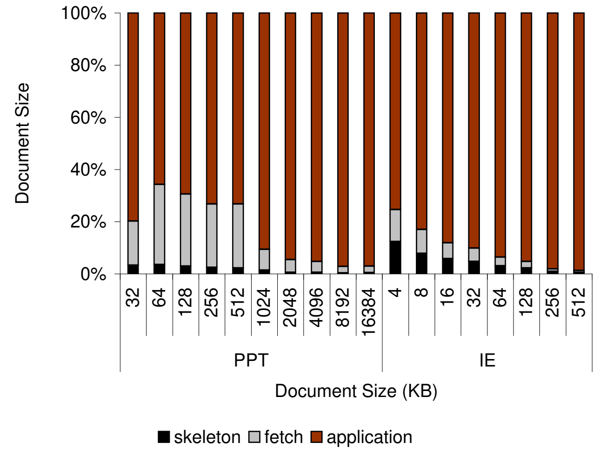

Figure 13: Data breakdowns for loading PowerPoint and HTML documents. The plot shows the proportion of application data, skeleton, and fetch data sent.

ments; and for HTML documents from 24.65% on small documents to 1.32% on large documents.

8 Future work

The Puppeteer project is just in it early stages. In the immediate future we plan to extend the prototype by adding support for writes and the ability to dynamically adapt to to the dynamically changing bandwidths that may occur in mobile computing environments.

The component-based approach of Puppeteer shows good promise for supporting writes. Puppeteer already tracks the fidelity of each component and should be able to support modifications, at least to components that have been transfered at high fidelity. If a document containing a low-fidelity component is edited, and the component is left unchanged, we expect we can make efficient use of network bandwidth by only saving the changes to the document, and allowing the server to reconstruct the final document from its original high-fidelity parts.

We also plan to explore the suitability of component-based adaptation for applications beyond those that arise in low-bandwidth environments. In particular, we are interested in using Puppeteer to adapt applications to collaborative work environments. We intend for Puppeteer's application drivers to stay relatively simple while adding new features to the adaptation logic. Our long-term goal is that, by simply writing a new application driver, a wide variety of adaptation behaviors become immediately available.

9 Related work

Transcoding has been used for some time now to customize information to the user's preferences [4, 8], reduce server load [2], or support thin clients [12].

Much work has gone into extending the system architecture to better support mobile clients [19] and to creating programming models that incorporate adaptation into the design of the application [15]. The project that most closely relates to Puppeteer is Odyssey [25], which splits the responsibility for adaptation between the application and the system. Puppeteer takes a similar approach, pushing common adaptation tasks into the system infrastructure and leaving the application-specific aspect of adaptation to application drivers. The main difference between the two systems lays in Puppeteer's use of existing run-time interfaces to adapt existing applications, whereas Odyssey requires applications to be modified to work with it.

Another project that uses similar ideas to Puppeteer is Dynamic Documents [17]. This instrumentation of the Mosaic Web browser uses Tcl scripts to set the policies for individual HTML documents. While Puppeteer uses the external interfaces provided by the application, Dynamic Documents use an internal script interpreter in the browser.

10 Conclusions and discussion

We presented the design and measured the effectiveness of Puppeteer, a system designed to adapt component-based applications, such as the Microsoft Office suite, to run in low-bandwidth conditions. Without modifying the application, Puppeteer supports complex adaptation policies that traditionally require significant modifications to the applications.

We demonstrated that a significant number of policies can be implemented efficiently using the run-time interfaces already supported by component-based applications. Our results show that for our two prototype applications, Puppeteer can achieve average reductions in user perceived latency of up to 84.22% for PowerPoint documents loaded over a 384 Kb/sec link and 76.42% for HTML documents loaded over 56 Kb/sec link.

Puppeteer's reliance on application specific drivers to provide tailored adaptation raises the question of porting new application to the system. In our experience applications drivers are not large, requiring just 1412 and 611 lines of Java code for PowerPoint and IE5. As these line counts suggest the size of the drivers and their development time is strongly related to the complexity of the application. Moreover, in our experience the bulk of the development time is spent understanding the application's file format and DMI. We estimate that a person already familiar with these features, for instance the application developer, could write a Puppeteer driver in just a couple of weeks.

Acknowledgments

Yuri Dotsenko set up the DummyNet network that we use in our experiments and ported the PBMPLUS and JPEG libraries that we used for image transcoding. Ping Tao downloaded and characterized the HTML documents.

References

[1] Independent JPEG Group. http://www.ijg.org/.

[2] ABDELZAHER, T. F., AND BHATTI, N. Web content adaptation to improve server overload behavior. In *Proceedings of The Eighth International World Wide Web Conference*.

[3] BAGRODIA, R., CHU, W. W., KLEINROCK, L., AND POPEK, G. Vision, issues, and architecture for nomadic computing. *IEEE Personal Communications 2*, 6 (Dec. 1995), 14–27.

[4] BARRETT, R., MAGLIO, P. P., AND KELLEM, D. C. How to personalize the web. In *Conference on Human Factors In Computing Systems (CHI 95)* (Denver, Colorado, May 1995).

