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 exported interfaces of these applications and the structured nature of the documents they manipulate to perform adaptation *without* modifying the applications. The system is structured in a modular fashion, allowing easy addition of new applications and adaptation policies.

Our initial prototype focuses on adaptation to limited bandwidth. It runs on Windows NT, and includes support for a variety of adaptation policies for Microsoft PowerPoint and Internet Explorer 5. We demonstrate that Puppeteer can support complex policies without any modification to the application and with little overhead. To the best of our knowledge, previous implementations of adaptations of this nature have relied on modifying the application.

1 Introduction

The need for application adaptation in mobile and wireless environments is well established [7, 12, 13, 20, 27, 32, 33]. On one hand, mobile environments are characterized by low and unstable resource availability. On the other hand, mobile users often want to access remote data using the same applications they use on their desktop machines. Unfortunately, many of these desktop applications require a rich and stable resource environment. They perform poorly when used on mobile clients, and require adaptation to provide acceptable levels of service. Many approaches to adaptation have been proposed before, and many taxonomies of adaptation are possible. We focus here on the types of adaptation policies as well as on where the adaptation is implemented.

Adaptation policies can be grouped into two types: *data* and *control*. Data adaptations transform the application's data. For instance, they transform the images in a document into a lower resolution format. Control adaptations modify the application's control flow (i.e., its behavior). For instance, a control adaptation could cause an application that otherwise returns control to the user only after an entire document is loaded to return control as soon as the first page is loaded.

Based on where the adaptation is implemented, we recognize a spectrum of possibilities with two extremes: system-based [21, 26] and application-based adaptation [14, 15, 18, 34]. With system-based adaptation, the system performs all adaptation by interposing itself between the application and the data; no changes are made to the application. With application-based adaptation, only the application is changed; the system is unaware of any adaptation. Application-based adaptation allows both data and control adaptation, while system-based adaptation is limited to data adaptation. System-based adaptation does not require modification of the applications, and provides centralized control, allowing the system to adapt several applications according to a system-wide policy.

In this paper, we present a novel approach to adaptation we call *component-based adaptation*. It enables application-specific control and data adaptation policies *without* requiring modifications to the application. It does so by using the exposed APIs of component-based applications and the structured nature of the documents they manipulate to implement application-specific control adaptation policies. Component-based adaptation attempts to bring together the benefits of system-based and application-based adaptation, namely to implement application-specific policies without modifying the applications. Since adaptation is done in the system, component-based adaptation retains the advantage of providing a centralized locus of control for adaptation of multiple applications.

Component-based adaptation enables policies that adapt by repeated use of *subsetting* and *versioning*. A subsetting policy creates a new virtual document consisting of a subset of the components of the original document (e.g., the first slide in a presentation). A versioning policy chooses among the multiple instantiations of a component (e.g., instances of an image with different resolution). The adaptation policies use the application's exposed API to extend the subset or to replace the version

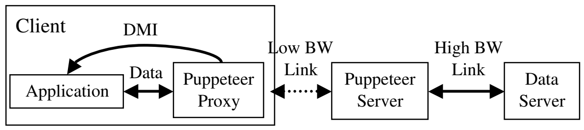

Figure 1: Overall system architecture.

of a component (e.g., load additional slides in a presentation or replace an image with one of higher fidelity). This iterative improvement is one of the key advantages of component-based adaptation over system-based adaptation.

Another approach that tries to strike a middle ground between system- and application-based adaptation is application-aware adaptation [6, 28]. Here, the system provides some common adaptation facilities, and serves as a centralized locus of control for the adaptation of all applications. The applications are modified to implement control adaptations and to perform calls to an adaptation API provided by the system. Component-based adaptation has similarities to application-aware adaptation in that both approaches delegate common adaptation tasks to the system. The approaches differ, however, in how control adaptation policies are implemented. In component-based adaptation, it is the applications that expose the interfaces, with the system invoking those interfaces to perform adaptation. The precise opposite occurs in application-aware adaptation where the applications are modified to call on the system's adaptation API. Component-based adaptation enables third parties to add new adaptation policies after the application has been released, while application-aware adaptation requires the application designer to foresee all necessary adaptations at the time the application is written.

Component-based adaptation is by nature restricted to component-based applications with exported APIs. While certainly a limitation, we observe that many desirable candidate applications for adaptation are already component-based, including the Microsoft Office Suite, Internet Explorer, Netscape Navigator, the KDE Office Suite, and Star Office. Recognizing the advantages of component-oriented software construction – independent of adaptation – we foresee an increasing number of applications being developed as components with exported APIs. Although traditionally associated with the Windows platform and with COM/DCOM technology, component-based technologies are becoming more common in the UNIX world as well, where the push for component-based technologies is led by the GNOME [1] and KDE [3] projects. A good example is KOffice [4], an open source productivity suite with powerful scripting capabilities. More recently, StarOffice [5] released version 5.2 of its popular cross-platform productivity suite, which implements a sophisticated object model that allows scripting by third party applications through a CORBA-based interface.

The more fundamental question about component-based adaptation is to what extent it can support the adaptation mechanisms that a customized application-based approach can achieve and with what performance. Furthermore, we wish to understand the scalability of component-based adaptation. Clearly, the system needs "drivers" for each application it wishes to support. For the concept to be scalable in terms of the number of applications it supports, the effort involved in writing an additional driver must be made small.

To address these questions, we have built a system we call Puppeteer. This paper describes the design of the Puppeteer system, its implementation on Windows NT, and our experience using this implementation to adapt two applications for low bandwidths, Microsoft PowerPoint (a presentation graphics system, hereafter "PowerPoint") and Internet Explorer 5 (a Web browser, hereafter "IE"). We demonstrate that Puppeteer can easily and efficiently support a number of desirable policies.

The rest of this paper is organized as follows. Section 2 presents the design of the Puppeteer system. Section 3 introduces the prototype implementation and the applications we use to evaluate it. Section 4 describes the experimental platform. Section 5 describes the documents we use in our experiments. Section 6 presents our experimental results. Section 7 discusses related work. Finally, Section 8 presents our conclusions.

