RICE UNIVERSITY

Component-Based Adaptation for Mobile Computing

by

Eyal de Lara

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

Willy Zwaenepoel
Hasselman Professor, Chair
Computer Science and Electrical and
Computer Engineering

Edward W. Knigthly
Associate Professor
Electrical and Computer Engineering

Dan S. Wallach
Assistant Professor
Computer Science

David B. Johnson
Associate Professor
Computer Science

Houston, Texas

April, 2002

Component-Based Adaptation for Mobile Computing

Eyal de Lara

Abstract

Component-based adaptation is a novel approach for adapting applications to the limited availability of resources such as bandwidth and power in mobile environments. Component-based adaptation works by calling on the run-time APIs that modern component-based applications export. Because source code modification is not necessary, even proprietary applications such as productivity tools from Microsoft's Office suite can be adapted. Moreover, new adaptive behavior can be added to applications long after they have been deployed. Even if source code is available, development time for implementing adaptation is much reduced.

In addition, the ease with which adaptations can be implemented in this framework has enabled me to explore new avenues in adaptation. First, I have developed the first adaptive system to support document editing and collaboration over bandwidth-limited links. The key insight gathered from this work is that support for adaptation is orthogonal to concurrency and consistency mechanisms, and therefore can be integrated easily in existing systems. Second, I have developed a hierarchical adaptive transmission scheduler to support coordinated multi-application adaptation.

I have demonstrated the effectiveness of component-based adaptation by implementing a system called Puppeteer, which has allowed me to adapt widely deployed applications, such as productivity tools from Microsoft's Office suite and Sun Microsystems' OpenOffice suite. Although the APIs of these applications impose some limitations, I have been able to implement a wide range of adaptation policies for

reading, editing, and collaboration, with modest implementation effort and good performance results.

Acknowledgments

I want to thank the members of my committee, Dr. Willy Zwaenepoel, Dr. Dan S. Wallach, Dr. David B. Johnson, and Dr. Edward Knightly, for their guidance and support in writing this thesis. I am convinced that without their assistance, this thesis would not have been possible.

I want to thank my advisor Willy Zwaenepoel, for allowing me to spend the last four years doing exciting research, even when this meant moving outside of his traditional research areas. His opportune advise ensured that, while I was having fun playing with mobile computers, I would still have at the end some real research to show for it.

I want to thank Dan S. Wallach for being my co-advisor, as well as a friend. My PhD experience would not have been nearly as full filing without the late-night cheese steak sandwiches and the high-speed downtown dashes to meet the last FEDEX pickup deadline.

I want to thank the members of the Puppeteer team for their support. I want to thank Nilesh Vaghela, Yogesh Chopra, and Rajnish Kumar for experimenting with applications from the OpenOffice suite and the Outlook email reader. The experimental results that I present in this thesis for these applications reflect their efforts. I want to thank Yuri Dotsenko for developing the JPEG transcoding support.

I would like to express my appreciation to my parents and brothers for always believing in me and encouraging me to pursue my dreams.

Last, but dearest to my hart, I want to thank my wife Monica for her support and encouragement. It is safe to say that without you, I would not have had the courage to pursue the path that this dissertation concludes. I love you.

Contents

Illustrations

Tables

Chapter 1

Introduction

The need for application adaptation in mobile and wireless environments is well established [13, 41, 52, 76, 102, 115, 121, 122, 131]. In an ideal world, users of mobile and wireless devices would be able to access remote data using the same applications that are available to them on their desktop computers. Unfortunately, desktop applications, such as office productivity suites, typically expect that resources such as bandwidth and power are available in abundance [35]. Mobile and wireless environments are, by contrast, characterized by limited and unreliable resource availability. For instance, a mobile client might access the network over a high bandwidth wireless connection when inside a building, but has to rely on a low bandwidth wireless connection when outdoors. Hence, to perform properly in these environments, the applications need to adapt to the available resources.

Several approaches for adapting applications to run on bandwidth-limited networks have been proposed. The need for extensive modifications to the application's source code in order to add adaptation has, however, limited the wide deployment of adaptation. Moreover, these previous approaches to bandwidth adaptation do not support system-wide adaptation policies, which coordinate the adaptation of all applications running on the bandwidth-limited device, and do not enable editing documents and engaging in collaborative work over bandwidth-limited networks.

This thesis has three main contributions. First, it presents *component-based adaptation*, a new approach to application adaptation. 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 over a bandwidth-limited link 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 replaces the browser's original images, using the browser's APIs. 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.

Second, this thesis describes, how by extending component-based adaptation with dynamic control over bandwidth scheduling, it is possible to adapt multiple applications in a coordinated fashion. Some sample system-wide bandwidth adaptation policies include: prioritizing the transmission of text over image data across all applications; and shifting most of the available bandwidth to the document that is currently in the foreground, while reducing the fidelity of the background documents. This thesis also presents the design, implementation, and evaluation of the Hierarchical Adaptive Transmission Scheduler (HATS), which provides fine grain control over the transmission of component data. HATS enables the adaptation system to enforce the network allocations determined by the adaptation policies, to react quickly to changes in network connectivity and user behavior (e.g., by shifting bandwidth to the higher priority components), and to implement sophisticated transmission policies that give different priorities to the hierarchy of applications, documents, and components running on the device. HATS also supports giving different priorities to various parts of a component's data. For example, fetching the first half of a component's data can have higher priority that fetching the rest of the data.

Third, this thesis presents CoFi, a unified architecture that combines the notions of *co*nsistency and *fi*delity, and supports document editing and collaborative work over bandwidth-limited links. CoFi supports editing documents that have been adapted on read and are only partially loaded at the bandwidth-limited device. CoFi exploits the component structure of the document to track changes made to each component by the user, and distinguishes those changes from the ones that result from adaptation. CoFi can then propagate only the user's modifications to components. CoFi enables users to collaborate while connected over bandwidth-limited links by supporting the partial and incremental propagation of modifications. For example, under low bandwidth conditions, a user can choose to propagate only a portion of the modified components, or can transcode components and propagate lower-fidelity versions. Later, on reconnecting over higher-bandwidth links, the user can propagate the remaining components or upgrade the fidelity of components that were previously propagated in low fidelity.

To evaluate the effectiveness of component-based adaptation, HATS, and CoFi, I have built a system called Puppeteer as a proof-of-concept implementation for experimenting with adapting widely deployed applications such as Internet Explorer, Microsoft PowerPoint, and Sun's OpenOffice. Puppeteer adapts applications for reading, editing, and collaboration over bandwidth-limited devices and supports system-wide adaptation policies that adapt multiple applications in concert. The programming effort to support new applications and new adaptation policies is modest, and code can be re-used to implement similar policies for different applications. Puppeteer supports powerful adaptations, showing significant latency and bandwidth reductions for various adaptations over bandwidth-limited links, and adds little overhead when no adaptation is done.

Finally, while the focus of this thesis is on adapting applications to bandwidth-limited networks, component-based adaptation and Puppeteer has proven effective in reducing power usage [49, 48].

