Component-Based Adaptation *

Eyal de Lara†, Yogesh Chopra§, Rajnish Kumar‡, Nilesh Vaghela§,
Dan S. Wallach*, and Willy Zwaenepoel°
delara@cs.toronto.edu, {yogesh, nilesh}@imimic.com, rajnish@cc.gatech.edu,
dwallach@rice.edu, willy.zwaenepoel@epfl.ch
†Department of Computer Science  § iMimic Networking, Inc.  ‡Department of Computer Science
University of Toronto  Georgia Institute of Technology
*Department of Computer Science  °School of Computer and Communication Science
Rice University  Ecole Polytechnique Federale de Lausanne, Switzerland

Abstract

Component-Based Adaptation is a novel approach to adaptation for mobile and wireless environments that supports powerful application-specific adaptations without requiring modifications to the application’s source code. Common productivity applications such as Netscape and PowerPoint export component-based APIs that allow external applications to control their behavior. The novel premise in component-based adaptation is that these APIs are sufficient to support a wide range of adaptation policies for applications running on resource-limited devices. This approach has a unique combination of advantages, including the ability to centrally manage the resource usage of multiple applications, and the ability to modify application behavior after the application has been deployed without making any changes to the application itself.

This paper evaluates the extent to which existing APIs can be used for the purposes of adaptation. In particular, we have implemented a large number of adaptations for applications from the Microsoft Office and the OpenOffice productivity suites and for Internet Explorer. Although we found limitations in their APIs, we were able to implement many adaptation policies without much complexity and with good performance. Moreover, component-based adaptation achieves performance similar to an approach that implements adaptation by modifying the application, while requiring a fraction of the coding effort.

*An earlier version of this paper appeared in the Proceeding of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS’01), San Francisco, California, March 2001. This paper extends the conference submission with results for more applications and a second operating system. It also compares component-based adaptation to in-application adaptation, an approach in which applications are modified to provide native support for adaptation.
1 Introduction

The need for application adaptation in mobile and wireless environments is well established [3, 7, 11, 15, 26]. Desktop applications, such as office productivity suites, typically expect that resources such as bandwidth and power are available in abundance [5]. 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 actually been deployed because existing approaches require extensive application source code modification [12, 14, 20] or have limited adaptive power [2, 16].

This paper presents Component-Based 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 component-based adaptation is that the applications provide mechanisms to enable adaptation by exposing run-time Application Programming Interfaces (API) to external programs. The adaptation system adapts applications without changing their source code by instead calling their APIs. For example, when browsing the Web on a mobile device the adaptation system reduces the 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 APIs to replace the original images. Applications thus become components that can be manipulated by the adaptation system. Because no source code modifications are necessary, component-based adaptation overcomes the principal roadblock to deploying adaptation.

Component-based adaptation allows sophisticated adaptations to be implemented without modifications to the applications. In particular, component-based adaptation supports adaptations that modify the behavior of the application (e.g., changing the order in which images are loaded) and not just the data the application operates on (e.g., scaling-down a large image). Modifying applications for the purpose of adaptation is at best unattractive, because of the complex nature of many of the applications, or 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 what other applications are run concurrently, such policy decisions should not be limited to application design time. Component-based 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 adaptation policies (e.g., for all applications, load all text first), which take into account the mix of applications running on the host [6].

Many applications, including office productivity suites and browsers, already provide well-documented APIs [19, 25]. 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 applications to reduce latency for reading multimedia documents over bandwidth-limited networks, but the same principles are applicable to reducing power usage [10].