2 Design

Figure 1 shows the four-tier Puppeteer system architecture. It consists of the application(s) to be adapted, the Puppeteer client proxy, the Puppeteer server proxy, and the data server. The application and data server are completely unmodified. The Puppeteer client proxy and server proxy work together to perform the adaptation.

The Puppeteer client proxy is in charge of executing the policies that adapt the applications. The Puppeteer server proxy is responsible for parsing documents, exposing their structure, and transcoding components as requested by the client proxy. The Puppeteer server proxy is assumed to have strong connectivity to the data server. In the most common scenario, it executes on the same machine as the data server. Data servers can be arbitrary repositories of data such as Web servers, file servers or databases.

2.1 Application Requirements

Puppeteer can adapt an application if it can uncover the component structure of its documents and if the appli-

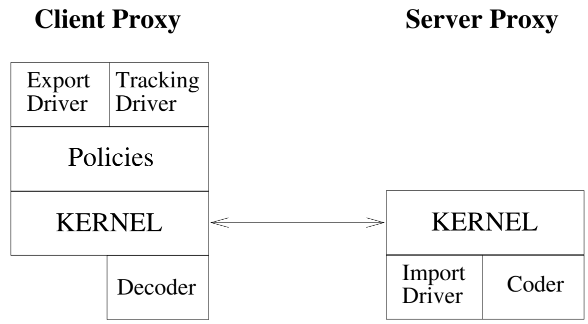

Figure 2: Internal Puppeteer architecture.

cation provides an API that enables Puppeteer to view and modify the data the application operates on. We refer to the latter feature as Data Manipulation Interface (*DMI*). Additionally, Puppeteer can benefit greatly from the ability to track the user's actions. We demonstrate next how Puppeteer implements adaptation once these requirements are met.

2.2 Puppeteer Architecture

The Puppeteer architecture consists of four types of modules: Kernel, Driver, Transcoder, and Policy (see Figure 2). The Kernel appears once in both the client and server Puppeteer proxy. A driver supports adaptation for a particular component type. A driver for a particular component type may call on a driver for another component type, if a component of the latter type is included in a component of the former type. At the top of this driver hierarchy sits the driver for a particular application (which itself is a component type). Drivers may execute both in the client and the server Puppeteer proxies, as may Transcoders which implement specific transformations on component types. Policies specify particular adaptation strategies and execute in the client Puppeteer proxy.

2.2.1 Kernel

The Kernel is a component-independent module that implements the Puppeteer protocol. The Kernel runs in both the client and server proxies and enables the transfer of document components. The Kernel does not have knowledge about the specifics of the documents being transmitted. It operates on a format-neutral description of the documents, which we refer to as the Puppeteer Intermediate Format (PIF). A PIF consists of a *skeleton* of *components*, each of which has a set of related *data items*. The skeleton 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 elements in the document. The skeleton is a multi-level data structure as components in any level can contain sub-components. The skeleton is component-independent, but components in the skeleton are component-specific. Component can have component-specific properties (e.g., slide title, image size) and one or more related data items that contain the component's native data.

When adapting a document, the Kernel first communicates the skeleton between the server and the client proxy. It then enables application policies to request a subset of the components and to specify transcoding filters to apply to the component's data. To improve performance, the Kernel batches requests for multiple components into a single message and supports asynchronous requests.

2.2.2 Drivers

For every component type it adapts, Puppeteer requires an import and an export driver. To implement complex policies, a tracking driver is also necessary. The import drivers parse the documents, extracting their component structure and converting them from their application-specific file formats to PIF.

In the common case where the application's 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(s) directly to uncover the structure of the data. This results in good performance, and enables clients and server to run on different platforms (e.g., running the Puppeteer client proxy on Windows NT while running the Puppeteer server proxy on Linux).

When the application only exposes a DMI, but has an opaque file format, Puppeteer runs an instance of the application on the server, and uses the DMI to uncover the structure of the data, in some sense using the application as a parser. This configuration allows for a high degree of flexibility and makes porting applications to Puppeteer more straightforward, since Puppeteer need not understand the application's file format. It creates, however, more overhead on the server proxy, and requires both the client and server to run the environment of the application, which in most cases amounts to running the same operating system on both servers and clients.

Parsing at the server does not work well for documents that choose what data to fetch and display by executing a script, or by other dynamic mechanisms. Instead, import drivers for dynamic content run in the Puppeteer client proxy, and rely on an intercept mechanism that traces requests.

Regardless of whether the skeleton is built statically in the server proxy or dynamically in the client proxy, any changes to the skeleton are reflected by the Kernel at

both ends to maintain a consistent view of the skeleton.
Export drivers un-parse the PIF and update the application using the DMI interfaces exposed by the application. A minimal export driver has to support inserting new components into a running application.
Tracking drivers are necessary for many complex policies. A tracking driver tracks which components are being viewed by the user and intercepts load and save requests. Tracking drivers can be implemented using polling or event registration mechanisms.

2.2.3 Transcoders

Puppeteer makes extensive use of transcoding to perform transformations on component data. Transcoders include the conventional ones, such as compression and reducing image resolution. A novel transcoding mechanism is used to enable loading subsets of components. Each element of the PIF skeleton has a number of associated data items that, among other things, encode in a component-specific format the relationship between the component and its children. To load a subset of the children of a given node, it is sometimes necessary to modify the data items associated with the parent node to reflect the fact that we are only loading some of its children. In effect, by transcoding the parent node's data items, we create a new temporary component that consists only of a subset of the children of the original component.