The rest of this thesis is organized as follows:

- Chapter 2 presents background information on adapting applications for reading documents over bandwidth-limited links and compares how component-based adaptation differs from earlier approaches to adaptation. This chapter also provides basic background on optimistic consistency models, which forms the basis for the contributions of this thesis in adding adaptive support for document editing and collaboration over bandwidth-limited links.

- Chapter 3 describes the techniques that component-based adaptation uses to adapt applications. This chapter also outlines the requirements that an application must meet in order to support component-based adaptation, and presents several examples of possible adaptation policies.
- Chapter 4 presents the design and architecture of the Puppeteer adaptation system. This chapter describes how Puppeteer's modular architecture allows adding new platforms, applications, and adaptation policies with modest effort.
- Chapter 5 describes how component-based adaptation adapts applications for reading over bandwidth-limited links. This chapter explores the extent to which run-time interfaces provided by existing applications support bandwidth adaptation. This chapter also presents experimental results for adapting several existing applications with the Puppeteer adaptation system for reading documents over bandwidth-limited links.
- Chapter 6 describes how, by extending component-based adaptation with dynamic control over bandwidth scheduling, it is possible to implement system-wide adaptation policies that adapt multiple applications in concert.
- Chapter 7 describes the CoFi protocol, which adds adaptive support for document editing and collaboration over bandwidth-limited links. This chapter also presents the implementation and evaluation of a CoFi prototype on top of the Puppeteer adaptation system.
- Chapter 8 summarizes the thesis and presents my conclusions and directions for future work.

Chapter 2

Background

This chapter presents general background on adapting applications for reading documents over bandwidth-limited links and on optimistic replication models. This information sets the stage for Chapters 5, 6, and 7, which describe the contributions of this thesis in adapting applications for reading, editing, and collaboration over bandwidth-limited links and for supporting system-wide adaptation policies that adapt multiple applications in concert.

2.1 Adaptation

Desktop applications, such as office productivity suites, typically expect that resources such as bandwidth and power are available in abundance [35]. Mobile and wireless environments are, by contrast, characterized by limited and unreliable resource availability. For instance, a mobile client might access the network over a high bandwidth wireless connection when inside a building, but has to rely on a low bandwidth wireless connection when outdoors. Hence, to perform properly in these environments, the applications need to adapt to the available resources.

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 instance, a control adaptation could cause an application that otherwise returns control to the user only after an entire document is loaded to return control as soon as the first page is loaded.

There are many possible implementations of bandwidth adaptation. Based on where the adaptation is implemented, I recognize a spectrum of possibilities with two extremes: system-based [34, 80, 98, 136, 140] and application-based adaptation [12, 31, 40, 74, 53, 54, 72, 71, 73, 75, 123]. 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 application-based adaptation, only the application is changed. Application-based adaptation allows both data and control adaptation, while system-based adaptation is limited to data adaptation. Another approach that tries to strike a middle ground between system- and application-based adaptation is application-aware adaptation [10, 103]. 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 application-based 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 adaptation policies are implemented. In component-based adaptation, the adaptation policies run inside the centralized adaptation system and adapt by invoking interfaces the applications export. The precise opposite occurs in application-aware adaptation where the applications are modified to implement adaptation policies that call on the system's adaptation API. I argue that component-based adaptation is a more flexible approach to adaptation. Since no source modifications are necessary, component-based adaptation enables third parties to adapt applications for which source code is not available, and to add new adaptation policies after the application has been deployed.

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. Embedding adaptation policies in the application furthermore 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 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.

2.2 Optimistic Replication

Maintaining consistency in mobile clients is hard. Mobile clients may operate for long periods of disconnection at a time, and tend to communicate over bandwidth-limited links. These characteristics, make it impractical for mobile clients to implement the strong replication models that are traditionally used in parallel applications [5, 18, 56, 77, 79, 148] and distributed file systems [88] such as NFS [119], Sprite [101], AFS [64], xFS [11], Frangipani [138]. To maintain consistency, these models require clients to get exclusive access to objects before modifying them [8, 57, 63, 139], and that all clients always see the most up to date version of an object.

The overhead of obtaining exclusive access is acceptable for static clients that communicate over stable high-bandwidth links. The strong connectivity enables static clients to delay obtaining exclusive access just prior to writing to a file; a process that usually involves requesting a lock. Since locks are held only while the client writes, other clients are prevented for accessing the object only for just a short period of time. The strong connectivity implies that invalidation can be delivered most of the time and the high network capacity enables quick delivery of invalidations and updates.

For mobile clients, however, lazy locking is not practical, as it is not possible to request locks while disconnected. Instead, if a system wants to ensure strong consistency, it has to grant write locks for any object cached by the mobile client at disconnection. This pessimistic approach is impractical, however, as it may lock-out other clients for long periods of time — until the mobile client reconnects again. Moreover, it is unlikely that the mobile client will even modify more than a small fraction of the objects it caches.

Instead, systems like Bayou [44, 110], Ficus [61, 62, 107, 112], Coda [42, 80, 124], Reconciliation [65] and variations of AFS [66, 67] enable mobile clients to modify objects without requiring exclusive access, using a technique know as *optimistic replication* [23, 32, 33]. In these systems, clients modify their local copy of an object and propagate their modifications to other servers and clients when they reconnect to the network. The weaker consistency model, however, may result in update conflicts [62], as multiple users might simultaneously modify the same files. To deal with this problem, researchers have experimented with automatic and application-specific conflict resolution [6, 43, 83, 85, 84, 116, 126, 137]. There always remains, however, a number of conflicts that can not be resolved automatically and that require user intervention.

The consistency unit of AFS, Coda, and Ficus is the file. This design choice is predicated on results from studies of file access in UNIX systems [14, 80, 105, 116] where concurrent modifications to a file are rare. These studies, however, reflect traditional research environments where there is little file sharing and when sharing occurs, it is usually under strict coordination between the users (e.g., modifications to disjointed parts of a source code tree). It is therefore not clear whether using files as the consistency unit is appropriate for environments where collaborative work, and therefore sharing, is more common. Bayou, on the other hand, supports application-specific consistency units. It, however, requires applications to conform to a special API for all of their data accesses.

Chapter 3

Component-based Adaptation

The novel premise in component-based adaptation is that the applications provide mechanisms to enable adaptation by exposing run-time Application Programming Interfaces (APIs) to external programs. The adaptation system adapts applications *without* changing their source code, by instead calling their APIs. For example, when browsing the Web over a bandwidth-limited link, 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 replaces the browser's original images using the browser's APIs. 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, as I will demonstrate in this thesis, it allows — without modification to the application — adaptations that modify the behavior of the application and not just the data that it is operating on.

Component-based adaptation implements adaptation by *repeated* use of *subsetting* 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). The 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 previously not been available except in applications expressly designed to include it, and is one of the key advantages of component-based adaptation over system-based adaptation [80, 98].

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 meaningful subsets or versions 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 instance, it makes sense to display a single slide in a presentation or a single email in a email reader. It also makes sense to display the text in a document and leave the images and other embedded documents out, or it may make sense to display transcoded versions of images. I now examine this requirement in more detail.