To investigate these issues, we have built a system we call Puppeteer that uses component-based adaptation to adapt applications for reading multimedia documents over bandwidth-limited networks. We named our prototype Puppeteer because it uses the exposed API’s of 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 have been able to adapt a significant number of applications on different platforms in a relatively short time. In particular, we have implemented Puppeteer on Linux and Windows. On Linux we have experimented with adaptation policies for Presentation and Writer from OpenOffice on Linux. On Windows we used Outlook, PowerPoint and Word from Microsoft Office and Internet Explorer. Much of the code has been re-used 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 discusses how component-based adaptation differs from previous approaches to adaptation. Section 3 describes the capabilities of component-based adaptation and the requirements it places on the applications. This section also reflects on the limitations, from the point of view of adaptation, of the exposed APIs and file formats of the applications we experimented with. Section 4 presents the design and implementation of the Puppeteer system. 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. Finally, Section 6 concludes the paper.
2 Related Work

2.1 System-Based, In-Application, and Application-Aware 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.

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 [20, 16] and in-application adaptation [12, 14, 27]. 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 Another approach that tries to strike a middle ground between system-based and in-application adaptation is application-aware adaptation [2, 21]. 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 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. Component-based 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 component-based adaptation, it is the applications that exports 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. We argue that component-based adaptation is a more flexible approach to adaptation than application-aware adaptation. Component-based adaptation enables third parties to add new adaptation policies after the application has been released. In contrast, application-aware adaptation requires the application designer to foresee all necessary adaptations at the time the application is written.
2.2 Other Related Work

Visual Proxies [27], an offspring of Odyssey [21], 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 [8, 18, 22] 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.

The term component adaptation [9, 13, 17] is also used to describe efforts that monitor and adapt the communication path between distributed CORBA or DCOM components. These approaches enable client components to control their bindings to server components. Instead of getting a default static stub, clients components can choose among available bindings, or adapt a binding by, for example, inserting a compression filter in the data path. These approaches differ from component-based adaptation in that they do not take advantage of application APIs and are limited to data adaptations.

3 Component-Based Adaptation

Applications that operate on multimedia documents can be adapted by repeated use of two techniques: sub-setting and 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). Versioning chooses among the multiple instantiations of a component (e.g., instances of an image with different fidelity).

In component-based adaptation, adaptation policies use 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 have it from the beginning, and is one of the key advantages of component-based adaptation over system-based adaptation.
3.1 Application and API Requirements

For component-based adaptation to work, the application and its API must allow the adaptation system to discover, construct and display any meaningful subset or version of a document. A subset or a version is considered meaningful if it makes sense for the adaptation system to make use of such a subset or version. For example, it makes sense to display a single slide in a presentation or a single e-mail in a mail reader. It also makes sense to display the text in a document, and leave out the images and other embedded components, or it may make sense to display transcoded versions of images. We now examine this requirement in more detail.

First, the adaptation system needs to be able to discover the overall component structure of a document. 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 the boundaries between components. 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. The adaptation system also needs to be able to discover the type of each component. For example, in order to reduce the resolution of an image, the adaptation must be able to first discover that a component is an image.

Second, the application’s API must provide support for inserting any meaningful subset or version 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 each meaningful subset or version 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 certain events both in the system (“bandwidth dropped”) or in the application (“I am moving my mouse over this picture”). In order to be able to respond to the latter type of events, the application needs to export through its API an event registration and notification mechanism, by which the adaptation system can be notified of all relevant events.

3.2 Adaptations

We have built a system, called Puppeteer, that implements component-based adaptation on Microsoft Windows and on Linux platforms. The architecture and the implementation of Puppeteer is described later in the paper (see Section 4). We first discuss our experiences in implementing a variety of adaptation policies
using Puppeteer. In general, our experience is very favorable, and we succeeded in implementing a large
number of adaptation policies for popular applications on both platforms. On the Windows platform, we
use as applications PowerPoint, Word, and Outlook from Microsoft Office, and Internet Explorer (IE). On
Linux we use Presentation and Writer from the OpenOffice suite.

As in systems-based adaptation, we support all data adaptations. We focus our discussion on control
adaptations (see Section 2). These policies would be difficult to implement in system-based adaptation, be-
cause 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 exported APIs. As will be demonstrated in Section 5 these
policies also result in significant benefit under limited bandwidth conditions.

