Iterative Adaptation for Mobile Clients Using Existing APIs

Eyal de Lara, *Member*, *IEEE*, Yogesh Chopra, Rajnish Kumar, *Student Member*, *IEEE*, Nilesh Vaghela, Dan S. Wallach, *Member*, *IEEE*, and Willy Zwaenepoel, *Fellow*, *IEEE*

Abstract—Iterative Adaptation is a novel approach to adaptation for resource-limited mobile and wireless environments that supports powerful application-specific adaptations *without* requiring modifications to the application's source code. Common productivity applications, such as browsers, word processors, and presentation tools, export APIs that allow external applications to control their operation. The novel premise in iterative adaptation is that these APIs are sufficient to support a wide range of adaptation policies for applications running on resource-limited devices. In addition to allowing adaptation without having to change the application's source code, this approach has a unique combination of advantages. First, it supports centralized management of resources across multiple applications. Second, it makes it possible to modify application behavior after the application has been deployed. This paper evaluates the extent to which existing APIs can be used for the purposes of adapting document-based applications to run on bandwidth-limited devices. In particular, we implement a large number of bandwidth adaptations for applications from the Microsoft Office and the OpenOffice productivity suites and for Internet Explorer. Although we find limitations in their APIs, we are able to implement many adaptation policies without much complexity and with good performance. Moreover, iterative adaptation achieves performance similar to an approach that implements adaptation by modifying the application, while requiring only a fraction of the coding effort.

Index Terms—Application adaptation, low-bandwidth operation, pervasive computing, middleware.

1 Introduction

The need for application adaptation in mobile and wireless environments is well established [1], [2], [3], [4], [5]. Desktop applications, such as office productivity suites, typically expect that resources such as bandwidth and power are available in abundance [6]. In contrast, mobile and wireless environments are characterized by limited and unreliable resource availability, requiring the applications to be adapted to perform properly in these environments. While there has been considerable research on adapting applications to mobile and wireless environments, very few adaptation systems have been deployed, because existing approaches require extensive application source code modification [7], [8], [9] or have limited adaptive power [10], [11].

This paper presents *Iterative Adaptation*, a novel approach to adaptation that supports powerful application-specific adaptations *without* requiring modifications to the application's source code. The novel premise in iterative adaptation is that applications provide mechanisms to enable adaptation by exporting runtime Application Programming Interfaces (APIs) to external programs. In iterative adaptation, the adaptation system adapts applications by calling their APIs, instead of changing their source code. Iterative adaptation allows sophisticated adaptations that iteratively improve the content that the application provides to the user. For example, when browsing the Web on a mobile device, the adaptation system reduces the initial time to load a Web page by providing low-fidelity versions of its images. Later, as the user reads the page, the adaptation system acquires higher-fidelity images and uses the browser's API to replace the original images. Applications thus become artifacts that can be manipulated by the adaptation system. This *iterative improvement* has not been available previously, except in applications expressly designed to include it from the beginning. Because no source code modifications are necessary, iterative adaptation overcomes the principal roadblock to deploying adaptation.

Modifying applications for the purpose of adaptation is at best unattractive, because of the complex nature of many of the applications, and may be impossible because the source is not available or the application has already been deployed. Furthermore, embedding adaptation policies in the application requires the application designer to foresee all necessary policies at the time the application is written. Given that adaptation policy may depend not only on operating environments, but also on the mix of applications running on the device, such policy decisions should not be limited to application design time. Iterative adaptation thus provides a proper division of policy and mechanism between the adaptation system and the application. Finally, this policy mechanism division has the additional benefit that adaptation can be centralized. This allows for a single specification of a policy to be reused for a variety of applications (e.g., load all text first), and for the implementation of system-wide

- *E. de Lara is with the Department of Computer Science, University of Toronto, 10 King's College Road, Room 3302, Toronto, Ontario M5S 3G4, Canada. E-mail: delara@cs.toronto.edu.*
- *Y. Chopra and N. Vaghela are with iMimic Networking, Inc., 2211 Norfolk St., Suite 626, Houston, TX 77098. E-mail: {yogesh, nilesh}@imimic.com.*
- *R. Kumar is with the College of Computing, Georgia Institute of Technology, 801 Atlantic Drive, Atlanta, GA 30332-0280. E-mail: rajnish@cc.gatech.edu.*
- *D.S. Walach is with the Department of Computer Science, Rice University, 3121 Duncan Hall, 6100 Main Street, Houston, TX 77005. E-mail: dwallach@rice.edu.*
- *W. Zwaenepoel is with the School of Computer and Communication Sciences, EPFL, Swiss Federal Institute of Technology, Building BC 407—Station 14, CH-1015, Lausanne, Switzerland. E-mail: willy.zwaenepoel@epfl.ch.*

Manuscript received 22 Aug. 2003; revised 2 July 2004; accepted 22 Oct. 2004; published online 22 Aug. 2005.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0144-0803.

adaptation policies (e.g., for all applications, load all text first), which take into account the mix of applications running on the host [12].

Many popular document-based applications, including office productivity suites and browsers, already export well-documented APIs [13], [14]. In this paper, we explore the extent to which these existing APIs can support adaptation. To the best of our knowledge, this is the first effort to use these interfaces for adaptation. In this paper, we focus on adapting document-based applications to reduce latency for reading multimedia documents over bandwidth-limited networks, but the same principles are applicable to reducing power usage [15].

To investigate these issues, we implement iterative adaptation in the Puppeteer system [16]. We call our system Puppeteer because it uses the exported API's of the applications—the puppets—as strings to control their behavior. Puppeteer has a modular architecture that allows adding platforms, applications, and adaptation policies with modest effort. Using Puppeteer, we adapt a significant number of document-based applications on different platforms in a relatively short time. In particular, we implement Puppeteer on Linux and Windows. On Linux, we experiment with adaptation policies for Presentation and Writer from the OpenOffice productivity suite. On Windows, we use Outlook, PowerPoint, and Word from Microsoft Office and Internet Explorer. Much of the code is reused between different platforms and applications. Furthermore, similar policies for different applications (e.g., load all text first) can often be implemented without writing additional code. Puppeteer achieves large reductions in download latency and bandwidth consumption and adds little overhead when no adaptation is done. Moreover, Puppeteer achieves performance similar to that of an approach that implements adaptation by modifying the application, while requiring just a fraction of the coding effort.

The rest of this paper is organized as follows: Section 2 describes the capabilities of iterative adaptation and the requirements it places on the applications. Section 3 presents the design and implementation of the Puppeteer system. Section 4 reflects on our experience using APIs of existing applications for bandwidth adaptation. Specifically, we describe the limitations, from the point of view of adaptation, in the exported APIs and file formats of the applications. Section 5 measures the effectiveness and the overhead of some sample adaptation policies and compares Puppeteer's performance to an approach that implements adaptation within the application by modifying its source code. Section 6 discusses how iterative adaptation differs from previous approaches to content adaptation. Finally, Section 7 concludes the paper.

2 Iterative Adaptation

Bandwidth adaptations 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-fidelity format. Control adaptations modify the application's control flow (i.e., its behavior). For example, 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. Data adaptations are usually implemented by interposing a proxy between the data source and the resource-limited device, without modifications to the client application. Control adaptations, however, have traditionally required modifications to the client application.

Iterative adaptation is a novel technique that supports both data and control adaptations *without* requiring modifications to the application's source code. Iterative adaptation implements control adaptation *indirectly* by using exported APIs to carefully control the data on which an application operates. For example, iterative adaptation adapts an application that returns control to the user only after loading an entire document by providing to the application an initial (shorter) version of the document that consists of just a few pages. Later, as the user interacts with the application, iterative adaptation can fetch the remaining pages and use the application's exported API to extend the application's version of the document. In effect, iterative adaptation does not change the application's algorithms, but instead fools it into believing that the initial document is shorter than its real size.

Iterative adaptation takes advantage of the modular structure of modern multimedia document formats (e.g., HTML, XML) for the purposes of adaptation. In the rest of this document, the term *component* refers to a clearly identifiable part of a modular multimedia document, such as an image or a page in a manuscript. Iterative adaptation adapts applications by *repeated* use of two techniques: *subsetting* and *fidelity versioning*. Subsetting creates a new virtual document consisting of a subset of the components of the original document (e.g., the first slide in a presentation). Fidelity versioning[1] chooses among multiple transcoded views of a component (e.g., instances of an image with different resolution). Iterative adaptation uses the application's exported API to extend the subset or to replace the version of a component (e.g., load additional slides in a presentation or replace an image with one of higher fidelity). This *iterative improvement* has not been available previously except in applications expressly designed to support it from the beginning and is one of the key advantages of iterative adaptation.

In the rest of this section, we first explore the types of adaptation policies that iterative adaptation supports. We then describe the requirements that iterative adaptation places on the application's exported API and file format.

2.1 Adaptations

As with other adaptation techniques [7], [8], [11], [9], [18], iterative adaptation supports all data adaptations. The novelty of iterative adaptation comes into play, however, in its support of adaptations that gradually improve the content available on the mobile device. Such adaptation policies have, to the best of our knowledge, only been implemented by modifying the application. In iterative adaptation, however, they are implemented by using the exported APIs.

1. Fidelity versioning differs from traditional approaches to versioning as implemented by RTS [17] and other revision control systems in that different versions of a component represent transcoded views of the data as opposed to different stages in the lifetime of the component.