First, the adaptation system needs to be able to discover the overall structure and the meaningful subsets 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 the subsets. For instance, in a presentation, the boundary between different slides must be visible. Alternatively, this can be done by running an instance of the application and extracting the structure and the subsets by means of API methods.

Second, the application's API must provide support for inserting any meaningful subset or version into the application. Since one version may be replaced by a different one, the API needs to support such replacements.

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

Finally, the adaptation system needs to be able to discover the type of a component. For instance, in order to reduce the resolution of an image, the adaptation must be able to first discover that a component is an image.

Additionally, to provide powerful adaptation policies, the adaptation system must be able to respond to certain events both in the system ('available bandwidth has 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 Some Sample Adaptation Policies

As in system-based adaptation, component-based adaptation supports all data adaptations. I focus our discussion on control adaptations — adaptations that modify the behavior of the application and not just the data on which it is operating. These policies would be difficult to implement in system-based adaptation, because they affect not only the data used by the application, but also its control flow. Such adaptation policies have, to the best of our knowledge, only been implemented by modifying the application. In component-based adaptation, however, they are implemented by using the exported APIs. As will be demonstrated in Chapters 5 and 7, these policies also result in significant benefit under limited bandwidth conditions.

A very large number of adaptation policies are possible. I 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 many 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. Differing policies can also be constructed by re-arranging the order in which images are loaded. For instance, small images could be loaded first. Subsets other than text can also be chosen for initial loading before control is returned to the user. For a presentation application, I could implement a policy in which the first slide is loaded initially, including text and images, and then control is returned to the user, with the remaining slides 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 OLE embedded object) 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 versioning. One policy that can be implemented for many applications uses Progressive JPEG compression to reduce latency. This adaptation policy first converts all images in the document into Progressive JPEG [1]. A prefix of the resulting JPEG image file is loaded, producing an image with limited resolution, and control is then returned to the user. 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 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 rearranging the order in which subsets or versions are loaded, based on certain events generated by the user. For instance, 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 with component-based adaptation, but these policies nonetheless attest to the power and the flexibility of component-based adaptation.

Chapter 4

Puppeteer

I have built a system, called Puppeteer, that implements component-based adaptation on Microsoft Windows and on Linux platforms. Figure 4.1 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. 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 strong connectivity to the data server(s). Data servers can be arbitrary repositories of data such as Web servers, file servers or databases.

4.1 Puppeteer Proxy Architecture

Puppeteer needs to be able to discover the structure of the document and extract from it meaningful subsets. 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 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 certain events that may trigger different adaptation choices. These events may originate both from the application and from the environment ("bandwidth has changed").

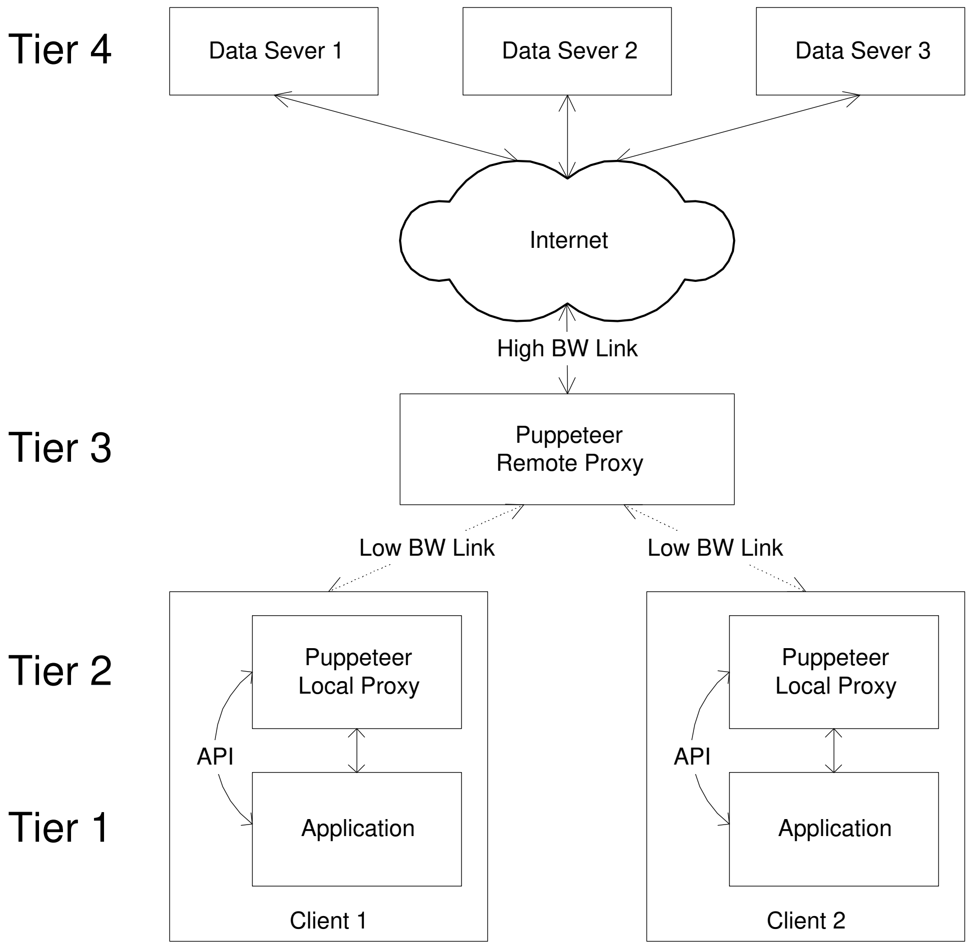


Figure 4.1 : Puppeteer system architecture.

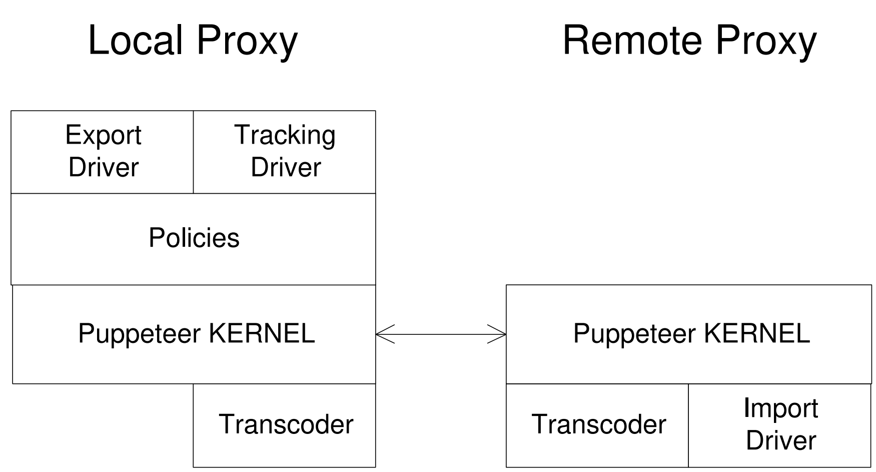


Figure 4.2 : Puppeteer local- and remote-proxy architecture

Clearly, Puppeteer will have to deal with different platforms, communication substrates, and applications with different document formats and APIs. In order to minimize the overhead in moving to a new platform or adding a new application or adaptation policy, we designed Puppeteer following a modular architecture.