2.2.4 Policies

Policies are modules that run on the client proxy and control the fetching of components. These policies traverse the skeleton, choosing what components to fetch and with what fidelity.
Puppeteer provides support for two types of policies: general-purpose policies that are independent of the component type being adapted (e.g., prefetching) and component-specific policies that use their knowledge about the component to drive the adaptation (e.g., fetch the first page only).
Typical policies choose components and fidelities based on available bandwidth and user-specified preferences (e.g., fetch all text first). Other policies track the user (e.g., fetch the PowerPoint slide that currently has the user's focus and prefetch subsequent slides in the presentation), or react to the way the user moves through the document (e.g., if the user skips pages, the policy can drop components it was fetching and focus the available bandwidth on fetching components that will be visible to the user).
Regardless of whether the decision to fetch a component is made by a general-purpose policy or by a component-specific one, the actual data transfer is performed by the Kernel, relieving the policy from the intricacies of communication.

2.3 The Adaptation Process

The adaptation process in Puppeteer is divided roughly into three stages: parsing the document to uncover the structure of the data, fetching the initially selected components at specific fidelity levels and supplying those to the application, and, if the policy so specifies, updating the application with newly fetched data.
When the user opens a (static) document, the Kernel on the Puppeteer server proxy instantiates an import driver for the appropriate document type. The import driver parses the document, extracts its skeleton and data, and generates a PIF. The Kernel then transfers the document's skeleton to the Puppeteer client proxy. The policies running on the client proxy ask the Kernel to fetch an initial set of components at a specified fidelity. This set of components is supplied to the application in return to its open call. 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. The policies in the client proxy now decide what further components or version of components to fetch. They instruct the Kernel to do so, and then the client proxy uses the DMI to feed those newly fetched components to the application.

3 Prototype

We use PowerPoint and IE as the initial applications for our prototype. Besides being widely popular, these two applications comply with the requirements for DMI, parsable file formats, and tracking mechanism from Section 2.1. Furthermore, PowerPoint and IE have radically different DMIs. By supporting both, we are more likely to accurately design the interfaces between the Puppeteer Kernel and the component-specific aspects of the system. Next, we discuss the design of the drivers, transcoders, and policies that we have implemented to adapt these two applications. Table 1 shows the code line counts for the various modules.

3.1 Drivers

3.1.1 Import Drivers

PowerPoint 2000 supports two native file formats: the traditional binary format based on OLE archives [22, 23], and a new XML-based format [24]. We choose to base the PowerPoint import drivers on the XML representation because it contains roughly the same informa-

Module		Code Lines
Kernel		8600
PPT	Import Driver	1114
	Export Driver	807
	Track Driver	112
	Transcoders	392
	Policies	287
	Total	2712
IE	Import Driver	314
	Export Driver	347
	Track Driver	65
	Transcoders	149
	Policies	334
	Total	1209

Table 1: Code line counts for Kernel, PowerPoint (*PPT*) and IE modules .

tion as the binary format, and the human readable nature of XML makes it easier to parse and manipulate the document. We have implemented import drivers for the following component types: PowerPoint, Slide, Images, Sound, Embedded Objects.

For IE, while HTML is straightforward to parse, the introduction of JavaScript in DHTML [16] has allowed for documents whose structure can change dynamically. For DHTML, the import driver intercepts URL requests, allowing it to dynamically add new images and components to a Web page's skeleton (see Section 2.2.2). We have implemented import drivers for the following component types: IE, Images.

3.1.2 Export Drivers

PowerPoint and IE DMIs are based on the Component Object Model (COM) [8] and the Object Linking and Embedding (OLE) [9] standards. The interfaces they provide are reasonably well documented [25, 31] and have traditionally been used to extend the functionality of third-party applications.

The PowerPoint and IE DMIs provide excellent access to compose and modify internal data structures. To support the policies we have implemented for this paper, the PowerPoint export drivers includes support for opening and closing presentations, and for inserting slides, images and embedded objects. The IE export driver includes support for navigating to a URL and for reloading individual components of a page.

To update an object in IE, the IE export driver instructs IE to reload only the URL associated with the object. PowerPoint supports a cut-and-paste interface to update a presentation. To paste new components into an active PowerPoint presentation, *active*, the PowerPoint export driver creates a new PowerPoint presentation, *helper*, that consists only of the new components. The update process has two stages. In *Stage 1*, the driver instructs PowerPoint to load *helper*. In *Stage 2*, for every component in *helper*, the driver copies it to the clipboard, pastes it into *active*, and deletes any earlier version of the same component from *active*.

3.1.3 Tracking Drivers

PowerPoint's event notification mechanism is primitive and encompasses just a handful of large-granularity events like opening or closing of documents, making it inadequate for tracking the behavior of the user. The PowerPoint tracking driver relies, instead, on polling the DMI to determine the slide currently being displayed.

The IE tracking driver uses IE's rich event mechanism that allows third-party applications to register call-back functions for a wide range of events. The driver uses this interface to detect when the user types a URL, presses the back or forward buttons, clicks on a link, or moves the mouse over an image. The former events are used to instruct the Kernel to open a new HTML document, while the latter is used by policies to drive image fetching and fidelity refinement (*e.g.*, refine the image currently pointed by the mouse).

3.2 Transcoders

The above policies use the following transcoders:

1 **Slide selector**. Creates a virtual presentation consisting of specific slides.
2 **OLE selector**. Creates a new file that contains only a subset of selected embedded OLE objects (PowerPoint stores embedded OLE objects in single file).
3 **Progressive JPEG**. Converts GIF and JPEG images into Progressive JPEG and back to JPEG.
4 **GZIP compressor**. Compresses and uncompresses text and binary data using gzip.

3.3 Policies

This section presents some sample adaptation policies that illustrate the power of component-based adaptation. These policies would be difficult to implement in system-based adaptation, because they affect not only the data used by the application, but also its control flow. Such adaptation policies have, to the best of our knowledge, only been implemented by modifying the application. In Puppeteer, however, they are implemented by using the external APIs. As will be demonstrated in Section 6 these policies also result in significant benefit under limited bandwidth conditions.

1 **PowerPoint: First slide**. Fetch only the components of the first slide at their highest fidelity, and

return control to the user. Fetch the rest of the presentation in the background.

2 **PowerPoint: Prefetch text**. Fetch all slides, but leave out any images and embedded objects. Monitor the user and fetch images and embedded objects of the slide that has the focus.