A very large number of adaptation policies are possible. We can, at best, give a sampling of the policies that can be implemented with iterative adaptation. We group adaptation policies into three classes. A first set of policies is based on repeated application of subsetting. One policy that works for most document-based applications is to load the text first, return control to the user immediately, and then load images and embedded elements in the background. Response time can be further improved by adding a data adaptation that compresses and decompresses the text. Policies can also be constructed that rearrange the order in which specific components, such as images, are loaded. For example, small images could be loaded first. Subsets other than text can also be chosen for initial loading before control is returned to the user. A useful policy for a presentation software loads the first slide, including text and images, returns control to the user, and then loads the remaining slides in the background. Other examples of subsetting policies that work for most document-based applications include choosing the subset based on the component type (e.g., fetch text and images, but leave out OLE embedded components) or the component size (e.g., fetch images smaller than 8 KB irrespective of their location in the document).

A second set of policies is based on repeated application of fidelity versioning. One example is a policy that uses Progressive JPEG compression to reduce latency. The adaptation policy first converts all images in the document into Progressive JPEG. A prefix of the resulting JPEG image file is loaded and control is returned to the user, producing an image with limited resolution. The remaining portions of the images are loaded in the background and inserted into the application using API calls, leading to progressively higher-resolution images. This set of policies can be combined with the first set in various ways.

A third and final example set of adaptation policies combines any of the previous policies by using certain events generated by the user to rearrange the order in which subsets or versions are loaded. For example, one policy first loads or refines the image over which the user moves the mouse.

Subsetting and fidelity versioning adaptations are effective for applications where users can do meaningful work with just a fraction of the original document or lower-fidelity versions of some of its components. The assumption is that by providing a limited (but still useful) version of the document, it is possible to significantly reduce the time users have to wait before they can interact with the document. Other components can then be fetched in the background or on demand. What constitutes a useful subset or fidelity version is application-specific. In constructing subsets, adaptation policies need to take into consideration the semantics of the document and ensure that, as the subset or version available to the user is iteratively extended, it remains meaningful in the context of the original document.

The previous list only provides a sample of the adaptation policies that can be implemented by iterative adaptation, but nonetheless attests to the power and the flexibility of the technique. In Section 5, we show that these policies provide significant reductions in user-perceived latencies for loading multimedia-rich documents over bandwidth-limited networks.

2.2 Requirements

Iterative adaptation is by nature restricted to applications with exported APIs. Moreover, for iterative adaptation to work, the application's API and file format must allow the adaptation system to discover, construct, and display component subsets of a document. We now examine these requirements in more detail.

First, the adaptation system needs to be able to discover the overall component structure of a document and the types of each component. This is commonly done by parsing the file(s) containing the input document. The requirement in this case translates into one that specifies that the document format needs to indicate component boundaries and component types. For example, in an electronic presentation, the boundary between different slides must be visible. Alternatively, if the application's file format is opaque and cannot be parsed, the document structure can still be extracted by running an instance of the application and calling on its API methods.

Second, the application's API must provide support for inserting component subsets into the application and for replacing a version of a component with a different higher-fidelity one.

Third, the application must be able to display component subsets or versions independently, without having other subsets or versions available. For example, the display of a slide should not be dependent on information about other slides or other information.

Additionally, to provide powerful adaptation policies, the adaptation system must be able to respond to events in the application (e.g., the user is moving the mouse over an image). Therefore, the application needs to export through its API an event registration and notification mechanism, by which the adaptation system can be notified of relevant events.

While the above requirements are certainly a limitation, we observe that many desirable candidate applications for adaptation, including the Microsoft Office Suite, Internet Explorer, Netscape Navigator, the KDE Office Suite, and Star Office already have file formats and exported APIs that largely meet these requirements. Recognizing the advantages of component-oriented software construction [19]—independent of adaptation—we foresee an increasing number of applications being developed with exported APIs.

3 Puppeteer

The Puppeteer system [16] implements iterative adaptation on Microsoft Windows and on Linux platforms.

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

The Puppeteer local proxy runs on the mobile client and is in charge of executing the policies that adapt the

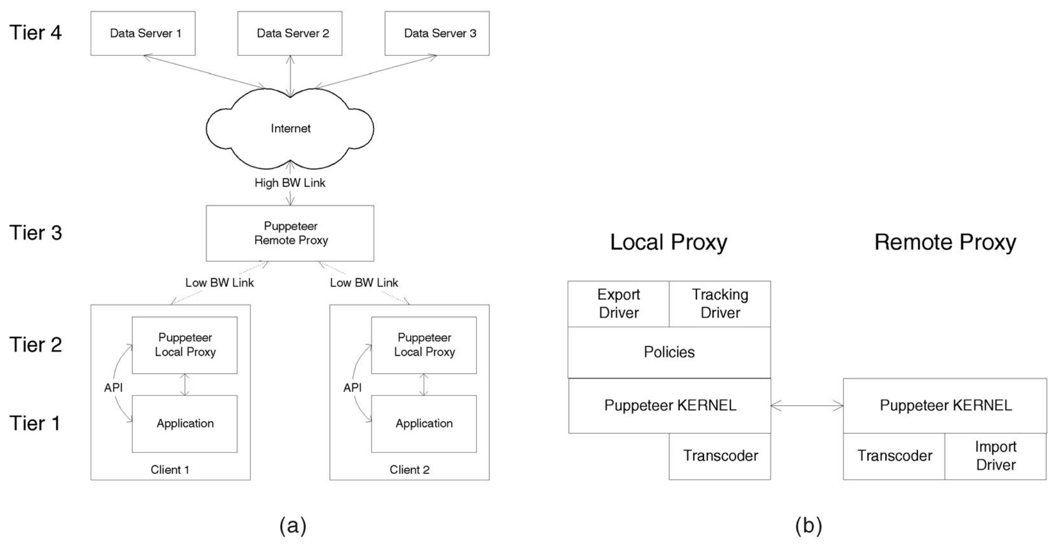


Fig. 1. Puppeteer architecture. (a) System architecture and (b) local and remote-proxy architectures.

applications by calling on their exported APIs. The Puppeteer remote proxy is responsible for parsing documents, exposing their structure, and transcoding components as requested by the local proxy. The Puppeteer remote proxy is assumed to have high-bandwidth connectivity (relative to the bandwidth-limited device) to the data servers. Data servers can be arbitrary repositories of data such as Web servers, file servers, or databases.

The realities of the mobile computing environment require Puppeteer to support different platforms, communication substrates, and applications with different document formats and APIs. Therefore, we design the Puppeteer proxies following a modular architecture, which minimizes the overhead in moving to a new platform or adding a new application or adaptation policy.

The Puppeteer local and remote proxies consist of four types of modules: Kernel, Drivers, Transcoders, and Policies (see Fig. 1b). The Puppeteer Kernel appears once in both the local and remote Puppeteer proxies. A driver supports adaptation for a particular component type. Transcoders implement specific transformations on component types. Drivers and transcoders may execute both in the local and the remote Puppeteer proxies. Policies specify particular adaptation strategies and execute in the local Puppeteer proxy.

3.1 Kernel

The Puppeteer Kernel is an application-independent module that runs in both the local and remote proxies and enables the transfer of document components. The Puppeteer 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 multilevel data structure, as components at any level can contain subcomponents. Skeleton nodes can have component-specific properties attached to them (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 Puppeteer Kernel first communicates the skeleton between the remote and the local proxy. It then enables adaptation policies to request a subset of the document's components and to specify transcoding filters to apply to specific components within the selected subset. To improve performance, the Puppeteer Kernel batches requests for multiple components into a single message and supports asynchronous requests.

3.2 Drivers

Puppeteer uses drivers to handle the lack of uniformity in document formats, exported APIs, and event handling mechanisms. Puppeteer requires an *import* and an *export* driver for every component type it adapts. To implement complex policies, a *tracking* driver is also necessary. Import drivers run on the remote proxy, while export and tracking drivers run on the local proxy. The import drivers parse 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 parses the file(s) directly to uncover the structure of the data. This results in good performance and enables local and remote proxies to run on different platforms (e.g., running the local proxy on Windows while running the Puppeteer remote proxy on Linux). When the application only exports an API, but has an opaque file format, Puppeteer runs an instance of the application on the remote proxy and uses the exported API 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 does not need to understand the application's file format. Unfortunately, running an instance of the application on the remote proxy creates significant overhead and requires both the local and remote proxies to run the environment of the application,

```
Document doc = PptrKernel.openDocument(strURI,MSPowerPoint.class)
MSPowerPoint comp = (MSPowerPoint)doc.getRootComponent();

comp.setSelected();
comp.children[0].setSelectedChildren();

PptrKernel.fetchComponents(doc);

MSPowerPointExportDriver expDriver = new MSPowerPointExportDriver();

expDriver.open(doc);
```

Fig. 2. Simple PowerPoint policy.

which, in most cases, amounts to running the same operating system on both the local and remote proxy. For these reasons, all the import drivers discussed in this paper uncover the document structure by directly parsing the document's files.

Parsing at the remote proxy does not work well for dynamic documents that choose what data to fetch and display by executing a script (e.g., JavaScript-enabled dynamic Web pages). Unlike static documents, these dynamic documents can choose to include different components at runtime, often in response to user input. Because of this, static parsing of the document is insufficient to learn every possible component that may be used. Instead, Puppeteer channels component requests made by applications displaying dynamic documents through a special import driver running on the local proxy. When the import driver detects a reference to a previously unknown component, it updates the document's skeleton appropriately.