A very large number of adaptation policies are possible. We can, at best, give a sampling of the policies
that can be implemented with component-based adaptation. A first set of policies is based on repeated
application of subsetting. One policy that works for all applications is to load text first, return control to the
user immediately, and then load images and embedded elements in the background. User response can be
further improved by adding a data adaptation that compresses and decompresses the text. Policies can also
be constructed that re-arrange the order in which 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. In Presentation and PowerPoint we implemented a policy in which the first slide is loaded initially,
including text and images, then control is returned to the user, and the remaining slides are loaded in the
background. Other examples of subsetting policies that work for all 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 on the document).

A second set of policies is based on repeated application of versioning. One policy that can be im-
plemented in all of the applications 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 remain-
ing 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 re-arranging
the order in which subsets or versions are loaded based on certain events generated by the user. For example,
one policy first loads or refines the image over which the user moves the mouse.
The previous list only provides a sample of the adaptation policies that can be implemented by Puppeteer, but nonetheless attest to the power and the flexibility of component-based adaptation.

3.3 Existing APIs Limitations

While we have been able to achieve a large number of powerful adaptations, these applications and their APIs were not originally designed for adaptation. It then comes as no surprise that we encounter limitations in these applications, which prevent or limit our ability to adapt. Some limitations truly prevent us from performing certain adaptations we want to do. Others simply make it more difficult.

The most restrictive limitations include limits on the ability to recognize meaningful subsets and dependencies between different subsets. 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 clearly qualify as meaningful subsets, they are not reflected 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, 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 loading paragraphs (which, as opposed to pages, do appear as recognizable components in the file format) independently. Cross references and bibliographic citations are not re-evaluated 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 create or update any subset of the document 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 limited. For example, to upgrade the fidelity of an image in a PowerPoint presentation, a new temporary document has to be created containing the higher-fidelity version of the image. The image is then cut from the temporary document and pasted in its final place.

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 component than otherwise required. For example, in PowerPoint there is no easy way for updating the master slides, which store properties that are common to all slide 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.

4 Puppeteer

We have built a system, called Puppeteer, that implements component-based adaptation on Microsoft Windows and on Linux platforms. The rest of this section first describes the architecture of the Puppeteer system. We then reflect on our experience using Puppeteer to adapt popular applications to run on bandwidth-limited environments.

4.1 Puppeteer Architecture

To implement the policies described in Section 3.2, Puppeteer needs to be able to discover the structure of the document and extract meaningful subsets from it. It needs to be able to transcode certain components with a transcoder that is suitable for the type of component. Furthermore, it has to transmit subsets and versions over the low-bandwidth link, and update the application with those new subsets or higher-fidelity versions as they arrive. Finally, it needs to implement the various adaptation policies. In doing so, it must be able to register for notification of events that may trigger different adaptation choices. These events may originate both from the application (“user focus has changed”) and from the environment (“bandwidth has changed”).

Puppeteer has to deal with different platforms, communication substrates, and applications with different document formats and APIs. We designed Puppeteer following a modular architecture, which minimizes the overhead in moving to a new platform or adding a new application or adaptation policy.

Figure 1(a) 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 applications by calling on their exposed APIs. The Puppeteer remote proxy is responsible for parsing
documents, exposing their structure, and transcodling 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 Puppeteer local and remote proxies consist of four types of modules: Puppeteer Kernel, Drivers,
Transcoders, and Policies (see Figure 1(b)). The Puppeteer Kernel appears once in both the local and
remote Puppeteer proxies. A driver supports adaptation for a particular component type. As components
can contain other components within the hierarchy, so likewise drivers for one component may call upon
drivers for other components contained within them. 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 local and the
remote Puppeteer proxies, as may Transcoders which implement specific transformations on component
types. Policies specify particular adaptation strategies and execute in the local Puppeteer proxy.

4.1.1 Puppeteer Kernel

The Puppeteer Kernel is a component-independent module that implements the Puppeteer protocol. The
Puppeteer Kernel runs in both the local and remote proxies and enables the transfer of document compo-
nents. The Puppeteer Kernel does not have knowledge about the specifics of the documents being trans-
mitted. 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 ele-
ments in the document. The skeleton is a multi-level data structure as components in any level can contain
sub-components. 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 application policies to request a subset of the document’s components
and to specify transcoding filters to apply to the components’ data. To improve performance, the Puppeteer
Kernel batches requests for multiple components into a single message and supports asynchronous requests.