The Puppeteer local and remote-proxy architectures consist of four types of modules: Kernel, Driver, Transcoder, and Policy (see Figure 4.2). The Kernel appears once in both the local and remote Puppeteer proxy. A driver supports adaptation for a particular component type. As components can contain other components within the hierarchy, 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 Kernel

The Puppeteer Kernel is a component-independent module that implements the Puppeteer protocol. The Kernel runs in both the local and remote proxyes and enables the transfer of document components. The Kernel does not have knowledge about the specifics of the documents being transmitted. It operates on a format-neutral description of the documents, which I refer to as the Puppeteer Intermediate Format (PIF). A PIF consists of a *skeleton* of *components*, each of which has a set of related *data items*. The skeleton captures the structure of the data used by the application. The skeleton has the form of a tree, with the root being the document, and the children being pages, slides, or any other elements in the document. The skeleton is a multi-level data structure, as components in any level can contain sub-components. Skeleton nodes can have component-specific properties attached to them (e.g., slide title, image size), as well as one or more related data items that contain the component's native data.

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

4.1.2 Drivers

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 NT 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 need not 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.

Parsing at the remote proxy does not work well for 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. To address this problem, references to unknown components must be trapped while the script runs inside the client application, updating the skeleton appropriately.

Export drivers un-parse the PIF and update the application using the application's exported API. A minimal export driver must support inserting new components into the running application.

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 or Star 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 or Netscape), the proxy instructs the application to reload the component (e.g., 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 adaptation policies use transcoders to perform both subsetting and versioning transformations. These transcoders operate by either modifying the encoding of a component's data or by changing the relationship between a component and its children.

4.1.4 Policies

Policies are modules that run on the local proxy and control the fetching of components. These policies traverse the skeleton, choosing what components to fetch and with what fidelity.

Puppeteer provides support for two types of policies: general-purpose policies that are independent of the component type being adapted (e.g., prefetching) and component-specific policies that use their knowledge about the component to drive the adaptation (e.g., fetch the first page only).

Typical policies choose components and fidelities based on available bandwidth and user-specified preferences (e.g., fetch all text first). Other policies track the user (e.g., fetch the PowerPoint slide that currently has the user's focus and prefetch subsequent slides in the presentation), or react to the way the user moves through the document (e.g., if the user skips pages, the policy can drop components it was fetching and focus the available bandwidth on fetching components that will be visible to the user).

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

4.2 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 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 Kernel then transfers the document's skeleton to the Puppeteer local proxy. The policies running on the local proxy ask the Kernel to fetch an initial set of components at a specified fidelity. This set of components is supplied to the application in return to its open call. The application, believing that it has finished loading the document, returns control to the user.

Meanwhile, Puppeteer knows that only a fraction of the document has been loaded. The policies in the local proxy now decide what further components or version of components to fetch. They instruct the Kernel to do so, and then use the application's exported APIs to feed those newly fetched components to the application.

Chapter 5

Adaptation for Reading Documents

Many applications, including office productivity suites and browsers, already provide well-documented APIs. This chapter addresses the question to what extent these existing APIs can support adaptation for reading documents over bandwidth-limited networks. To the best of my knowledge, this is the first effort to use these interfaces for adaptation.

The rest of this chapter is organized as follows. Section 5.1 discusses my experiences in implementing a variety of adaptation policies using Puppeteer. In general, my experience was very favorable, and I succeeded in implementing a large number of adaptation policies for popular applications on both Windows and Linux platforms. Section 5.2 evaluates the performance and overheads of several adaptation policies. Section 5.3 reflects on limitations of existing APIs that prevent or complicate adaptation. Section 5.4 reviews related work in document adaptation and uses of component-based technologies to extend application functionality. Finally, Section 5.5 summarizes and concludes the chapter.

5.1 Implementation

The Puppeteer system is written in Java. This thesis reports on experiments with PowerPoint, Word, and Outlook from Microsoft Office, and Internet Explorer (IE) running on the Windows platform. The thesis also presents results for experiments with Presentation and Writer from the Sun's OpenOffice suite running on Linux.

The export and tracking drives for the Microsoft Windows-based applications are implemented using COM interfaces [96, 97, 118, 26]. The drivers for the Linux-based

		PowerPoint	Word	Outlook	IE
Import driver	Method	Parsing	Parsing	IMAP	Parsing
	Format	XML	XML	IMAP	(D)HTML
	Comp. types	PowerPoint Slide Image Sound OLE	Word Image Sound OLE	Email Image	HTML Image
Export driver		Cut & paste	Cut & paste	Reload	Reload
Tracking driver events		Open Select slide	Open Select image	Read Inbox Open attachment	Browse Mouse over image

Table 5.1 : Import, export and tracking drivers for PowerPoint, Word, Outlook, Internet Explorer (IE). The table shows for each import 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.

applications use UNO interfaces, which are based on the CORBA standard.

For these applications, I implemented, in collaboration with Nilesh Vaghela, Yogesh Chopra, and Rajnish Kumar, the adaptation policies described in Section 5.2.1. Tables 5.1 and 5.2 provide 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 I 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 in the import driver, with the exception of Outlook, which uses the IMAP protocol to obtain the mailbox structure and its components from the mail server.

The modular Puppeteer architecture has proven successful in supporting a variety of adaptations for different applications on different platforms. In terms of portability, the Kernels used on the Windows and Linux platforms are identical. In terms of programming effort to implement new applications and new policies, Table 5.3 shows

<table>
<tr><td colspan="2"></td><td>Presentation</td><td>Writer</td></tr>
<tr><td rowspan="3">Import driver</td><td>Method</td><td>Parsing</td><td>Parsing</td></tr>
<tr><td>Format</td><td>XML</td><td>XML</td></tr>
<tr><td>Comp. types</td><td>Presentation
Slide
Image</td><td>Writer
Image</td></tr>
<tr><td colspan="2">Export driver</td><td>Reload</td><td>Reload</td></tr>
</table>

Table 5.2 : Import and export drivers for Presentation and Writer. The table shows for each import 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.

the code line counts for the various modules I implemented. The line counts for the Kernel module include the implementations of Puppeteer protocol and support for text and progressive JPEG image compression. The relevant conclusion from this table is that the application- and policy-independent 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 policies is small as well, on average requiring less than a 150 lines, even including some of the more complicated adaptations.

I also note that both Presentation and Writer [2] store their data natively in XML format, and both PowerPoint and Word support XML formats as well [95]. This allowed me to use a common XML parsing package, supplied as part of Sun's Java library, reducing the size and complexity of our input drivers.