Export drivers unparse the PIF and update the application using the application's exported API. A minimal export driver has to support inserting new components into the running application and replacing the contents of a component with a higher-fidelity version. Typical export drivers implement one of two update modalities that match the way most applications function. For applications that support a cut-and-paste mechanism (e.g., Microsoft Office), the driver uses the clipboard to insert new versions of the components. For applications that support reloading individual items they display (e.g., IE and Netscape), the driver instructs the application to reload the component (e.g., asking IE to refetch an image embedded in a HTML page). Tracking drivers are necessary for many complex policies. A tracking driver tracks which components are being viewed by the user and intercepts load requests. Tracking drivers can be implemented using polling (e.g., PowerPoint and Word) or event registration mechanisms (e.g., IE and Outlook).

3.3 Transcoders

Puppeteer's adaptations policies use transcoders to perform both subsetting and versioning transformations. These transcoders operate by either modifying the encoding of a component's data (e.g., compressing a bitmap into a low-fidelity JPEG image) or by changing the relationship between a component and its children (e.g., creating a new PowerPoint presentation with only a subset of the slides in the original document).

3.4 Policies

Policies run on the local proxy and control the fetching of components. Typical policies choose components and fidelities based on available bandwidth and user-specified preferences (e.g., fetch all text first). Other policies rely on tracking drivers to monitor the user (e.g., fetch the images in the PowerPoint slide that currently has the user's focus and prefetch the text of 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 the components it is fetching and focus the available bandwidth on fetching components that become visible to the user).

Policies choose what components to fetch by traversing the skeleton using a navigation interface that is similar to the interface provided by the W3C Document Object Model (DOM) [20] for XML documents. To select a component, a policy first gets a handle to it by navigating through the various layers of the document skeleton, and then sets the component's *select* property to true. A policy specifies how a component is to be transcoded by appending the Java class names of one or more transcoders (together with the proper transcoder parameters) to the component's transcoding queue.

After selecting components and setting transcoding transformations, the policy calls on the Puppeteer Kernel, which takes care of the actual data transfer and transcoding. When the content is available at the local proxy, the policy calls on the application's export driver to load the newly fetched components into the application.

Fig. 2 shows the code of a simple adaptation policy for PowerPoint that loads only the first slide of a presentation and all its embedded elements (e.g., images and OLE Objects). The policy first instructs the Puppeteer Kernel to open the presentation. It then selects the root element (the presentation) and the first slide with all its embedded items (its children). The policy then instructs the Puppeteer Kernel to fetch the components. Finally, the policy uses the PowerPoint export driver to load the document subset.

At present, Puppeteer supports only one active policy per document. Users can, however, change the active policy for a document at any time (see Section 3.5). In the future, we plan to extend Puppeteer to allow users to combine the effects of multiple policies by assigning multiple active policies to a document.

		PowerPoint	Word	Outlook	IE	Presentation	Writer
Import driver	Format	XML	XML	IMAP	(D)HTML	XML	XML
	Comp. types	PowerPoint Slide Image Sound OLE	Word Image Sound OLE	Email Image	HTML Image	Presentation Slide Image	Writer Image
Export driver		Cut & paste	Cut & paste	Reload	Reload	Reload	Reload
Tracking driver events		Open Select slide	Open Select img.	Read Inbox Open attach.	Browse Mouse on img.		

Fig. 3. Import, export, and tracking drivers for PowerPoint, Word, Outlook, Internet Explorer (IE), Presentation, and Writer. The table shows for each import driver the document format and component types that the driver recognizes. The table also shows the technique used by export drivers and the set of events trapped by tracking drivers.

3.5 User Interaction with Puppeteer

The applications' toolbars are extended with extra fields for selecting an adaptation policy that determines the fidelity level at which a document is opened or saved. Eventually, Puppeteer could rely on monitoring bandwidth or other resources to automatically choose a particular policy.

Puppeteer also provides a *Component Viewer* window that shows the current state of components in a document. Using this window, users can determine what components are currently loaded in the application and what components and what versions are in the process of being loaded. Users can also interact with the Component Viewer to control the fetching of component versions.

3.6 The Iterative Adaptation Process in Puppeteer

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 these components to the application.

When the user opens a (static) document, the Puppeteer Kernel on the Puppeteer remote 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 Puppeteer Kernel then transfers the document's skeleton to the Puppeteer local proxy. The policies running on the local proxy ask the Puppeteer Kernel to fetch an initial set of components at a specified fidelity and use the application's export driver to supply this set of components 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 local proxy now decide what further components or versions of components to fetch. They instruct the Puppeteer Kernel to do so, and then use the application's export driver to feed those newly fetched components to the application.

3.7 Implementation Summary

The Puppeteer system is written in Java. On the Windows platform, we use as applications Internet Explorer (IE), and PowerPoint, Word, and Outlook from Microsoft Office. On Linux, we use Presentation and Writer from the OpenOffice suite. For these applications, we implement the adaptation policies described in Section 2.1. Fig. 3 provides a summary of the implementation. Most of the entries in this table are self-explanatory. There are no tracking drivers for OpenOffice Presentation and Writer, as their APIs do not yet support event handling in the version used in this paper. This version also does not yet support embedded objects, explaining the absence of that entry under the component types supported for Presentation and Writer. All import drivers parse the document file to uncover the document's component structure, with the exception of Outlook, which uses the IMAP protocol to obtain the mailbox structure and its components from the mail server. We build the export and tracking drivers for our Microsoft Windows-based and Linux-based applications using COM and CORBA interfaces, respectively.

4 Implementation Experience

Our experience in implementing a variety of adaptation policies using iterative adaptation is very favorable. In the rest of this section, we first comment on the portability of Puppeteer. We then describe the functionality provided by the APIs of the applications we adapt. Finally, we reflect on the limitations we encounter in the APIs and file formats of these applications.

4.1 Portability

The modular Puppeteer architecture has proven successful in supporting a large number of adaptation policies for popular applications on different platforms. The Puppeteer Kernels used on the Windows and Linux platforms are identical. In terms of programming effort to implement new applications and new policies, Fig. 4 shows the code line counts for the various modules. The line counts for the Puppeteer Kernel module include the implementations of the Puppeteer protocol and support for text and progressive JPEG image compression. The relevant conclusion from this table is that the application-independent Puppeteer Kernel constitutes the bulk of the code. The amount of code specific to each application is much smaller. Similarly, the amount of code for a specific adaptation policy is small as well, on average requiring less than 250 lines, even including some of the more complicated adaptations. This is significant, as it shows that once the effort to develop new drivers for the application has been made, developing new policies is relatively easy. Moreover, it is possible to reuse adaptation policies between applications that operate on documents with similar component structures (e.g., PowerPoint and

Module		Code Lines
Puppeteer Kernel		9193
Policies	First Slide & Text	177
	Text & Small Images	159
	JPEG & Text Compression	334
	Total	670
Common to PowerPoint and Word	Import Driver	396
	Transcoder	88
	Total	484
PowerPoint	Drivers (Import, Export, Track)	761, 153, 146
	Transcoder	133
	Total	1193
Word	Drivers (Import, Export, Track)	250, 158, 129
	Total	537
Internet Explorer	Drivers (Import, Export, Track)	314, 175, 65
	Total	554
Outlook email	Drivers (Import, Export, Track)	787, 200, 74
	Total	1061
Common to Presentation and Writer	Import Driver	163
Presentation	Drivers (Import, Export)	374, 241
	Total	615
Writer	Drivers (Import, Export)	242, 197
	Total	439

Fig. 4. Code line counts for the Puppeteer kernel, policies, and drivers for PowerPoint, Word, Outlook e-mail, Internet Explorer, and OpenOffice Presentation and Writer.

Presentation). We also note that Presentation and Writer store their data natively in XML format and that PowerPoint and Word support XML formats as well. This allows us to use a common XML parsing package, supplied as part of Sun's Java library, reducing the size and complexity of our input drivers.

4.2 Existing API Functionality

The exported APIs of our applications are implemented based on the CORBA (Presentation and Writer) or COM (PowerPoint, Word, IE, and Outlook) standards. COM and CORBA are language-independent and support Java clients provided the Java applications have the necessary bindings.[2] To a Java application, the PowerPoint bindings appear to be regular Java objects. A call to a function of a binding object, however, results in the remote invocation of a function in the application being adapted.

We encounter a large degree of diversity in the functionality supported by the exported APIs of our applications. Typically, the exported APIs mimic the functionality available to a user interacting with the application through its GUI. For example, PowerPoint and Presentation provide APIs for opening presentations, navigating between slides, creating and reordering slides, and inserting embedded objects such as images. Some applications provide APIs with functionality beyond what is available to an interactive user. For example, IE provides full access to the DOM, allowing external programs to dynamically add elements to an HTML document or reload embedded objects, such as images.

There is an even greater variation in event notification mechanisms. For example, 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 to determine the slide currently being displayed. In contrast, IE has a rich event mechanism that allows third-party applications to register call-back functions for a wide range of events. The IE tracking driver uses this interface to detect when the user types a URL or moves the mouse over an image. We use the former event to instruct the Puppeteer 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).

4.3 Limitations

While we have been able to achieve a large number of powerful adaptations, these applications and their APIs and file formats are not designed with adaptation in mind. It then comes as no surprise that we encounter limitations in these APIs when we try to use them for the purpose of adaptation. Some limitations truly prevent us from performing adaptations that we want to implement. Others simply make it more difficult.