3 **IE: Incremental rendering**. Convert all GIF and JPEG images in a HTML page into Progressive JPEG. Load only the first 1/7 of the image, before returning control to user. Refetch with progressively higher fidelity the image pointed by the mouse.

3.4 Adding New Functionality

To adapt a new application with Puppeteer we need to implement drivers, policies, and transcoders for each new component type that is not currently supported by Puppeteer. For example, to enable MS Word we need to add drivers for the Word component type, but we can reuse the drivers and transcoders for the image and embedded object component types that we have implemented for PowerPoint (see Table 1).

While the effort in adding new applications and new policies is limited by the modular design of Puppeteer, the lack of standard DMIs, event models, and file formats requires new drivers to be written. Designing such standard interfaces is part of our ongoing research.

4 Experimental 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 [30]. This setup allows us to control the bandwidth between client and server to emulate various network technologies. For each application, we use three different bandwidths: one at which the application is network-bound, one at which it is CPU-bound, and one in-between.

All our experiments access data stored on an Apache 1.3 Web server. For the experiments where we measure 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 proxy run on the same machine.

5 Data Sets

We select the set of PowerPoint documents used in our experiments from a collection of Microsoft Office documents that we characterized earlier [11]. The full collection includes 2,167 documents downloaded from 334 Web sites with sizes ranging from 20 KB to 21 MB. We obtain our HTML documents by re-executing the traces of Web client accesses collected and characterized by Cunha *et al.* [10]. These traces include accesses from two user groups made during a period of 7 months from November 1994 through May 1995. These traces have 46,830 unique URLs corresponding to 3,026 Web sites. For every URL that we are able to access (many pages had either disappeared or were corrupted), we download the HTML file and any images referenced by them. We do not download any documents linked from these pages. In this manner we acquire 3,796 HTML files and 15,329 images, comprising 89 MB of data downloaded from 1,009 sites. Documents range in size from a few bytes to 773 KB, including images.

Because these data sets are so large, transmitting them at low bandwidth takes prohibitively long. We therefore run our experiments on just 92 PowerPoint documents and 182 HTML documents. For those subsets, the longest experiment requires 138 minutes for PowerPoint, and 55 minutes for HTML. For completeness, however, we run one test over the full sets of both document types over a high-bandwidth network, verifying that our selected documents and the full document sets produce similar results.

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

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

6 Experimental Results

The fundamental question that we want to answer in this section is how much overhead we pay for doing the adaptation outside of the application, as opposed to by modifying the application. To answer this question in a definitive way, we would need to modify the original applications to add the adaptation behavior that we achieve with Puppeteer, and compare the resulting performance to the performance of the applications running with Puppeteer. This is not possible, since we do not have access to the source code of the applications. Instead, we present some experiments to measure the various factors contributing to the Puppeteer overhead.

This overhead consists of two elements: a one-time initial cost and a continuing cost. The one-time initial cost consists of the CPU time to parse the document to extract its PIF and the network time to transmit the skeleton and some additional control information. Continuing costs come from the overhead of the various DMI commands used to control the application. We assume that other costs, such as network transmission, transcoding, and rendering of application data are similar for both implementations.

The remainder of this section is organized as follows. First, we measure the one-time initial adaptation costs of Puppeteer. Second, we measure the continuing adaptation costs. Finally, we present several examples of policies that significantly reduce user-perceived latency.

6.1 Initial Adaptation Costs

To determine the one-time initial costs, we compare the latency of loading PowerPoint and HTML documents in their entirety using the native application (*PPT.native, IE.native*) and the application with Puppeteer support (*PPT.full, IE.full*). In the latter configuration, Puppeteer loads the document's skeleton and all its components at their highest fidelity. This policy represents the worst possible case as it incurs the overhead of parsing the document to obtain the PIF but does not benefit from any adaptation.

Figures 3 and 4 show the percentage overhead of PPT.full and IE.full over PPT.native and IE.native for a variety of document sizes and bandwidths. Overall, the Puppeteer overhead for PowerPoint documents varies from 2% for large documents over 384 Kb/sec to 57% for small documents over 10 Mb/sec, and for HTML documents from 4.7% for large documents over 56 Kb/sec. to 305% for small document over 10 Mb/sec. These results show that, for large documents transmitted over medium to slow speed networks, where adaptation would normally be used, the initial adaptation costs of Puppeteer are small compared to the total document loading time.

Figure 5 plots the data breakdown for PowerPoint and HTML documents. We divide the data into application data and Puppeteer overhead, which we further decompose into data transmitted to fetch the skeleton (*skeleton*) and data transmitted to request components (*control*). This data confirms the results of Figures 3 and 4. The Puppeteer data overhead becomes less significant as document size increases. The data overhead varies for PowerPoint documents from 2.9% on large documents to 34% on small documents, and for HTML documents from 1.3% on large documents to 25% on small docu-

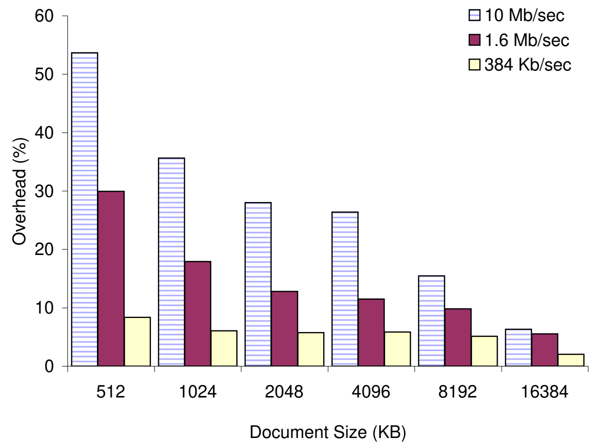

Figure 3: Percentage overhead of PPT.full over PPT.native for various document sizes and bandwidths.