5.2 Experiments

All experiments were performed on a platform that consists of three Pentium III 500 MHz PCs running either Windows 2000 or RedHat Linux 7.0. One PC functions as a data server running Apache 1.3 or Cyrus IMAP 1.6.24 and stores all the documents and emails used in the experiments. The second PC runs a Puppeteer remote proxy. The third PC runs the user's applications and the Puppeteer local proxy. The

Module		Code Lines
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	Import Driver	761
	Export Driver	153
	Track Driver	146
	Transcoder	133
	Total	1193
Word	Import Driver	250
	Export Driver	158
	Track Driver	129
	Total	537
Internet Explorer	Import Driver	314
	Export Driver	175
	Track Driver	65
	Total	554
Outlook	Import Driver	787
	Export Driver	200
	Track Driver	74
	Total	1061
Common to Presentation and Writer	Import Driver	163
Presentation	Import Driver	374
	Export Driver	241
	Total	615
Writer	Import Driver	242
	Export Driver	197
	Total	439
Grand Total		14746

Table 5.3 : Code line counts for Kernel, Policies, and Drivers for PowerPoint, Word, Outlook email, Internet Explorer, OpenOffice Presentation, and OpenOffice Writer.

local and remote Puppeteer proxies communicate via a fourth PC running the DummyNet network simulator [117]. This setup allows me to emulate various network technologies, by controlling the bandwidth between the local and remote Puppeteer proxies. For each application, I use 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 Puppeteer remote proxy and the data server communicate over a high speed LAN. For experiments that measure the latency of loading the documents using the native application, the client PC communicates directly with the Apache 1.3 or IMAP server over the DummyNet simulator.

For PowerPoint, Word, Presentation, and Writer the data sets consist of subsets of the documents downloaded from the Web and characterized in de Lara et al. [35]. 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 by re-executing the Web traces of Cunha et al. [30]. Finally, Outlook loads synthetic emails, created by adding image attachments of different sizes to a simple text message.

The remainder of this section is organized as follows. First, I present several examples of policies that significantly reduce user-perceived latency. Then I measure the overhead of the implementation.

5.2.1 Some Adaptation Policies

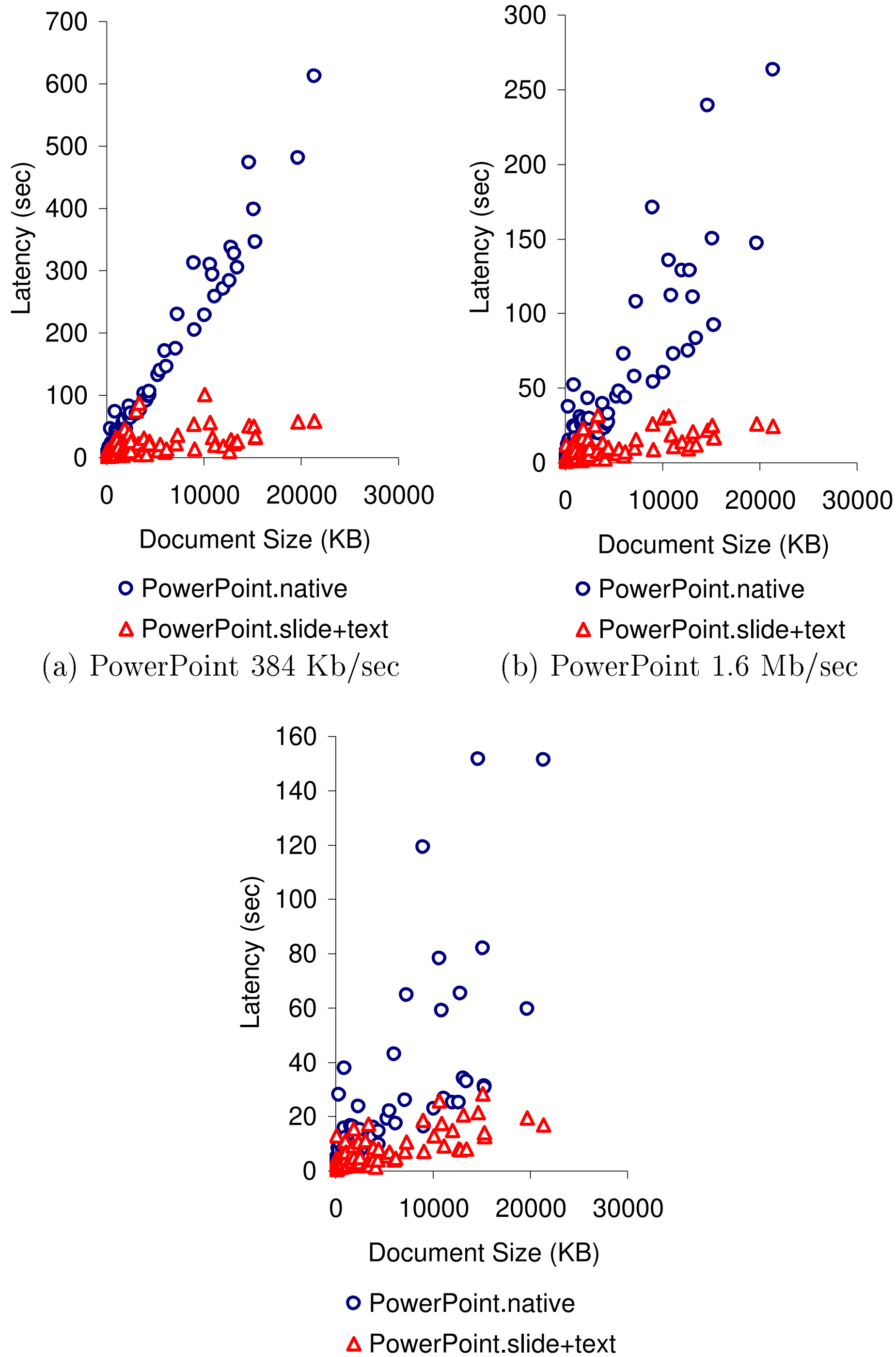
This section illustrates the performance of some sample adaptation policies implemented in Puppeteer. Figures 5.1, 5.2, 5.3, 5.4, 5.5, and 5.6 show the latencies for these sample policies as a function of the total size of the documents. All figures show a common trend. For low bandwidths, the network 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 structure (number of images, embedded objects, pages, etc.), in addition to the size of the document.

PowerPoint and Presentation: Fetch First Slide and Text

This experiment measures the latency for PowerPoint and Presentation 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. Afterward 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.

The results of these experiments appear, under the labels *PowerPoint.slide+text* and *Presentation.slide+text*, respectively, in Figures 5.1 and 5.2 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 (*PowerPoint.native*) or native Presentation (*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 (*PowerPoint.slide+text, Presentation.slide+text*).

I expected that reduced network traffic would improve latency with the slower 384 Kb/sec network. The savings for 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.