The most restrictive limitations include limits on the ability to recognize component boundaries and dependencies between components, which prevent independent loading. Both of these limitations occur in Word and Writer. The Word and Writer file formats store all the document's text in a single component. While pages in Word or Writer intuitively qualify as components, they are not reflected as individual components in the document format. Instead, pages are rendered dynamically when loading the document. As a result, adaptations like "load the first page of the document, return control to the user,

2. We synthesize the PowerPoint bindings from their COM IDL descriptions using off-the-self software [21].

and then load the rest of the document in the background" cannot be implemented for Word and Writer. Furthermore, Word and Writer evaluate cross-references and bibliographic citations when first loading the document. The eager evaluation of dependencies prevents us from independently loading paragraphs (which, as opposed to pages, do appear as recognizable components in the file format). Cross-references and bibliographic citations are not reevaluated when inserting a new paragraph and result in broken links.

Limitations in the exported APIs for updating the application make the implementation of some adaptations for some applications more cumbersome than needed. Ideally, we would like to load into the application any document component directly from its persistent representation. For example, we would like to create a new slide or update an image by loading its content from a file. Unfortunately, support for creating and updating components in this way is not available in most of our applications. For example, to add a slide to a PowerPoint presentation, we create a temporary document that contains the new slide and load it in the background. We then use the Java bindings to copy the slide from the temporary document and paste it in the active presentation. PowerPoint supports loading presentations in *invisible* mode, so the user is not aware of the existence of the temporary document. The use of cut and paste to update images, however, changes the contents of the clipboard and may therefore affect the normal operation of the application. Specifically, users may get an object different from what they expect if Puppeteer uses the clipboard to update the document between the times a user copies and pastes content. While the above example concentrates on PowerPoint, Presentation has similar limitations and requires a similar solution for adding slides into an open presentation.

The lack of an easy way to add new components or update existing components may require loading a larger or higher-fidelity set of initial components than otherwise required. For example, in PowerPoint, there is no easy way to update the *master slides*, which store properties that are common to all slides in the presentation (e.g., logos, background color, or font size). For this reason, all our PowerPoint policies, even those that load only the first slide or just the text components of the presentation, load all the elements (images, embedded OLE objects) of the master slides. Loading the master slides in all cases simplifies the implementation (it does not have to deal with all the properties of the master slides) at the expense of the extra latency incurred to transfer the data for the master slides.

While these limitations are real, we have nonetheless been able to implement a large number of adaptations for the different applications without much complexity and with good performance as shown in the next section.

5 Experimental Evaluation

In this section, we present experimental results for loading multimedia-rich documents using three configurations: native, Puppeteer, and in-application. *Native* uses the unmodified application without any adaptive support. This configuration represents the normal operating mode of the applications we use, where the application opens a direct link to the document repository (e.g., file server, IMAP, WWW) to download content. *Puppeteer* uses the Puppeteer system to add adaptive support to the applications. The applications are unmodified, but document data flows from the document repository to the application through the Puppeteer proxies. Finally, *in-application* uses applications that we modify to add support for adaptation within the application. The adaptive versions of the applications consist of two parts: a proxy that can service individual document components and perform transcoding transformations, and the modified application which interacts with the user and requests components from the proxy. Document data flows from the document repository to the proxy and from there to the application running on the mobile device.

We perform our experiments on a platform that consist of three Pentium III 500 MHz PCs running either Windows 2000 or Redhat Linux 7.0. One PC is configured as a data server running Apache 1.3 or Cyrus IMAP 1.6.24. This PC stores all the documents and emails we use in our experiments. A second PC plays the role of the mobile wireless client and runs the user's applications. For our experiments with Puppeteer, this PC also runs the Puppeteer local proxy. For our experiments with Puppeteer and in-application adaptation, we use a third PC that runs either the Puppeteer remote proxy or the proxy that supports the adaptive application.

To control the bandwidth between the PC playing the role of the mobile wireless client and the rest of the testbed, we use an extra PC running the DummyNet network simulator [22]. The placement of the PC running DummyNet depends on the specific configuration. For the *native*, *Puppeteer*, and *in-application* configurations, the PC running DummyNet is placed between, on one hand, the PC playing the role of the mobile wireless client and, on the other hand, the data server, the PC running the Puppeteer remote proxy, and the PC running the proxy that supports the adaptive application, respectively. The Puppeteer remote proxy and the proxy that supports the adaptive application communicate with the data server over a high-speed LAN.

We report results for three different bandwidths: one at which the application is network-bound, one at which it is CPU-bound, and one in-between. Although one would expect to use adaptation only at low bandwidths, the higher-bandwidth results are included for completeness.

The data sets we use for PowerPoint, Word, Presentation, and Writer consist of Microsoft Office documents downloaded from the Web through October 1999 and characterized in de Lara et al. [6]. We experiment with PowerPoint and Word documents ranging in size from 52 KB to 21 MB and from 280 KB to 2.4 MB, respectively. For Presentation and Writer, we convert the documents from their original PowerPoint and Word formats to the OpenOffice formats. IE loads HTML documents downloaded from the Web through March 2000 by reexecuting the Web traces collected by Cunha et al. [23]. We present results for HTML documents ranging in size from just over 4 KB to 756 KB. Finally, the Outlook emailer loads synthetic emails with

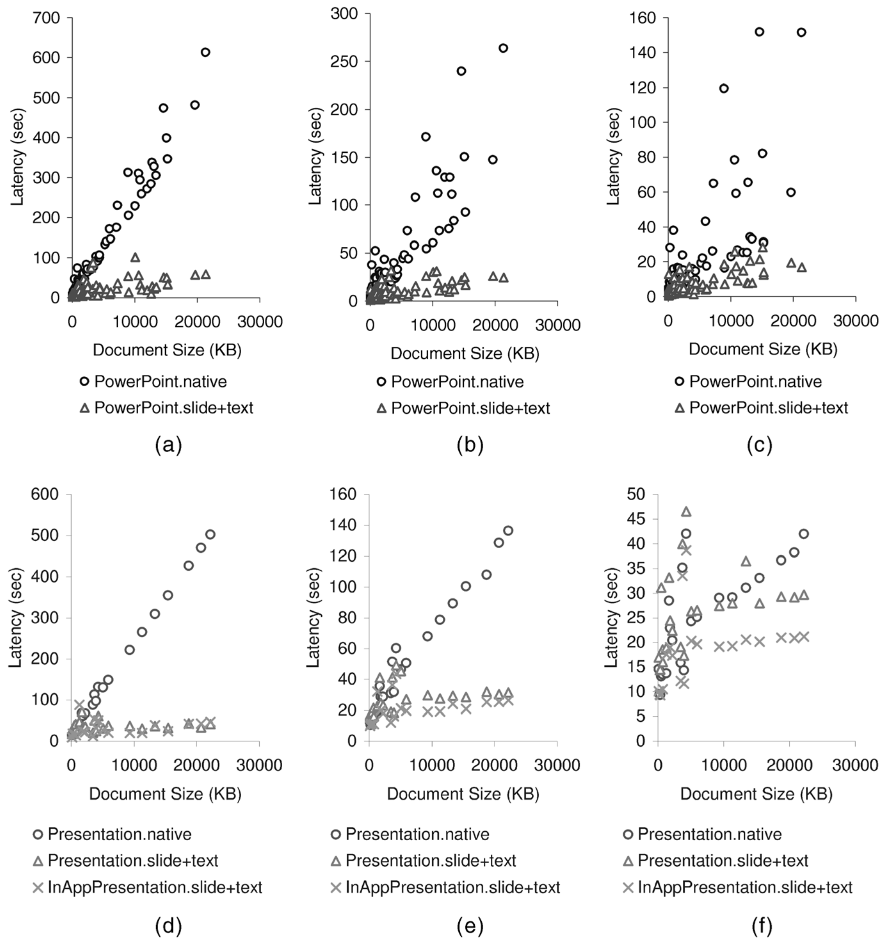


Fig. 5. **Fetch First Slide and Text**. Latency for loading documents with PowerPoint and Presentation over 384 Kb/sec, 1.6 Mb/sec, and 10 Mb/sec. Shown are latencies for native PowerPoint (*PowerPoint.native*) and native Presentation (*Presentation.native*), Puppeteer runs for loading just the components of the first slide and the text of the remaining slides (*PowerPoint.slide+text, Presentation.slide+text*), and runs of a modified version of Presentation which implements adaptation within the application and loads just the components of the first slide and the text of the remaining slides (*InAppPresentation.slide+text*). (a) PowerPoint 384 Kb/sec, (b) PowerPoint 1.6 Mb/sec, (c) PowerPoint 10 Mb/sec, (d) Presentation at 384 Kb/sec, (e) Presentation 1.6 Mb/sec, and (f) Presentation 10 Mb/sec.

image attachments. The emails ranged in size from 7 KB to 2.9 MB.

In the rest of this section, we first measure the effectiveness of the Puppeteer adaptation system for several sample adaptation policies. We then compare Puppeteer's performance to the in-application approach that implements adaptation within the application. Finally, we quantify the Puppeteer overhead.

5.1 Some Adaptation Policies

This section illustrates the performance of some sample adaptation policies implemented in Puppeteer. Figs. 5, 6, and 7 show the latencies for these sample policies as a function of the size of the documents. We compute latency as the time interval between the time of the initial request to load a document and the time when some version of the document is rendered by the application and control is returned to the user. All figures show a common trend. For low bandwidths, the network is the bottleneck, and the benefits of adaptation are most significant. The latencies are largely dependent on the size of the data transferred. They grow more or less linearly as document size gets larger, and the latency data points lie in a straight line. For higher bandwidths, the data points become more dispersed. The experiments become CPU-bound, and the latency is governed by the time it takes the application to parse and render the document, which depends on the document's size, as well as its structure (number of images, embedded objects, pages, etc.).