4.1.2 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. 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 can parse 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 understand the application’s file format. Unfortunately, running an instance of the application on the remote proxy creates more overhead and requires both the local and remote proxies to run the environment of the application, 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 we implemented for 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 un-parse 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. On the other hand, for applications that must explicitly read every item they display (e.g., IE, Netscape), the driver instructs the application to reload the component (i.e., asking IE to refetch a URL).

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.

4.1.3 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 HTML page with only a subset of images embedded in the original page).
4.1.4 Policies

Policies are modules that run on the local proxy and control the fetching of components. 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 Puppeteer Kernel, relieving the policy from the intricacies of communication.

4.1.5 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 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. 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 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 exported APIs to feed those newly fetched components to the application.
Table 1: Import, export and tracking drivers for PowerPoint, Word, Outlook email, Internet Explorer (IE), Presentation, and Writer. The table shows for each tracking driver the method it uses for uncovering the skeleton, the document format, and object types that the driver recognizes. The table also shows the technique used by export drivers and the set of events trapped by tracking drivers.

### 4.1.6 User Interaction with Puppeteer

In our current prototype, 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 of 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 are in progress of being loaded. Users can also interact with the Component Viewer to control the fetching of component versions.

### 4.1.7 Implementation Summary

The Puppeteer system is written in Java. On the Windows platform, we use as applications PowerPoint, Word, and the Outlook emailer from Microsoft Office, and Internet Explorer (IE). On Linux we use Presentation and Writer from the OpenOffice suite. For these applications, we implemented the adaptation policies described in Section 3.2. Table 1 provides a summary of the rest of the implementation. Most of the entries in this table are self-explanatory. There are no events handled in OpenOffice Presentation and Writer, as their APIs do not yet support event handling in the version that we used. 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
Module | Code Lines
---|---
Puppeteer Kernel | 9193
Policies | 177
First Slide & Text | 159
Text & Small Images | 334
JPEG & Text Compression | 670
Total |
Common to PowerPoint and Word | 484
Import Driver | 396
Transcoder | 88
Total |
PowerPoint | 1193
Import Driver | 761
Export Driver | 153
Track Driver | 146
Transcoder | 133
Total |
Word | 537
Import Driver | 250
Export Driver | 158
Track Driver | 129
Total |
Internet Explorer | 554
Import Driver | 314
Export Driver | 175
Track Driver | 65
Total |
Outlook email | 1061
Import Driver | 787
Export Driver | 200
Track Driver | 74
Total |
Common to Presentation and Writer | 615
Import Driver | 163
Presentation | 374
Export Driver | 241
Total |
Writer | 439
Import Driver | 242
Export Driver | 197
Total |

Table 2: Code line counts for Puppeteer Kernel, Policies, and Drivers for PowerPoint, Word, Outlook email, Internet Explorer, and OpenOffice Presentation and Writer.

its components from the mail server. We built the export and tracking drives for our Microsoft Windows-based applications using COM interfaces. For our Linux-based applications we used UNO interfaces, which are based on the CORBA standard.

4.2 Implementation Experience

Our experience in implementing a variety of adaptation policies using component-based adaptation is very favorable. The modular Puppeteer architecture has proven successful in supporting a large number of adaptation policies for popular applications on different platforms. In terms of portability, the Puppeteer Kernels
used on the Windows and Linux platforms are identical. In terms of programming effort to implement new applications and new policies, Table 2 shows the code line counts for the various modules we implemented. 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 a 150 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. 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 allowed 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.

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 modified 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 also 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 used 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 [24]. 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 the PC playing the role of the mobile wireless client and 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 datasets we use for PowerPoint, Word, Presentation, and Writer consist of subsets of the documents downloaded from the Web and characterized in de Lara et al. [5]. For Presentation and Writer we converted the documents from their original PowerPoint and Word formats to the OpenOffice formats. IE loads HTML documents downloaded from the Web by re-executing the Web traces on Cunha et al. [4]. Finally, the Outlook emailer loads synthetic emails we created by adding image attachments of different sizes to a simple text message.