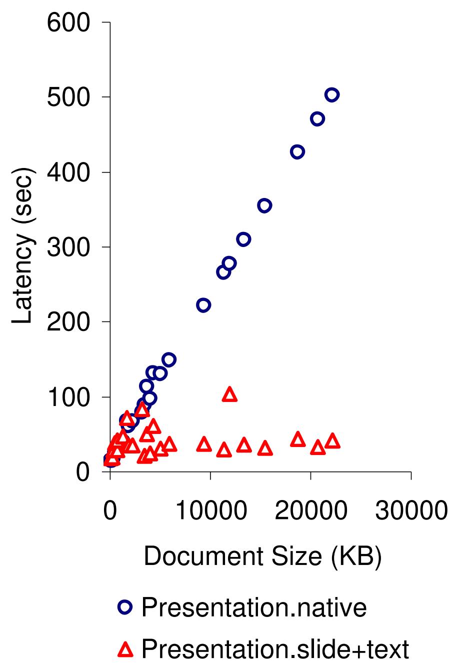


(a) PowerPoint 384 Kb/sec

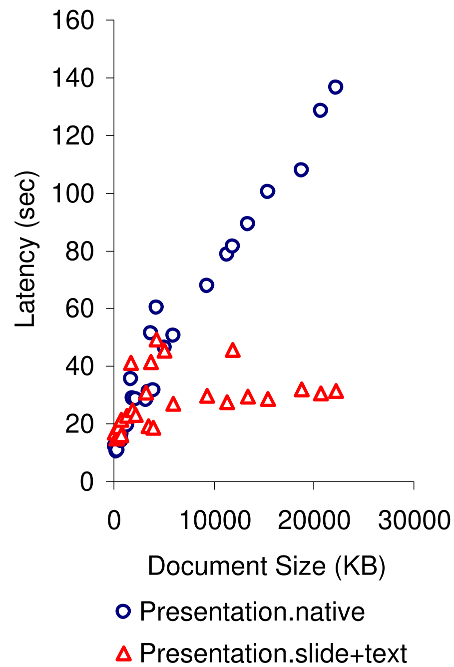
(b) PowerPoint 1.6 Mb/sec

(c) PowerPoint 10 Mb/sec

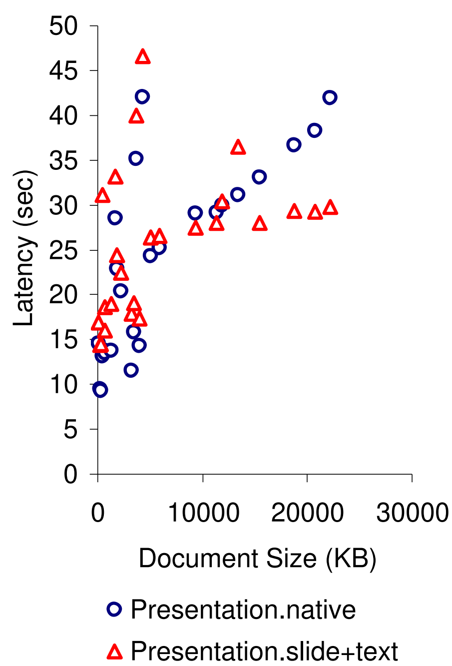
Figure 5.1 : PowerPoint fetch first slide and text. Latency for loading documents with PowerPoint over 384 Kb/sec (a), 1.6 Mb/sec (b), and 1.6 Mb/sec (c). Shown are latencies for native PowerPoint (*PowerPoint.native*) and Puppeteer runs for loading just the components of the first slide and the text of the remaining slides (*PowerPoint.slide+text*).



(a) Presentation at 384 Kb/sec



(b) Presentation 1.6 Mb/sec



(c) Presentation 10 Mb/sec

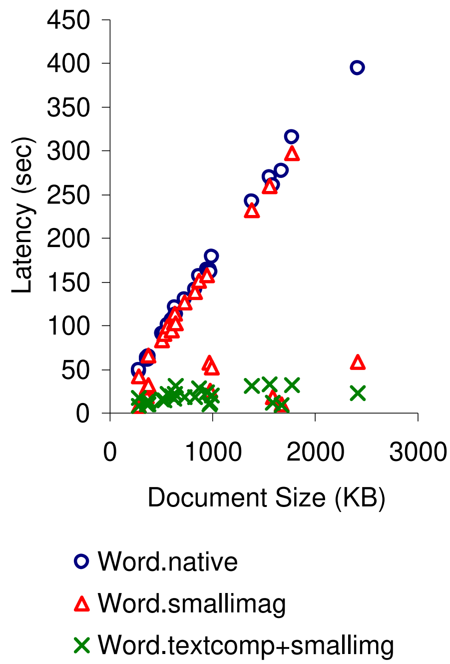
Figure 5.2 : Presentation fetch first slide and text. Latency for loading documents with Presentation over 384 Kb/sec (a), 1.6 Mb/sec (b), and 1.6 Mb/sec (c). Shown are latencies for native Presentation (*Presentation.native*), and Puppeteer runs for loading just the components of the first slide and the text of the remaining slides (*Presentation.slide+text*).

On average, for PowerPoint, *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, *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). These results are consistent with our earlier findings [35] that text accounts for only a small fraction of the total data in large PowerPoint documents. These results suggest that text should be fetched in almost all situations and that the lazy fetching of components is more appropriate for the larger images and embedded objects that appear in the documents.

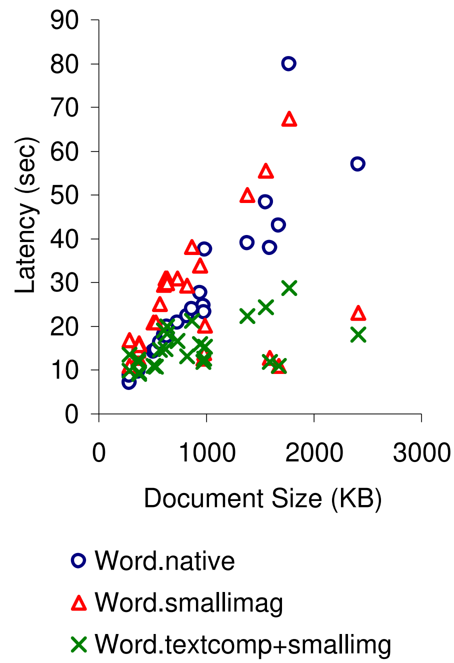
Word and Writer: Text and Small Images

This experiment reduces 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. The experiment also explores the use of text compression to further reduce download latency. Figures 5.3 and 5.4 show the latency for loading the Word and Writer documents over 56 Kb/sec, 384 Kb/sec, and 10 Mb/sec networks. The figures show latencies for native Word and Writer (*Word.native, Writer.native*), and for two Puppeteer configurations 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.comptext+smallimag*, *Writer.comptext+smallimag*).