5.1.1 PowerPoint and Presentation: Fetch First Slide and Text

In this experiment, we measure the latency for loading PowerPoint and Presentation documents with an adaptation policy that returns control to the user once the application loads all the components of the first slide, but just the text component of all remaining slides. Afterward, the other components of the remaining slides are loaded in the background. With these adaptations, user-perceived latency is much reduced compared to loading the entire document before returning control to the user.

Fig. 5 shows the results of these experiments for 384 Kb/sec, 1.6 Mb/sec, and 10 Mb/sec network links. For each document, the figures contain two vertically aligned points representing the latency in two system configurations: native

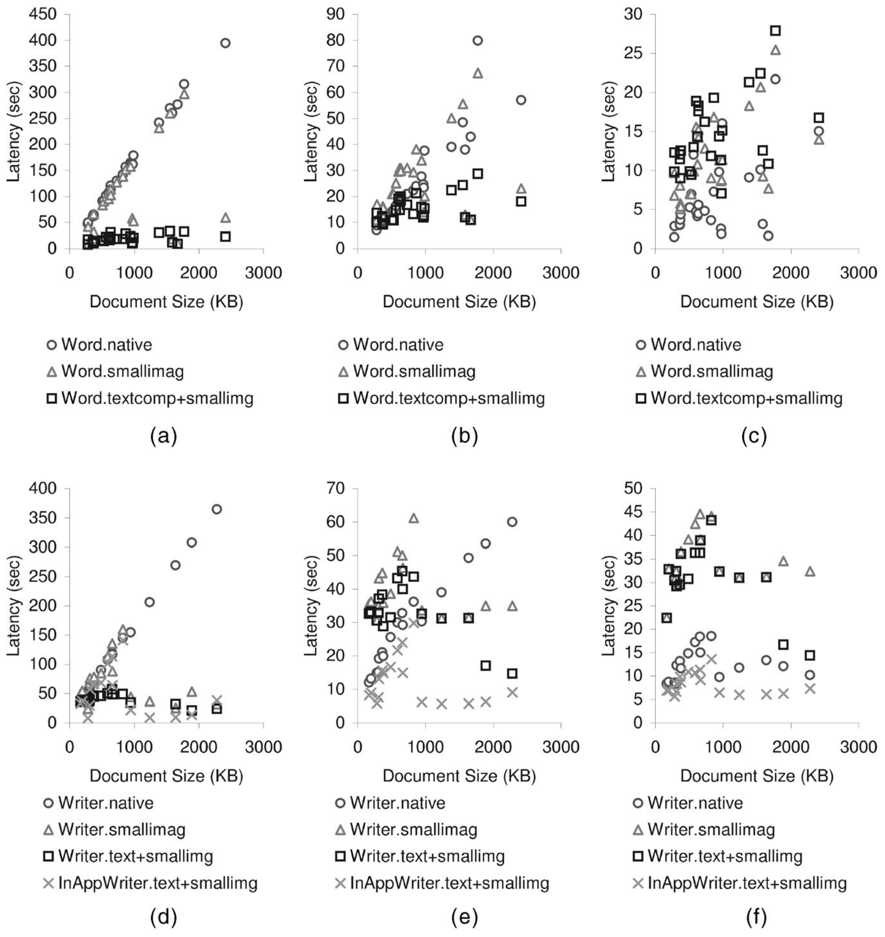


Fig. 6. **Text and Small Images**. Latency for loading documents with Word and Writer over 56 Kb/sec, 384 Kb/sec, and 10 Mb/sec. Shown are latencies for native Word and Writer (*Word.native, Writer.native*), Puppeteer runs that load text and images smaller than 4 KB (*Word.smallimag, Writer.smallimag*) and load compressed text and images smaller than 4 KB (*Word.text+smallimag, Writer.text+smallimag*), and runs of a modified version of Writer which implements adaptation within the application, and loads compressed text and images smaller than 4 KB (*InAppWriter.text+smallimg*). (a) Word 56 Kb/sec, (b) Word 384 Kb/sec, (c) Word 10 Mb/sec, (d) Writer 56 Kb/sec, (e) Writer 384 Kb/sec, and (f) Writer 10 Mb/sec.

PowerPoint (*PowerPoint.native*) or native Presentation (*Presentation.native*), and Puppeteer runs for PowerPoint and Presentation for loading all the components of the first slide and the text of all remaining slides (*PowerPoint.slide+text, Presentation.slide+text*). In addition to the data for native Presentation and Presentation with Puppeteer, Figs. 5d, 5e, and 5f also plot latencies for loading the documents with a modified version of Presentation (*InAppPresentation.slide+-text*), which supports adaptation within the application. We defer the discussion of these results to Section 5.2.

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

On average, for documents larger than 1 MB, *PowerPoint.slide+text* achieves latency reductions of 75 percent, 71 percent, and 54 percent on 384 Kb/sec, 1.6 Mb/sec, and 10 Mb/sec networks, respectively. *Presentation.slide+text* achieves latency reductions of 61 percent, 36 percent, and 13 percent, respectively, for the same documents on the same networks. These results show that, for large documents, it is possible to return control to the user after loading just a small fraction of the document's total data (about 10.9 percent for documents larger than 4 MB).

Latency reductions for Presentation are lower because rendering is done more efficiently in Presentation than in PowerPoint. The Puppeteer overhead for adapting the same document is similar for both applications. Presentation's lower rendering time, however, effectively increases the relative contribution of the Puppeteer overhead to overall latency. This, in turn, results in lower relative latency reductions. We discuss Puppeteer's overhead in more detail in Section 5.3.

5.1.2 Word and Writer: Text and Small Images

In this experiment, we reduce user-perceived download latency for Word and Writer documents by loading only the text of the documents and images smaller than 4 KB before returning control to the user. We also explore the use of text compression to further reduce download latency. Fig. 6 shows the latency for loading the Word and Writer documents over 56 Kb/sec, 384 Kb/sec, and 10 Mb/sec

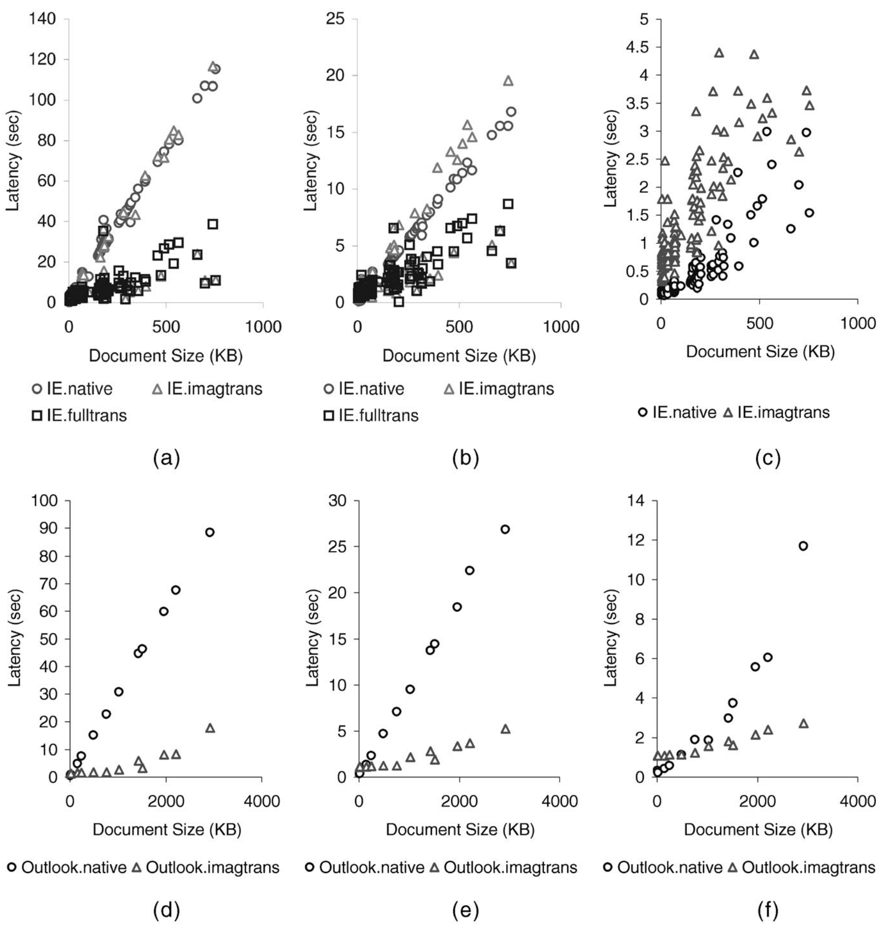


Fig. 7. **JPEG and Text Compression**. Latency for loading document with IE over 56 Kb/sec, 384 Kb/sec, and 10 Mb/sec, and emails with Outlook over 384 Kb/sec, 1.6 Mb/sec, and 10 Mb/sec. Shown are latencies for native IE and Outlook (*IE.native, Outlook.native*), and Puppeteer runs that load transcoded images (*IE.imagtrans, Outlook.imagtrans*). For IE, the figures also show the latency for Puppeteer runs that load transcoded images and gzip-compressed text (*IE.fulltrans*). (a) IE 56 Kb/sec, (b) IE 384 Kb/sec, (c) IE 10 Mb/sec, (d) Outlook 384 Kb/sec, (e) Outlook 1.6 Mb/sec, and (f) Outlook 10 Mb/sec.

networks. The figures show latencies for native Word and Writer (*Word.native, Writer.native*) and for Puppeteer runs that load only the text and images smaller than 4 KB (*Word.smallimag, Writer.smallimag*), and load gzip-compressed text and images smaller than 4 KB (*Word.textcomp+smallimag, Writer.textcomp+smallimag*). In addition to the data for native Writer and Writer with Puppeteer, Figs. 6d, 6e, and 6f also plot latencies for loading the documents with a modified version of Writer (*InApp-Writer.text+smallimg*), which supports adaptation within the application. We differ the discussion of these results to Section 5.2.