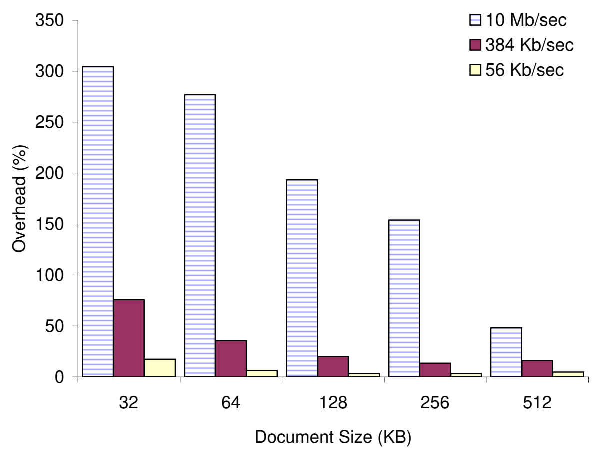

Figure 4: Percentage overhead of IE.full over IE.native for various document sizes and bandwidths.

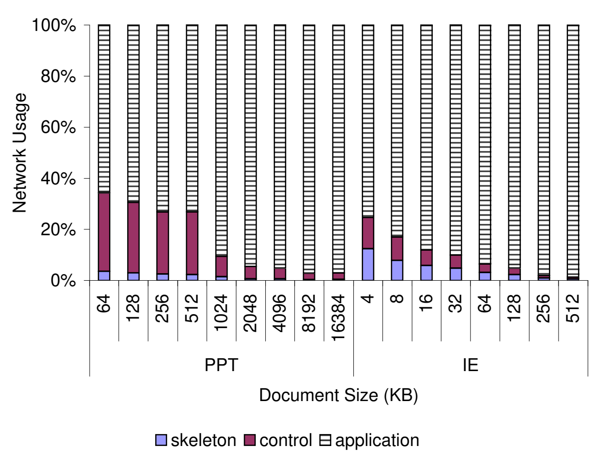

Figure 5: Data breakdowns for loading PowerPoint and HTML documents.

Operation		Cost (ms / component) Single Avg	Stdev	Additional Avg	Stdev
Slide (PPT)	Stage 1	746	723	417	492
	Stage 2	148	96	113	99
Image (IE)	Synthetic	N/A	N/A	29	9
	DMI	33	19	32	12

Table 2: Continuing adaptation costs for PowerPoint (*PPT*) slides and IE images. The table shows the cost of executing OLE calls that append PowerPoint slides or upgrade the fidelity of IE images

ments.

6.2 Continuing Adaptation Costs

The continuing costs of adapting using the DMI are clearly dependent on the application and the policy. Our purpose is not to give a comprehensive analysis of DMI-related adaptation costs, but to show that they are small compared to the network and rendering times inherent in the application. We perform two experiments: loading and pasting newly fetched slides into a PowerPoint presentation, and replacing all the images of an HTML page with higher fidelity versions. To prevent network effects from affecting our measurements we make sure that the data is present locally at the client before we load it into the application.

We determine the PowerPoint DMI overhead by measuring the time that the PowerPoint export driver spends loading the new slides, *Stage 1*, and cutting and pasting, *Stage 2*, as described in section 3.1.2. We expect that an in-application approach to adaptation would have to perform *Stage 1*, but would not need to perform *Stage 2*. For IE, we determine the DMI overhead for upgrading the images in two different ways: *DMI*, which uses the DMI to update the images; and *Synthetic*, which approximates an in-application adaptation approach. *Synthetic* measures the time to load and render previously generated pages that already contain the high fidelity images. *Synthetic* is not a perfect imitation of in-application adaptation, because it requires IE to re-load and parse the HTML portion of the page, which an in-application approach could dispense with. We avoid this problem by using only pages where the HTML content is very small (less than 5% of total page size), so that HTML parsing and rendering costs are minimal.

Table 2 shows the results of these experiments. For each policy, it shows the cost of updating a *single* component (i.e., one slide or one image) and the *additional* cost incurred by every extra component that is updated simultaneously. For PowerPoint, the table shows the time spent in *Stage 1* and *Stage 2*. For IE, the table shows the times for the *DMI* and *Synthetic* implementations.

The PowerPoint results show that the time spent cutting and pasting, *Stage 2*, is small compared to the time spent loading slides, *Stage 1*, which an in-application also has to carry out. Moreover, the time spent updating the application (*Stage 1* + *Stage 2*) is small compared to the network time. For example, the average network time to load a slide over the 384 Kb/sec network is 2994 milliseconds, with a standard deviation of 3943 milliseconds, while the average time for updating the application with a single slide is 994 milliseconds, with a standard deviation of 819 milliseconds.

The IE results show that the *DMI* implementation comes within 10% of *Synthetic*. Moreover, the image update times are small compared to the average network time. For instance, the average time to load an image over a 56 Kb/sec network is 565 milliseconds with a standard deviation of 635 milliseconds, compared to updating the application which takes on average 33 milliseconds with a standard deviation of 19 milliseconds.

The above results suggest that the cost of using DMI calls for adaptation is small, and that most of the time that it takes to add or upgrade a component is spent transferring the data over the network and loading it into the application. These two factors are expected to be similar whether we implement adaptation outside or inside the application.

6.3 Some Adaptation Policies

We conclude this section by presenting the results, as the end user would perceive them, of some of the Puppeteer adaptation policies we have implemented so far (see Section 3.3). These results also provide some indication of the circumstances under which these adaptations are profitable.

6.3.1 PowerPoint: Fetch First Slide and Text

In this experiment we measure the latency for a PowerPoint adaptation policy that loads only the first slide and the titles of all other slides of a PowerPoint presentation before it returns control to the user, and afterwards loads the remaining slides in the background. We also present results for an adaptation policy that, in addition, fetches all of the text in the PowerPoint document before returning control. With these adaptations, user-perceived latency is much reduced compared to the application policy of loading the entire document before returning control to the user.