In the remainder 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 by modifying its source code. Finally, we quantify the Puppeteer overhead.

### 5.1 Some Adaptation Policies

This section illustrates the performance of some sample adaptation policies implemented in Puppeteer. Figures 2, 3, and 4 show the latencies for these sample policies as a function of the total size of the documents. 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 solely dependent on the size of the data transferred, growing 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.).
Figure 2: Fetch First Slide and Text. Latency for loading documents with PowerPoint over 384 Kb/sec (a), 1.6 Mb/sec (b), and 10 Mb/sec (c), and with Presentation over 384 Kb/sec (d), 1.6 Mb/sec (e), and 10 Mb/sec (f). 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).
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 adaptation policies that load just the components of the first slide and the text component of all remaining slides before they return control to the user. Afterwards the components of the remaining slides are loaded in the background. 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.

Figure 2 shows the results of these experiments under the labels \textit{PowerPoint.slide+text} and \textit{Presentation.slide+text} 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 PowerPoint (\textit{PowerPoint.native}) or native Presentation (\textit{Presentation.native}), and Puppeteer runs for PowerPoint and Presentation for loading in all the elements of the first slide and the text for all remaining slides (\textit{PowerPoint.slide+text}, \textit{Presentation.slide+text}). Figures 2(d) through 2(f) also plot latencies for loading the documents with a modified version of Presentation, which supports bandwidth adaptation within the application. We differ the discussion of these results to Section 5.2.

We expected 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 are too small to account for the savings. The savings result, instead, from reducing the parsing and rendering time.

On average, for PowerPoint, \textit{PowerPoint.slide+text} achieves latency reductions of 75%, 71%, and 54% for documents larger than 1 MB on 384 Kb/sec, 1.6 Mb/sec, and 10 Mb/sec networks, respectively. For Presentation, \textit{PowerPoint.slide+text} achieves latency reductions of 61% and 36% for documents larger than 1 MB on 384 Kb/sec and 1.6 Mb/sec networks, and 13% for documents larger than 10 MB on the 10 Mb/sec network, respectively. Moreover, the results show 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 10.9% for documents larger than 4 MB).

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. Figures 3 shows the latency for loading the Word and Writer documents over 56 Kb/sec, 384 Kb/sec, and 10 Mb/sec networks. The figures
Figure 3: **Text and Small Images.** Latency for loading documents with Word over 56 Kb/sec (a), 384 Kb/sec (b), and 10 Mb/sec (c), and with Writer over 56 Kb/sec (d), 384 Kb/sec (e), and 10 Mb/sec (f). 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).
show latencies for native Word and Writer (Word.native, Writer.native), and for Puppeteer runs that load only the text an 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). Figures 3(d) through 3(f) also plot latencies for loading the documents with a modified version of Writer, which supports bandwidth 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 document, smallimag achieves an average reduction in latency of 55% and 57% for Word and Writer over 56 Kb/sec respectively. Word.textcomp+smallimag and Writer.textcomp+smallimag further reduces latency for all documents. On average, textcomp+smallimag achieves a reduction in latency of 85% and 59% for all Word and Writer documents over 56 Kb/sec respectively, and of 61% and 50% for Word and Writer documents larger than 1 MB over 384 Kb/sec, respectively. The textcomp+smallimag latency reductions for Writer are smaller than for Word because of the larger Puppeteer overhead for Writer documents. For example, the average Puppeteer overhead for Writer documents smaller than 1 MB is 27% versus just 6% for Word documents. Figure 3(c) and (f) 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 technology to reduce 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 elements.

Our prototype converts, at run time, GIF and JPEG images embedded in the HTML documents or emails into a progressive JPEG format using the PBMPlus [23] and Independent JPEG Group [1] libraries. We then transfer only the first $\frac{1}{7}$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 run the transcoder.