Word.smallimag and *Writer.smallimag* show how loading images smaller than 4 KB reduces latency for about half of the documents. For these documents, *smallimag* achieves an average reduction in latency of 55 percent for Word and 57 percent for Writer, over 56 Kb/sec. *Word.textcomp+smallimag* and *Writer.textcomp+smallimag* further reduce latency for all documents. For all documents, *textcomp+smallimag* achieves on average a reduction in latency of 85 percent for Word and 59 percent for Writer, over 56 Kb/sec. For documents larger than 1 MB, it achieves a reduction of 61 percent for Word and 50 percent for Writer over 384 Kb/sec. The *textcomp+smallimag* latency reductions for Writer are smaller than for Word. As was the case for Presentation and PowerPoint, Writer is more efficient at parsing and rendering documents than Word and, therefore, the relative contribution of Puppeteer's overhead to the overall latency is higher for Writer than for Word. We discuss Puppeteer's overhead in more detail in Section 5.3. Finally, Figs. 6c and 6f show that, for most documents on 10 Mb/sec networks, text compression is detrimental to performance.

5.1.3 Outlook and IE: JPEG and Text Compression

In this experiment, we explore the use of progressive JPEG compression to reduce the user-perceived latency for HTML pages and emails. Our goal is to reduce the time required to display a page or load an email by lowering the fidelity of some of the document's images.

Our prototype converts, at runtime, GIF and JPEG images embedded in the HTML documents or e-mails into a progressive JPEG format using the PBMPlus [24] and Independent JPEG Group [25] libraries. We then transfer only the first $\frac{1}{7}\text{th}$ 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 IE or Outlook as though 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 to run the transcoder.

Fig. 7 shows the latency for loading the HTML documents over 56 Kb/sec, 384 Kb/sec, and 10 Mb/sec networks, and for loading the e-mails over 384 Kb/sec, 1.6 Mb/sec, and 10 Mb/sec networks. The figures show latencies for native IE and Outlook (*IE.native, Outlook.native*) and for Puppeteer runs that load transcoded images (*IE.imagtrans, Outlook.imagtrans*). For IE, the figures also show the latency for Puppeteer runs that load transcoded images and gzip-compressed text (*IE.fulltrans*). We do not show Outlook runs with gzip-compressed text, since the text components of our emails are small (under 8 KB) and the results are similar to *Outlook.imagtrans*.

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 percent and 35 percent for 56 Kb/sec and 384 Kb/sec, respectively. A closer examination reveals that roughly two thirds of the documents see some latency reduction. The remaining third 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 adaptation. The *IE.fulltrans* run shows that, with image and text transcoding, Puppeteer achieves average reductions in latency, for all documents larger than 128 KB, of 76 percent and 50 percent, for 56 Kb/sec and 384 Kb/sec, respectively.

Outlook.imagtrans shows that on 10 Mb/sec networks, transcoding is useful only for emails with image attachments larger than 512 KB. The latency savings at 10 Mb/sec result from reducing the parsing and rendering time of the attachments. *Outlook.imagtrans* achieves an average reduction in latency, for documents larger than 128 KB, of 85 percent and 71 percent, for 384 Kb/sec and 1.6 Mb/sec, respectively.

Overall transcoding time takes between 11.5 percent to less than 1 percent of execution time. Moreover, since Puppeteer overlaps image transcoding with data transmission, the overall effect on execution time diminishes as network speed decreases.

5.2 Comparison with In-Application Adaptation

In this section, we compare iterative adaptation to in-application adaptation, an approach where applications have native support for adaptation. Our comparison focuses on the performance of these approaches for loading remote content over bandwidth-limited links and the ease with which new adaptation policies can be implemented.

5.2.1 Performance

Extending an application to support in-application adaptation requires access to its source code. This requirement prevents us from adding native adaptation support to our Windows-based applications. Fortunately, Presentation and Writer are part of the OpenOffice suite, an open-source initiative, which enables us to modify them to add native support for adaptation.

We modify Presentation and Writer to implement the adaptive policies described in Section 2.1. In the rest of this discussion, we refer to the modified applications, which have native support for bandwidth adaptation, as Adaptive Presentation and Adaptive Writer.

Figs. 5d, 5e, and 5f plot the latencies for loading presentations over 384 Kb/sec, 1.6 Mb/sec, and 10 Mb/sec network links with native Presentation *without* any adaptation support (*Presentation.native*), Presentation with Puppeteer support (*Presentation.slide+text*), and Adaptive Presentation (*InApplication.slide+text*). The Puppeteer and Adaptive Presentation runs implement an adaptation policy that returns control to the user after it loads the components of the first slide and the text component of all remaining slides. The components of the remaining slides are loaded afterward in the background.

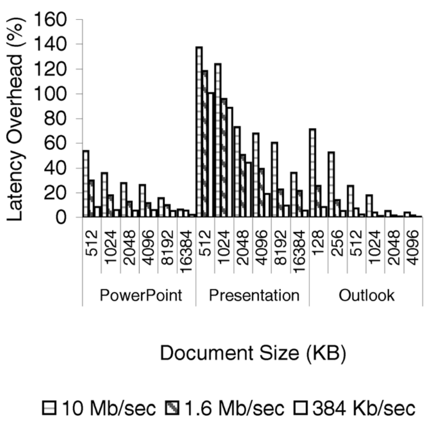
The results for native Presentation and Presentation with Puppeteer support are discussed in Section 5.1.1. We now focus on the results for Adaptive Presentation. Our first observation is that Adaptive Presentation always outperforms Presentation with Puppeteer support. This result is expected as Adaptive Presentation incurs less extra processing overhead. While in Puppeteer every document is parsed twice, first by the Puppeteer system and then by the application being adapted, Adaptive Presentation only has to parse the document once. The performance gap between Adaptive Presentation and Puppeteer is, however, small, averaging 4 percent, 13 percent, and 28 percent, for presentations larger than 2 MB, over 384 Kb/sec, 1.6 Mb/sec, and 10 Mb/sec, respectively. Moreover, the performance gap narrows rapidly as document size grows and network speed goes down. The results for Adaptive Writer and Writer with Puppeteer support shown in Figs. 6d, 6e, and 6f are similar to those of Adaptive Presentation.

5.2.2 Programming Cost

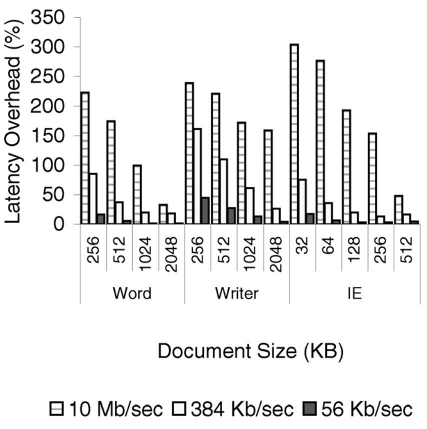
While the source code for OpenOffice Presentation and Writer is freely available on the Web, modifying these applications to add native support for adaptation is a challenging undertaking. OpenOffice sources consist of more than 20,000 files with over 8 million lines of code, for a code base of roughly 200 MB.

Independent of the approach to adaptation, the cost of adding adaptive support for an application can be split into two components: a base cost that includes the required infrastructure for making the application adaptable and an incremental cost incurred for each adaptation policy that is implemented. For in-application adaptation, the base adaptation cost to support Presentation and Writer consists of the programming cost of providing a transcoding proxy. Fortunately, the OpenOffice distribution already includes a proxy server that can service individual document components to OpenOffice applications over the network. This limits our programming effort to extending the proxy with transcoding support. The incremental cost consists of adaptation policies that modify the control flow of the application, changing the order in which it loads components or their fidelity. In our experience, adding a new adaptation policy to Presentation or Writer typically requires close to 1,000 lines of code.

In the case of Puppeteer, the base adaptation cost consists of the programming effort required to develop drivers for the application, which handle the lack of uniformity in document formats, APIs, and event handling mechanisms. Once these drivers are available, however, creating a new adaptation policy is straightforward. On



(a)



(b)

Fig. 8. Initial adaptation costs. (a) and (b) show latency overhead for loading documents and emails with Puppeteer for PowerPoint, Presentation, Outlook, Word, Writer and IE.

average, our adaptation policies require less than 250 lines of code. Moreover, we are able to reuse adaptation policies between applications that operate on documents with similar component structures (e.g., PowerPoint and Presentation).

The difference in the programming effort for implementing adaptation policies in iterative adaptation and in-application adaptation derives from the level of abstraction at which adaptation policies are defined. Adaptation policies in Puppeteer are written at a high level of abstraction, as the exported APIs that Puppeteer uses for adaptation tend to encompass significant functionality in a single function call. The API shields the Puppeteer policy developer from much of the complexity involved in implementing the actual functionality supported by the interface. In contrast, in-application adaptation policies are written using lower-level primitives, requiring more work from the policy developer.