The results of these experiments appear, under the labels *PPT.slide* and *PPT.slide+text*, respectively, in Figures 6, 7, and 8 for 384 Kb/sec, 1.6 Mb/sec, and 10 Mb/sec network links. Figure 9 shows the data trans-

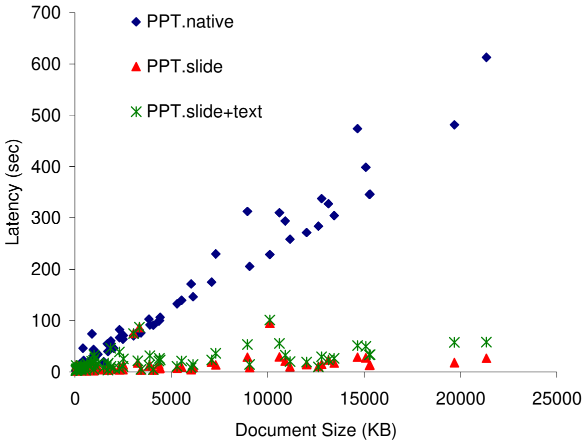

Figure 6: Load latency for PowerPoint documents at 384 Kb/sec. Shown are latencies for native PowerPoint (*PPT.native*), and Puppeteer runs for loading just the components of the first slide (*PPT.slide*), and loading, in addition, the text of all slides (*PPT.slide+text*).

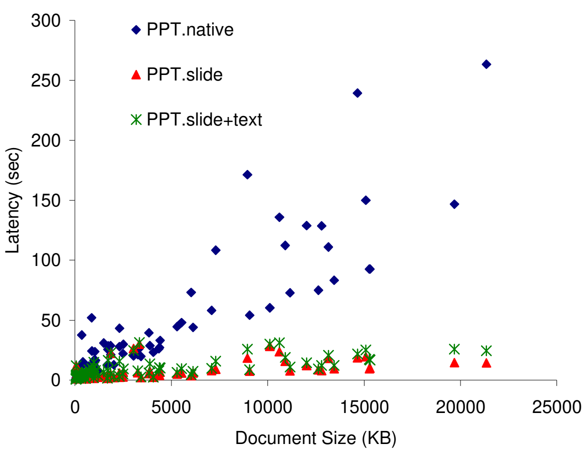

Figure 7: Load latency for PowerPoint documents at 1.6 Mb/sec.

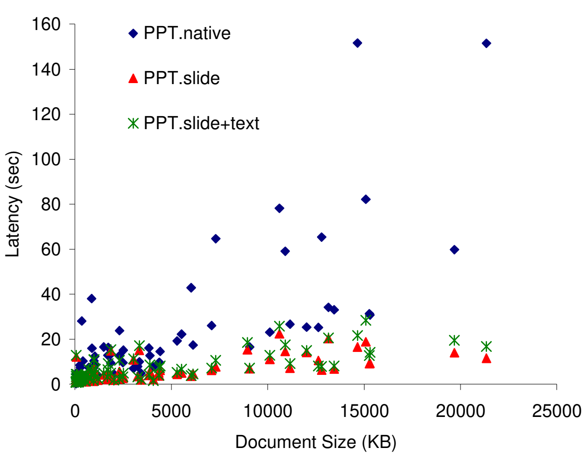

Figure 8: Load latency for PowerPoint documents at 10 Mb/sec.

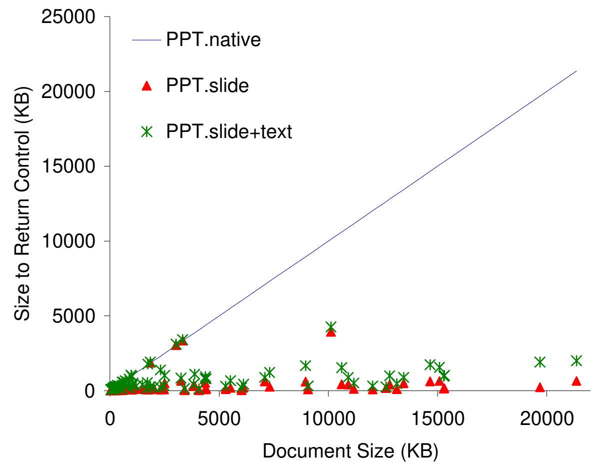

Figure 9: Data transfered to load PowerPoint documents.

fered in each of the three scenarios. For each document, the figures contain three vertically aligned points representing the latency or data measurements in three system configurations: native PowerPoint (*PPT.native*), Puppeteer loading only the components of the first slide and the titles of all other slides (*PPT.slide*), and Puppeteer loading in addition the text for all remaining slides (*PPT.slide+text*).

We expect that reduced network traffic would improve latency with the slower 384 Kb/sec and 1.6 Mb/sec networks. The savings over the 10 Mb/sec network come as a surprise. While Puppeteer achieves most of its savings on the 384 Kb/sec and 1.6 Mb/sec networks by reducing network traffic, the transmission times over the 10 Mb/sec are too small to account for the savings. The savings result, instead, from reducing the parsing and rendering time.

On average, *PPT.slide* achieves latency reductions of 86%, 78%, and 62% for documents larger than 1 MB on 384 Kb/sec, 1.6 Mb/sec, and 10 Mb/sec networks, respectively. The data in Figure 9 also 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).

When comparing the data points of *PPT.slide+text* to *PPT.slide*, we see that the latency has moved up only slightly. The latency is still significantly lower than for *PPT.native*, achieving savings of, on average, 75%, 72%, and 54% for documents larger than 1 MB over 384 Kb/sec, 1.6 Mb/sec, and 10 Mb/sec networks, respectively. Moreover, the increase in the amount of data transfered, especially for documents larger than 4 MB, is small, amounting to only an extra 6.4% above the data sent for the first slide. These results are consistent with our earlier findings [11] that text accounts for

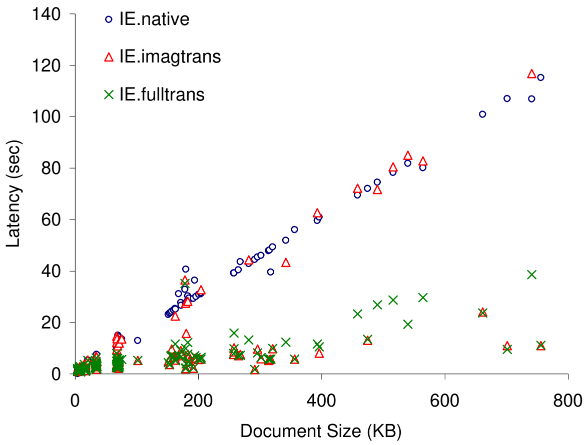

Figure 10: Load latency for HTML documents at 56 Kb/sec. Shown are latencies for native IE (*IE.native*), and Puppeteer runs that load only the first 1/7 bytes of transcoded images (*IE.imagtrans*), load transcoded images and text (*IE.fulltrans*).