Figure 4 shows the latency for loading the HTML documents over 56 Kb/sec, 384 Kb/sec, and 10 Mb/sec networks and the emails 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-
Figure 4: JPEG and Text Compression. Latency for loading document with IE over 56 Kb/sec (a), 384 Kb/sec (b), and 10 Mb/sec (c), and emails with Outlook over 384 Kb/sec (d), 1.6 Mb/sec (e), and 10 Mb/sec (f). 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).
compressed text since the text components of our emails were small (under 8 KB) and the results were 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% and 35% 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 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 of 76% and 50% 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 71% and 85% for 384 Kb/sec and 1.6 Mb/sec, 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.

5.2 Comparison with In-Application Adaptation

In this section, we compare component-based 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 the Microsoft-based applications we experimented with. 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 modified Presentation and Writer to implement the adaptive policies described in Section 3.2. 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.

Figures 2(d) through (f) 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 just the components of the first slide and the text component of all remaining slides. The components of the remaining slides are loaded afterwards in the background.

The results for native Presentation and Presentation with Puppeteer support were 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 was expected as Adaptive Presentation incurs in less extra processing. 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. However, the performance gap between Adaptive Presentation and Puppeteer is small averaging 4%, 13%, and 28% 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.

Figures 3(d) through (f) plot the latencies for loading documents over 384 Kb/sec, 1.6 Mb/sec, and 10 Mb/sec network links with native Writer without any adaptation support (Writer.native), Writer with Puppeteer support (Writer.smallimag and Writer.text+smallimg), and Adaptive Writer (InApplication.text+smallimg). The Puppeteer and Adaptive Writer runs implement an adaptation policy that returns control to the user after it loads just the text and image components smaller than 4 KB. The remaining images are loaded afterwards in the background. The results for Adaptive Writer are similar to those for 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.

Adding native support for bandwidth adaptation to an application requires creating a proxy. The proxy is located at a well-connected part of the network, at the other end of the bandwidth-limited link. The proxy has to support downloading documents from their storage location (e.g., network file system, WWW), servicing
individual document components, and performing transcoding transformations. The modified application running on the mobile device then interacts with the proxy to request individual document components.

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 a 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 limited 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 consist 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 average our adaptation policies required less than 150 lines of code. Moreover, we were 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 component-based adaptation and in-application adaptation derives from the level of abstraction at which adaptation policies are defined. Adaptation policies in Puppeteer tend to be written at a high level of abstraction, as the APIs applications expose, and that Puppeteer uses to implement adaptations, 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, and vendors are limited to adding new calls to an exposed API to reflect new functionality.

On the upside, because in-application adaptation has full access to the application’s source code, it

\[1\text{In COM this requires creating a brand new interface, which allows new clients to exploit the new functionality, while older clients retain forward compatibility.}\]
is not limited by the specific functionality that the application choose to export through its API, or by other application-defined constraints. Therefore, in principle a wider set of adaptation policies could be implemented (albeit at a higher programming cost) with in-application adaptation than with component-based 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 component-based adaptation. However, the experimental results show 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 component-based adaptation.

5.3 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 APIs 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, email, 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).

Figures 5(a) and (b) show latency overhead and Figure 6 shows data overhead for loading entire documents with PowerPoint, Word, IE, Outlook, Presentation, and Writer. We maximized the data overhead by simulating 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
Figure 5: Initial adaptation costs. Figures (a) and (b) show latency overhead for loading documents and emails with Puppeteer for PowerPoint, Word, IE, Outlook, Presentation, and Writer.

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% over 384 Kb/sec and 4.7% over 56 Kb/sec, respectively.

Figure 6 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 Figures 5(a) and (b). 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% and 1.3%, 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 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.

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 2994 milliseconds, with a standard deviation of 3943 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 dataset, 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%), and that most of the time that it takes to add or upgrade a component is spent transferring the data over the network.

6 Conclusions

This paper presents the concept of component-based adaptation. Underlying component-based 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 component-based adaptation in the Puppeteer system. We have
used Puppeteer to evaluate the extent to which existing APIs can be used for the purposes of adaptation. 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 proved 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