Moreover, because exported APIs are meant to be used by a large set of users, they tend to be better documented and change at a much lower rate than the internals of an application. For example, COM interfaces once published are considered immutable.[3]

Conversely, because in-application adaptation has full access to the application's source code, it is not limited by the specific functionality that the application chooses to export through its API. Therefore, a wider set of adaptation policies can be implemented (albeit at a higher programming cost) with in-application adaptation than with iterative adaptation.

5.2.3 Summary

The results presented in this section confirm the intuition that an in-application approach to adaptation can achieve larger latency reductions than a centralized approach to application adaptation, such as iterative adaptation. The experimental results show, however, that the performance gap is small and narrows rapidly as document size grows and network speed goes down. In contrast, the programming effort required to add a new adaptation policy is significantly larger for in-application adaptation than for iterative adaptation.

3. In COM, a vendor needs to create a brand new interface in order to add new calls to an exported API. This constraint ensures that older clients retain forward compatibility, while newer clients can take advantage of the new functionality.

5.3 Puppeteer Overhead

The Puppeteer 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 exported API commands used to control the application.

5.3.1 Initial Adaptation Costs

To determine the one-time initial costs, we compare the latency of loading PowerPoint, Word, HTML, e-mail, Presentation, and Writer documents in their entirety using the native application and the application with Puppeteer support. This policy represents the worst possible case; it incurs the overhead of parsing the document to obtain the PIF and it does not benefit from any adaptation (i.e., it loads all components at their highest fidelity).

Figs. 8a and 8b show the latency overhead, and Fig. 9 shows the data overhead for loading entire documents with PowerPoint, Word, IE, Outlook, Presentation, and Writer. We maximize the overhead by using a policy that requests components individually. This policy does not benefit from the batching of control messages and, instead, incurs a separate control message for every loaded component. Latency and data overheads are normalized by the latencies and data traffic for loading the documents with the native applications. Overall, for all these applications, the Puppeteer latency overhead becomes less significant as document size increases and network speed decreases. Moreover, for large documents transmitted over medium to slow-speed networks, where adaptation would normally be used, the Puppeteer latency overhead is small compared to the total document loading time. For example, the overhead for PowerPoint and IE for large documents is just 2 percent over 384 Kb/sec and 4.7 percent over 56 Kb/sec, respectively.

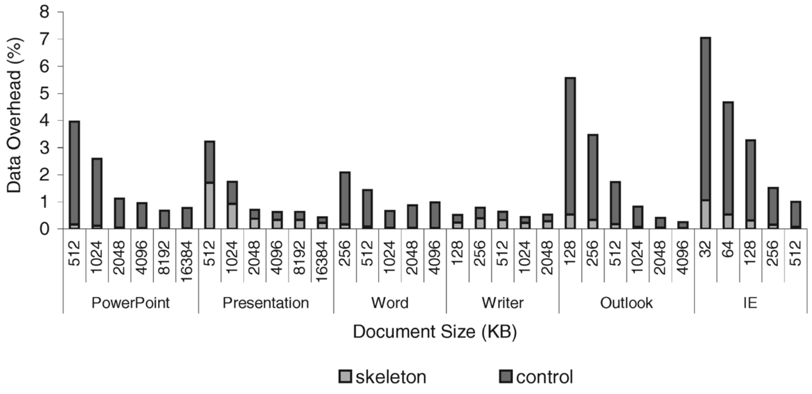


Fig. 9. Data overhead for loading documents and emails with Puppeteer for PowerPoint, Presentation, Word, Writer, Outlook, and IE.

Puppeteer's overhead is larger for Presentation and Writer than for PowerPoint and Word because Presentation and Writer are more efficient parsing and rendering documents than PowerPoint and Word. While the Puppeteer execution time is virtually identical for the two sets of applications, the lower parsing and rendering time increases Puppeteer's relative contribution to overall latency.

Fig. 9 decomposes the data overhead into data transmitted to fetch the skeleton (*skeleton*) and data transmitted to request components (*control*). This data confirms the results of Figs. 8a and 8b. The Puppeteer data overhead becomes less significant as document size increases. For example, the data overhead for PowerPoint and HTML documents is as little as 2.9 percent and 1.3 percent, respectively.

5.3.2 Continuing Adaptation Costs

The continuing costs of adaptation using the exported APIs are clearly dependent on the application and the adaptation policy. Our purpose is not to give a comprehensive analysis of exported API-related adaptation costs, but to show that they are small compared to the network transfer times.

We perform two experiments: loading and pasting new 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.

For PowerPoint, we find that the average time to load a single slide in a presentation is 894 milliseconds with a standard deviation of 819 milliseconds. For each additional slide inserted in the application with the same API call, the average time is 539 milliseconds with a standard deviation of 591 milliseconds. In comparison, the average network time to load a slide over the 384 Kb/sec network is 2,994 milliseconds, with a standard deviation of 3,943 milliseconds.

For IE, the average time to load an image in a page is 33 milliseconds with a standard deviation of 19 milliseconds. Loading additional images as part of the same update takes an average of 33 milliseconds per image with a standard deviation of 12 milliseconds. These image update times are small compared to the average network time. For our data set, the average time to load an image over a 56 Kb/sec network is 565 milliseconds with a standard deviation of 635 milliseconds.

The above results suggest that the cost of using exported API calls for adaptation is small (e.g., for IE, the API overhead of loading an image is 5.8 percent) and that most of the time that it takes to add or upgrade a component is spent transferring the data over the network.

6 Related Work

There are many possible implementations of bandwidth adaptation. Based on where the adaptation is implemented, we recognize a spectrum of possibilities with two extremes: system-based [9] [11] and in-application adaptation [7], [8], [18]. With system-based adaptation, the system performs all adaptation by interposing itself between the application and the data. No changes are needed in the application. System-based adaptation also provides centralized control, allowing the system to adapt several applications according to a system-wide policy. With in-application adaptation, the application is changed to add the required adaptive behavior. System-based adaptation is limited to data adaptation, while in-application adaptation allows both data and control adaptation. Iterative adaptation attempts to bring together the benefits of system-based and in-application adaptation, namely, to implement control adaptations without modifying the applications and to retain centralized control over adaptation.

Another approach that tries to strike a middle ground between system-based and in-application adaptation is application-aware adaptation [10], [26]. 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. Iterative adaptation has similarities to application-aware adaptation. Both approaches delegate common adaptation tasks to the system, which provides a centralized locus of control for adaptation of multiple applications. The approaches differ, however, in how control adaptation policies are implemented. In iterative adaptation, the applications export the interfaces and the system invokes 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. Iterative adaptation is a more flexible approach to adaptation than application-aware adaptation. Whereas application-aware adaptation requires the application designer to foresee all necessary adaptations at the time the application is written, iterative adaptation enables third parties to add new adaptation policies after the application has been released.

Visual Proxies [18], an offspring of Odyssey [26], implements application-specific adaptation policies without modifying the application by using interposition between the window system and the application. This technique is limited to window system commands and does not use exported application APIs. While it enables some of the same adaptations, it requires much more complicated application constructs.

Several groups [27], [28], [29] have suggested the need for centralized adaptation systems that implement system-wide adaptation policies. In these approaches, the adaptation system monitors system resources and user behavior and coordinates the adaptation of the applications running on the client by issuing commands that force the applications to adapt. Although similar in nature to Puppeteer, these efforts have been mainly focused on streaming media and limit the adaptation system to switching the application between a few predefined operation modes.

Iterative adaptation differs from approaches that use exported APIs to change the application's algorithms by dynamically reconfiguring the application's component structure [30], [31], [32], [33], [19]. While dynamic component-based application reconfiguration is a technique that is arguably more powerful than iterative adaptation, at the time of writing, we are not aware of any widely-deployed commercial application that supports dynamic component reconfiguration.

7 Conclusions

This paper presents the concept of *Iterative Adaptation*. Underlying iterative adaptation is the idea that the adaptation system controls applications by calling methods in the APIs that the applications export. This approach has a number of advantages: it allows a wide variety of adaptation policies to be implemented with popular office productivity applications, it achieves significant latency reductions over low-bandwidth links, and it does not require any modifications to the applications.

We have implemented the concept of iterative adaptation in the Puppeteer system. We have used Puppeteer to evaluate the extent to which existing APIs can be used for the purposes of adapting document-based applications for bandwidth-limited devices. In particular, we have implemented a number of adaptation policies for popular applications from the Microsoft Office and the OpenOffice productivity suites and for Internet Explorer. Although we have found some limitations in their APIs, we have been able to implement a large number of adaptations policies without much complexity and with little overhead. Moreover, Puppeteer achieves performance similar to in-application adaptation, an approach that implements adaptation by modifying the application, while requiring just a fraction of the coding effort.

Puppeteer's modular architecture is specifically designed to limit the amount of development for integrating new applications and new adaptations. Overall, we have found that the bulk of the code is platform and application-independent. Due to the lack of uniform document formats, APIs, and event handling mechanisms, some amount of development remains necessary to support a new application. Once the application-specific code for Puppeteer is implemented, however, writing new adaptation policies proves much easier.

In future work, we plan to explore requirements for standard interfaces and file formats that would make applications more amenable to adaptation and limit the programming effort that goes into supporting new component types. We are also pursuing related research in specifying and enforcing complex adaptation policies that provide fair use of system resources across multiple applications.