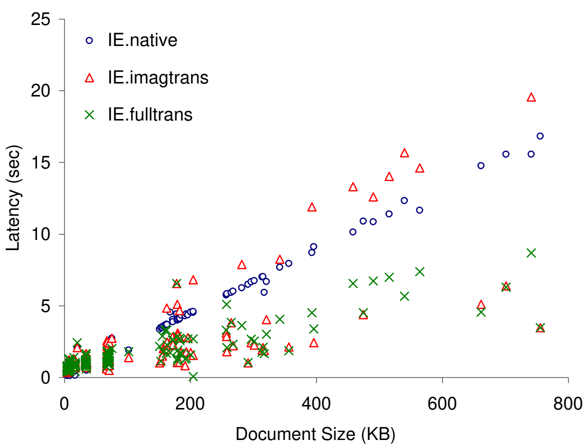

Figure 11: Load latency for HTML documents at 384 Kb/sec.

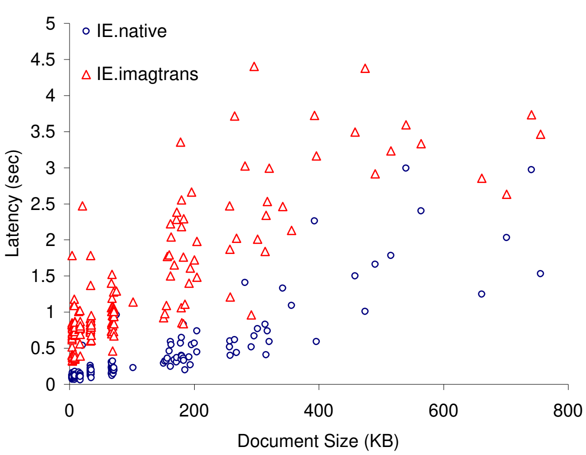

Figure 12: Load latency for HTML documents at 10 Mb/sec.

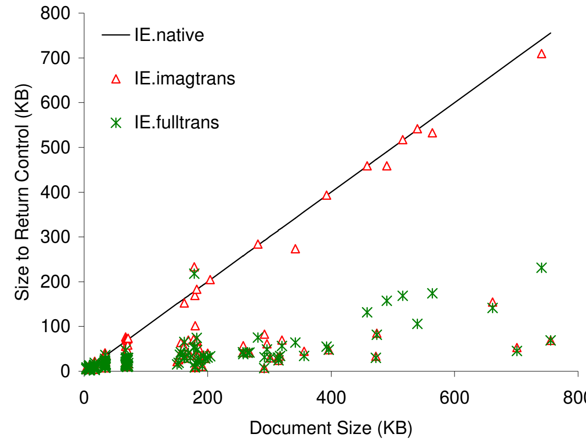

Figure 13: Data transfered to load HTML documents.

only a small fraction of the total data in large PowerPoint documents. These results suggest that text should be fetched in almost all situations and that the lazy fetching of components is more appropriate for the larger image and OLE embedded objects that appear in the documents.

Finally, an interesting characteristic of the figures is the large variation in user-perceived latency at high network speeds versus the alignment of data points into straight lines as the network speed decreases. The high variability at high network speeds results from the experiment being CPU-bound. Under these conditions, user-perceived latency is mostly dependent on the time that it takes PowerPoint to parse and render the presentation. For PowerPoint, 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 objects) in the presentation.

6.3.2 IE: 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 display a page by lowering the fidelity of some of the page's elements.

Our prototype converts, at run time, GIF and JPEG images embedded in an HTML document into progressive JPEG format[1] using the PBMPlus [29] and Independent JPEG Group [2] libraries. We then transfer only the first 1/7th of the resulting image's bytes. In the client we convert the low-fidelity progressive JPEG back into normal JPEG format and supply it to the browser as though

[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.

it comprised the image at its highest fidelity. Finally, the prototype only transcodes images that are greater than a user-specified size threshold. The results reported in this paper reflect a threshold size of 8 KB, below which it becomes cheaper to simply transmit an image rather than run the transcoder.

Figures 10, 11, and 12 show the latency for loading the HTML documents over 56 Kb/sec, 384 Kb/sec, and 10 Mb/sec networks. Figure 13 shows the data transfered to load the documents. The figures show latencies for native IE (*IE.native*), and for Puppeteer runs that load only the first $1/7$ bytes of transcoded images (*IE.imagtrans*), and load transcoded images and gzip-compressed text (*IE.fulltrans*).

IE.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, Puppeteer achieves an average reduction in latency for documents larger than 128 KB of 59% and 35% for 56 KB/sec and 384 KB/sec, respectively. A closer examination reveals that roughly $2/3$ of the documents see some latency reduction. The remaining $1/3$ of the documents, those seeing little improvement from transcoding, are composed mostly of HTML text and have little or no image content. To reduce the latency of these documents we add gzip text compression to the prototype. The *IE.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% and 50%, respectively.

Overall transcoding time takes between 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.

As with PowerPoint, we notice in the figures for IE that for low bandwidths the data points tend to fall in a straight line, while for higher bandwidths the data points become more dispersed. The reason is the same as for PowerPoint: At high bandwidths the experiment becomes CPU-bound and governed by the time it takes IE to parse and render the page. For IE, parsing and rendering time depends on the content types in the HTML document.