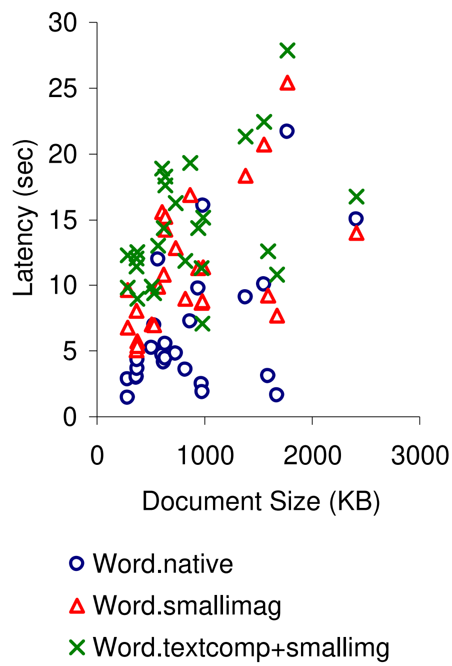
Word.smallimag and *Writer.smallimag* show how loading images smaller than 4 KB reduces latency for about 1/2 of the documents. For these document, *smallimag* achieves an average reduction in latency of 55% and 57% for Word and Writer over



(a) Word 56 Kb/sec

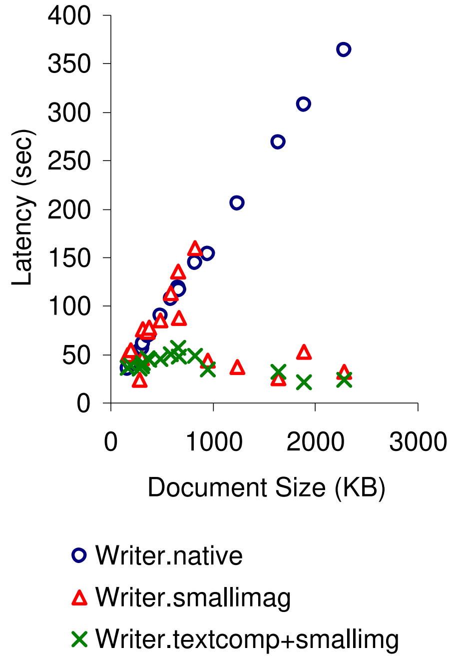


(b) Word 384 Kb/sec

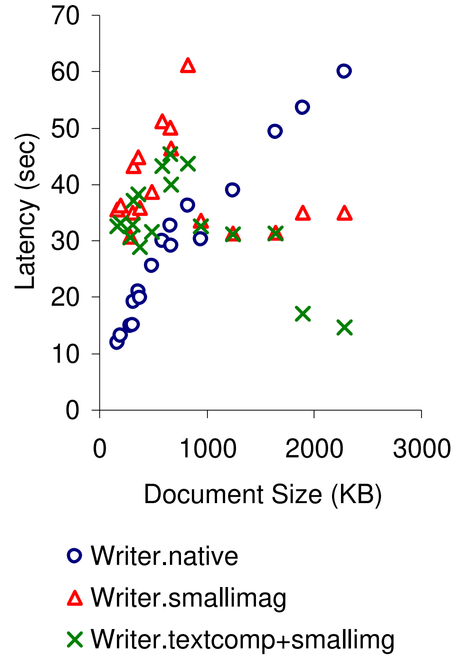


(c) Word 10 Mb/sec

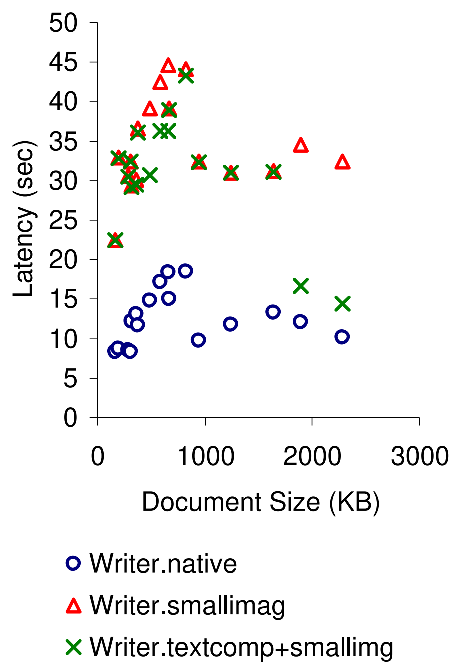
Figure 5.3 : Word text and small images. Latency for loading documents with Word over 56 Kb/sec (a), 384 Kb/sec (b), and 10 Mb/sec (c). Shown are latencies for native Word (*Word.native*), and Puppeteer runs that load text and images smaller than 4 KB (*Word.smallimag*), and load compressed text and images smaller than 4 KB (*Word.comptext+smallimag*).



(a) Writer 56 Kb/sec



(b) Writer 384 Kb/sec



(c) Writer 10 Mb/sec

Figure 5.4 : Writer text and small images. Latency for loading documents with Writer over 56 Kb/sec (a), 384 Kb/sec (b), and 10 Mb/sec (c). Shown are latencies for native Writer (*Writer.native*), and Puppeteer runs that load text and images smaller than 4 KB (*Writer.smallimag*), and load compressed text and images smaller than 4 KB (*Writer.comptext+smallimag*).

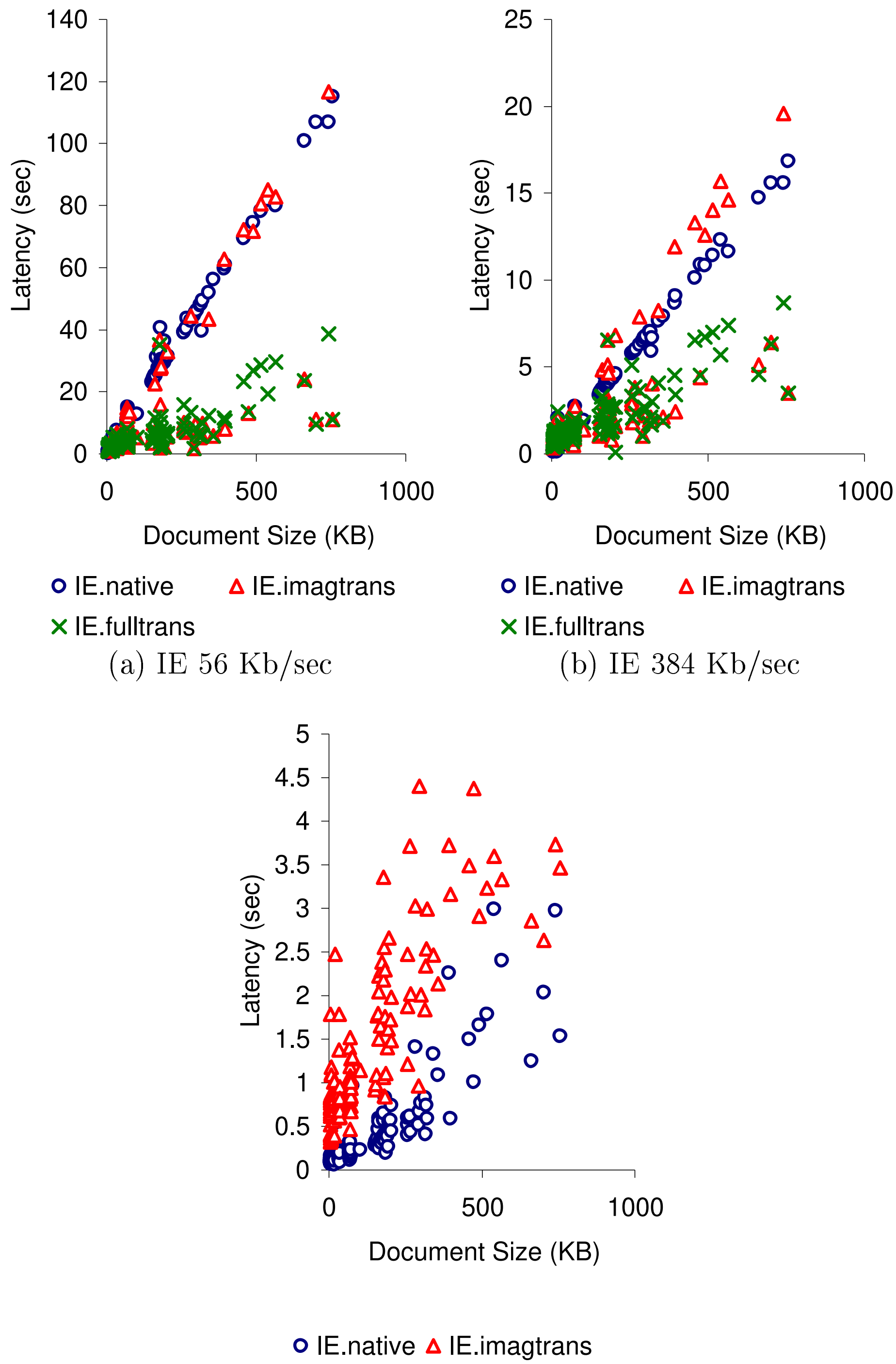
56 Kb/sec respectively. *Word.comptext+smallimag* and *Writer.comptext+smallimag* further reduces latency for all documents. On average, *comptext+smallimag* achieves a reduction in latency of 85% and 59% for all Word and Writer documents over 56 Kb/sec respectively, and of 61% and 50% for Word and Writer documents larger than 1 MB over 384 Kb/sec, respectively. The *comptext+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. Figures 5.3(c) and 5.4(c) show that for most documents on 10 Mb/sec networks text compression is detrimental to performance.