References

[1] R. Bagrodia, W.W. Chu, L. Kleinrock, and G. Popek, "Vision, Issues, and Architecture for Nomadic Computing," *IEEE Personal Comm.*, vol. 2, no. 6, pp. 14-27, Dec. 1995.

[2] D. Duchamp, "Issues in Wireless Mobile Computing," *Proc. Third Workshop Workstation Operating Systems*, pp. 1-7, Apr. 1992.

[3] G.H. Forman and J. Zahorjan, "The Challenges of Mobile Computing," *Computer*, pp. 38-47, Apr. 1994.

[4] R.H. Katz, "Adaptation and Mobility in Wireless Information Systems," *IEEE Personal Comm.*, vol. 1, no. 1, pp. 6-17, 1994.

[5] M. Satyanarayanan, "Fundamental Challenges in Mobile Computing," *Proc. 15th ACM Symp. Principles of Distributed Computing*, May 1996.

[6] E. de Lara, D.S. Wallach, and W. Zwaenepoel, "Opportunities for Bandwidth Adaptation in Microsoft Office Documents," *Proc. Fourth USENIX Windows Symp.*, Aug. 2000.

[7] 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 Comm.*, vol. 5, no. 4, pp. 10-19, Aug. 1998.

[8] A.D. Joseph, J.A. Tauber, and M.F. Kaashoek, "Building Reliable Mobile-Aware Applications Using the Rover Toolkit," *Proc. Second ACM Int'l Conf. Mobile Computing and Networking (MobiCom '96)*, Nov. 1996.

[9] L. Mummert, M. Ebling, and M. Satyanarayanan, "Exploiting Weak Connectivity for Mobile File Access," *Proc. 15th ACM Symp. Operating Systems Principles*, Dec. 1995.

[10] D. Andersen, D. Basal, D. Curtis, S. Srinivasan, and H. Balakrishnan, "System Support for Bandwidth Management and Content Adaptation in Internet Applications," *Proc. Fourth Symp. Operating Systems Design and Implementation*, Oct. 2000.

[11] J.J. Kistler and M. Satyanarayanan, "Disconnected Operation in the CODA File System," *ACM Trans. Computer Systems*, vol. 10, no. 1, pp. 3-25, Feb. 1992.

[12] E. de Lara, D.S. Wallach, and W. Zwaenepoel, "HATS: Hierarchical Adaptive Transmission Scheduling for Multi-Application Adaptation," *Proc. 2002 Multimedia Computing and Networking Conf. (MMCN '02)*, Jan. 2002.

[13] *Microsoft Office 97/Visual Basic Programmer's Guide.* Microsoft Press, 1997.

[14] S. Roberts, *Programming Microsoft Internet Explorer 5.* Microsoft Press, 1999.

[15] J. Flinn, E. de Lara, M. Satyanarayanan, D.S. Wallach, and W. Zwaenepoel, "Reducing the Energy Usage of Office Applications," *Proc. IFIP/ACM Int'l Conf. Distributed Systems Platforms (Middleware)*, Nov. 2001.

[16] E. de Lara, D.S. Wallach, and W. Zwaenepoel, "Puppeteer: Component-Based Adaptation for Mobile Computing," *Proc. Third USENIX Symp. Internet Technologies and Systems*, Mar. 2001.

[17] W.F. Tichy, "RCS—A System for Version Control," *Software—Practice and Experience*, vol. 15, no. 7, pp. 637-654, 1985.

[18] M. Satyanarayanan, J. Flinn, and K.R. Walker, "Visual Proxy: Exploiting OS Customizations without Application Source Code," *Operating Systems Rev.*, vol. 33, no. 3, pp. 14-18, July 1999.

[19] C. Szyperski, *Component Software Beyond Object-Oriented Programming.* Addison Wesley, 1998.

[20] L. Wood, "Document Object Model (DOM) Level 1 Specification," http://www.w3.org/TR/REC-DOM-Level-1/, Oct. 1998.

[21] "Bridge2java," IBM AlphaWorks, http://www.alphaworks.ibm.com/tech/bridge2javam, 2005.

[22] L. Rizzo, "DummyNet: A Simple Approach to the Evaluation of Network Protocols," *ACM Computer Comm. Rev.,* vol. 27, no. 1, pp. 13-41, Jan. 1997.

[23] C.R. Cunha, A. Bestavros, and M.E. Crovella, "Characteristics of WWW Client-Based Traces," Technical Report TR-95-010, Boston Univ., Apr. 1995.

[24] J. Poskanzer, "PBMPLUS," http://www.acme.com/software/pbmplus, 2005.

[25] Independent JPEG Group, http://www.ijg.org/, 2005.

[26] B.D. Noble, M. Satyanarayanan, D. Narayanan, J.E. Tilton, J. Flinn, and K.R. Walker, "Agile Application-Aware Adaptation for Mobility," *Operating Systems Rev. (ACM),* vol. 51, no. 5, pp. 276-287, Dec. 1997.

[27] C. Efstratio, K. Cheverst, N. Davies, and A. Friday, "Architectural Requirements for the Effective Support of Adaptive Mobile Applications," *Proc. Second Int'l Conf. Mobile Data Management,* Jan. 2001.

[28] B. Li and K. Nahrstedt, "Qualprobes: Middlewate QoS Profiling Services for Configuring Adaptive Applications," *Proc. IFIP/ACM Int'l Conf. Distributed Systems Platforms (Middleware),* Apr. 2000.

[29] G.J. Nutt, S. Brandt, A.J. Griff, S. Siewert, M. Humphrey, and T. Berk, "Dynamically Negotiated Resource Management for Data Intensive Application Suites," *Knowledge and Data Eng.,* vol. 12, no. 1, pp. 78-95, 2000.

[30] T. Fitzpatrick, G. Blair, G. Coulson, N. Davies, and P. Robin, "Supporting Adaptive Multimedia Applications through Open Bindings," *Proc. Int'l Conf. Configurable Distributed Systems (ICCDS '98),* May 1998.

[31] B.N. Jorgensen, E. Truyen, F. Matthijs, and W. Joosen, "Customization of Object Request Brokers by Application Specific Policies," *Proc. IFIP/ACM Int'l Conf. Distributed Systems Platforms (Middleware),* Apr. 2000.

[32] R. Kosner and T. Kramp, "Structuring QoS-Supporting Services with Smart Proxies," *Proc. IFIP/ACM Int'l Conf. Distributed Systems Platforms (Middleware),* Apr. 2000.

[33] P. Oreizy, M.M. Gorlick, R.N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D.S. Rosenblum, and A.L. Wolf, "An Architecture-Based Approach to Self-Adaptive Software," *IEEE Intelligent Systems,* vol. 14, no. 3, pp. 54-62, May 1999.

Eyal de Lara received the BS degree in computer science from the Instituto Tecnologico y de Estudios Superiores de Monterrey, Mexico, in 1995 and the MS and PhD degrees in electrical and computer engineering from Rice University in 1999 and 2002, respectively. He is an assistant professor in the Department of Computer Science at the University of Toronto. His research interests include distributed systems, networking, and mobile and pervasive computing. He is a member of the IEEE and the IEEE Computer Society.

Yogesh Chopra received the BS degree in electronics from Mumbai University, India, in 1996 and the MS degree in computer science from the University of Houston in 2002. He performed his master's research work at Rice University. He is currently a software development manager for iMimic Networking inc, and his research includes Web-caching, Web-acceleration, and cluster management for caches.

Rajnish Kumar received the BTech degree in computer science and engineering from the Indian Institute of Technology, Kharagpur, in 1999. He received the MS degree in computer science from Rice University. Currently, he is a PhD student in the College of Computing at the Georgia Institute of Technology. His research interests include systems and architectural support for mobile and distributed computing and wireless sensors networks. He is a student member of the IEEE and the IEEE Computer Society.

Nilesh Vaghela received the BS degree in electronics engineering from the University of Mumbai, India, in 1996. He received the MS degree in computer science from the University of Houston in 2002. He received the Best Master's student award from the University of Houston. His current research interest is in the area of distributed content delivery. Currently, he is a system software architect at iMimic Networking Inc.

Dan S. Wallach received the BS degree from the University of California, Berkeley, in 1993 and the PhD degree from Princeton University in 1998. He is an assistant professor in the Department of Computer Science at Rice University in Houston, Texas. His research involves computer security and the issues of building secure and robust software systems for the Internet. He also studies security issues that occur in distributed and peer-to-peer systems as well as the current generation of paperless electronic voting systems. Dr. Wallach is a member of the ACM, the IEEE, the IEEE Computer Society, and Usenix, and is a member of the editorial board of *IEEE Internet Computing*.

Willy Zwaenepoel received the BS degree from the University of Gent, Belgium, in 1979 and the MS and PhD degrees from Stanford University in 1980 and 1984, respectively. He is the Dean of the School of Computer and Communications Sciences of the Swiss Federal Institute of Technology Lausanne (EPFL). Before joining EPFL, he was on the faculty of Rice University, where he was the Karl F. Hasselmann Professor of Computer Science and Electrical and Computer Engineering. His interests are in all aspects of distributed computing. While at Stanford University, he was involved in the design and implementation of the V-System. At Rice University, he worked on two distributed shared memory systems, Munin and TreadMarks, on checkpoint/restart through coordinated checkpointing and message logging in the Manetho system. He was elected fellow of the IEEE in 1998 and fellow of the ACM in 2000. In 2000, he also received the Rice Graduate Student Association Teaching and Mentoring Award. He is a member of the IEEE Computer Society.