[5] BROCKSCHMIDT, K. *Inside OLE*. Microsoft Press, 1995.

[6] CHAPPELL, D. *Understanding ActiveX and OLE*. Microsoft Press, 1996.

[7] CUNHA, C. R., BESTAVROS, A., AND CROVELLA, M. E. Characteristics of WWW client-based traces. Tech. Rep. TR-95-010, Boston University, Apr. 1995.

[8] DE LARA, E., PETERSEN, K., TERRY, D. B., LAMARCA, A., THORNTON, J., SALISBURY, M., DOURISH, P., EDWARDS, K., AND LAMPING, J. Caching documents with active properties. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems (HotOS-VII)* (Rio Rico, Arizona, Mar. 1999).

[9] DE LARA, E., WALLACH, D. S., AND ZWAENEPOEL, W. Opportunities for bandwidth adaptation in Microsoft Office documents. In *Proceedings of the Fourth USENIX Windows Symposium* (Seattle, Washington, Aug. 2000).

[10] DUCHAMP, D. Issues in wireless mobile computing. In *Proceedings of Third Workshop on Workstation Operating Systems* (Key Biscayne, Florida, Apr. 1992), pp. 1–7.

[11] FORMAN, G. H., AND ZAHORJAN, J. The challenges of mobile computing. *IEEE Computer* (Apr. 1994), 38–47.

[12] FOX, A., GRIBBLE, S. D., BREWER, E. A., AND AMIR, E. Adapting to network and client variability via on-demand dynamic distillation. *Sigplan Notices 31*, 9 (Sept. 1996), 160–170.

[13] FOX, A., GRIBBLE, S. D., CHAWATHE, Y., AND BREWER, E. A. Adapting to network and client variation using infrastructural proxies: Lessons and perspectives. *IEEE Personal Communications 5*, 4 (Aug. 1998), 10–19.

[14] GARDNER, D. Beginner's guide to DHTML. http://wsabstract.com/howto/dhtmlguide.shtml.

[15] JOSEPH, A. D., DELESPINASSE, A. F., TAUBER, J. A., GIFFORD, D. K., AND KAASHOEK, M. F. Rover: a toolkit for mobile information access. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)* (Copper Mountain Resort, Colorado, Dec. 1995), pp. 156–171.

[16] JOSEPH, A. D., TAUBER, J. A., AND KAASHOEK, M. F. Building reliable mobile-aware applications using the Rover toolkit. In *Proceedings of the 2nd ACM International Conference on Mobile Computing and Networking (MobiCom '96)* (Rye, New York, Nov. 1996).

[17] KAASHOEK, M. F., PINCKNEY, T., AND TAUBER, J. A. Dynamic documents: mobile wireless access to the WWW. In *Proceedings of the Workshop on Mobile Computing Systems and Applications (WMCSA '94)* (Santa Cruz, California, Dec. 1994), IEEE Computer Society, pp. 179–184.

[18] KATZ, R. H. Adaptation and mobility in wireless information systems. *IEEE Personal Communications 1*, 1 (1994), 6–17.

[19] KISTLER, J. J., AND SATYANARAYANAN, M. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems 10*, 1 (Feb. 1992), 3–25.

[20] MICROSOFT CORPORATION. *Microsoft Office 97 Drawing File Format*. Redmond, Washington, 1997. MSDN Online, `http://msdn.microsoft.com`.

[21] MICROSOFT CORPORATION. *Microsoft PowerPoint File Format*. Redmond, Washington, 1997. MSDN Online, `http://msdn.microsoft.com`.

[22] MICROSOFT CORPORATION. *Microsoft Office 2000 and HTML*. Redmond, Washington, 1999. MSDN Online, `http://msdn.microsoft.com`.

[23] MICROSOFT PRESS. *Microsoft Office 2000 / Visual Basic Programmer's Guide*, 1999.

[24] MUMMERT, L. B., EBLING, M. R., AND SATYANARAYANAN, M. Exploiting weak connectivity for mobile file access. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (Copper Mountain Resort, Colorado, Dec. 1995).

[25] NOBLE, B. D., SATYANARAYANAN, M., NARAYANAN, D., TILTON, J. E., FLINN, J., AND WALKER, K. R. Agile application-aware adaptation for mobility. *Operating Systems Review (ACM) 51*, 5 (Dec. 1997), 276–287.

[26] POSKANZER, J. PBMPLUS. http://www.acme.com/software/pbmplus.

[27] RIZZO, L. DummyNet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review* (Jan. 1997).

[28] ROBERTS, S. *Programming Microsoft Internet Explorer 5*. Microsoft Press, 1999.

[29] SATYANARAYANAN, M. Hot topics: Mobile computing. *IEEE Computer 26*, 9 (Sept. 1993), 81–82.

[30] SATYANARAYANAN, M. Fundamental challenges in mobile computing. In *Fifteenth ACM Symposium on Principles of Distributed Computing* (Philadelphia, Pennsylvania, May 1996).