7 Related Work

Much work has gone into supporting mobile clients [21] and into creating programming models that incorporate adaptation into the design of the application [17]. The project that most closely relates to Puppeteer is Odyssey [28], 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.

Visual Proxies [34], an offspring of Odyssey, implements application-specific adaptation policies without modifying the application by using interposition between the X-server and the application. While this technique enables many adaptations that are possible with Puppeteer, it requires much more complicated application drivers.

The Dynamic Documents [19] system uses instrumentation of the Mosaic Web browser by Tcl scripts to set the policies for individual HTML documents. While Puppeteer uses the external interfaces provided by the application, Dynamic Documents uses an internal script interpreter in the browser.

8 Conclusions

We presented the design and measured the effectiveness of Puppeteer, a system for adapting component-based applications in mobile environments. Puppeteer implements adaptation by using the exposed APIs of component-based applications, enabling application-specific adaptation policies *without* requiring modifications to the application.

We described the architecture of Puppeteer and its implementation. The architecture allows for the modular addition of new applications, component types, transcoders, and policies. We demonstrated that complex policies, that traditionally require significant application modifications, can be implemented easily and efficiently in Puppeteer.

Puppeteer's reliance on application specific drivers to provide tailored adaptation raises the question of porting new application to the system. In our experience, the most time consuming part of porting an application is building the import driver that builds a PIF of the document by parsing the application specific file format. Once we had the necessary import and export drivers (the export drivers where considerably easier to implement), implementing policies proved surprisingly simple. In fact, most policies required less than a 50 lines of code.

With respect to standard file formats, the current trend towards XML-based formats has good promise. The only requirement is that components and their dependencies be made explicit. While we found the effort required to build export drivers to be modest, we are developing a set of standard APIs suitable for adaptation, including facilities for data manipulation and event registration.

References

[1] *GNOME*. `http://www.gnome.org`.

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

[3] *KDE*. `http://www.kde.org`.

[4] *KOffice*. `http://koffice.kde.org`.

[5] *StarOffice*. `http://www.stardivision.com`.

[6] D. Andersen, D. Basal, D. Curtis, S. Srinivasan, and H. Balakrishnan. System support for bandwidth management and content adaptation in Internet applications. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, San Diego, CA, Octuber 2000.

[7] Rajive Bagrodia, Wesley W. Chu, Leonard Kleinrock, and Gerald Popek. Vision, issues, and architecture for nomadic computing. *IEEE Personal Communications*, 2(6):14–27, December 1995.

[8] Kraig Brockschmidt. *Inside OLE*. Microsoft Press, 1995.

[9] David Chappell. *Understanding ActiveX and OLE*. Microsoft Press, 1996.

[10] Carlos R. Cunha, Azer Bestavros, and Mark E. Crovella. Characteristics of WWW client-based traces. Technical Report TR-95-010, Boston University, April 1995.

[11] Eyal de Lara, Dan S. Wallach, and Willy Zwaenepoel. Opportunities for bandwidth adaptation in Microsoft Office documents. In *Proceedings of the Fourth USENIX Windows Symposium*, Seattle, Washington, August 2000.

[12] Dan Duchamp. Issues in wireless mobile computing. In *Proceedings of Third Workshop on Workstation Operating Systems*, pages 1–7, Key Biscayne, Florida, April 1992.

[13] G H. Forman and J Zahorjan. The challenges of mobile computing. *IEEE Computer*, pages 38–47, April 1994.

[14] A. Fox, S. D. Gribble, E. A. Brewer, and E. Amir. Adapting to network and client variability via on-demand dynamic distillation. *Sigplan Notices*, 31(9):160–170, September 1996.

[15] A. Fox, S. D. Gribble, Y. Chawathe, and E. A. Brewer. Adapting to network and client variation using infrastructural proxies: Lessons and perspectives. *IEEE Personal Communications*, 5(4):10–19, August 1998.

[16] D. Gardner. Beginner's guide to DHTML. http://wsabstract.com/howto/dhtmlguide.shtml.

[17] Anthony D. Joseph, Alan F. deLespinasse, Joshua A. Tauber, David K. Gifford, and M. Frans Kaashoek. Rover: a toolkit for mobile information access. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 156–171, Copper Mountain Resort, Colorado, December 1995.

[18] Anthony D. Joseph, Joshua A. Tauber, and M. Frans Kaashoek. 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, November 1996.

[19] M. Frans Kaashoek, Tom Pinckney, and Joshua A. Tauber. Dynamic documents: mobile wireless access to the WWW. In *Proceedings of the Workshop on Mobile Computing Systems and Applications (WMCSA '94)*, pages 179–184, Santa Cruz, California, December 1994. IEEE Computer Society.

[20] Randy H. Katz. Adaptation and mobility in wireless information systems. *IEEE Personal Communications*, 1(1):6–17, 1994.

[21] James J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, February 1992.

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

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

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

[25] Microsoft Press. *Microsoft Office 2000 / Visual Basic Programmer's Guide*, 1999.

[26] Lily B. Mummert, Maria R. Ebling, and M. Satyanarayanan. Exploiting weak connectivity for mobile file access. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Copper Mountain Resort, Colorado, December 1995.

[27] B. Noble and M. Satyanarayanan. A research status report on adaptation for mobile data access. In *SIGMOD Record*, volume 24, December 1995.

[28] Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile application-aware adaptation for mobility. *Operating Systems Review (ACM)*, 51(5):276–287, December 1997.

[29] Jeff Poskanzer. PBMPLUS. http://www.acme.com/software/pbmplus.

[30] L. Rizzo. DummyNet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, January 1997.

[31] Scott Roberts. *Programming Microsoft Internet Explorer 5*. Microsoft Press, 1999.

[32] M. Satyanarayanan. Hot topics: Mobile computing. *IEEE Computer*, 26(9):81–82, September 1993.

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

[34] M. Satyanarayanan, J. Flinn, and K. R. Walker. Visual proxy: Exploiting OS customizations without application source code. *Operating Systems Review*, 33(3), July 1999.