Outlook and IE: JPEG and Text Compression

This experiment explores the use of progressive JPEG compression technology to reduce user-perceived latency for HTML pages and emails. The experiment's 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.

The prototype converts, at run time, GIF and JPEG images embedded in the HTML documents or emails into a progressive JPEG format* using the PBMPlus [113] and Independent JPEG Group [1] libraries. The prototype then transfers only the first 1/7th of the resulting image's bytes. In the client the prototype converts 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.

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

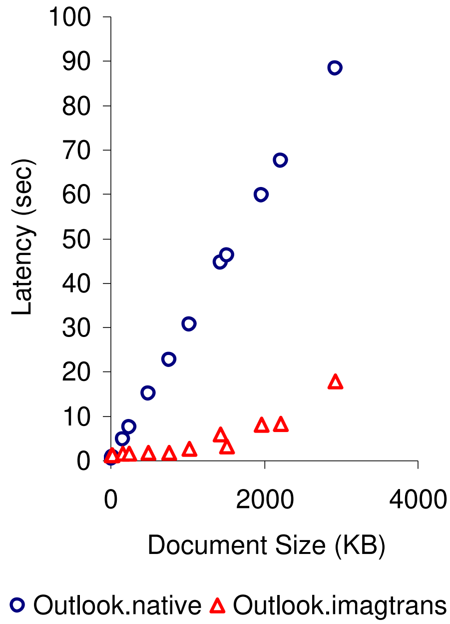


(a) IE 56 Kb/sec

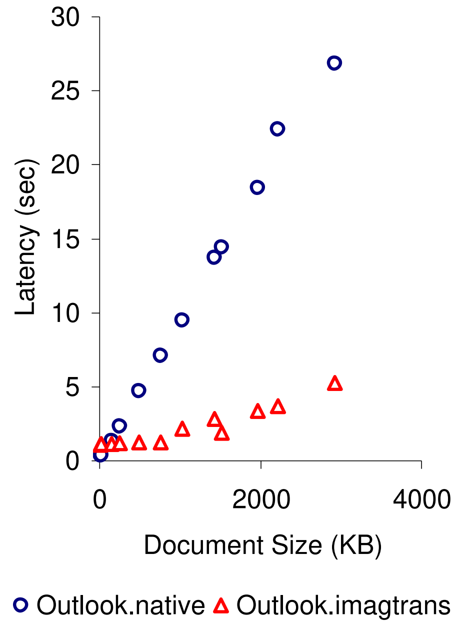
(b) IE 384 Kb/sec

(c) IE 10 Mb/sec

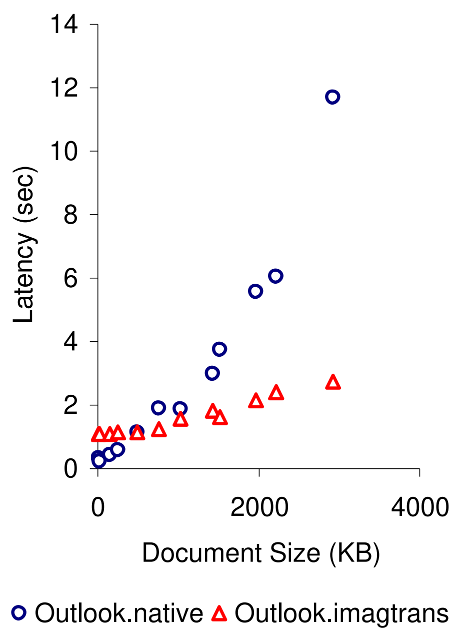
Figure 5.5 : IE JPEG and text compression. Latency for loading document with IE over 56 Kb/sec (a), 384 Kb/sec (b), and 10 Mb/sec (c). Shown are latencies for native IE (*IE.native*), and Puppeteer runs that load only the first 1/7 bytes of transcoded images and load transcoded images and gzip-compressed text(*IE.imagtrans*,*IE.fulltrans*).



(a) Outlook 384 Kb/sec



(b) Outlook 1.6 Mb/sec



(c) Outlook 10 Mb/sec

Figure 5.6 : Outlook JPEG and text compression. Latency for loading emails with Outlook over 384 Kb/sec (a), 1.6 Mb/sec (b), and 10 Mb/sec (c). Shown are latencies for native Outlook (*Outlook.native*), and Puppeteer runs that load only the first 1/7 bytes of transcoded images (*Outlook.imagtrans*).

Figures 5.5 and 5.6 show the latency for loading the HTML documents over 56 Kb/sec, 384 Kb/sec, and 10 Mb/sec networks and the email 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 only the first 1/7 bytes of 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*). The figures do not show Outlook runs with gzip-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 2/3 of the documents see some latency reduction. The remaining 1/3 of the documents, those seeing little improvement from transcoding, are composed mostly of HTML text and have little or no image content. To reduce the latency of these documents, the prototype was extended with gzip text compression. 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.

5.2.2 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.

Initial Adaptation Costs

To determine the one-time initial costs, I compare the latency of loading PowerPoint, Word, HTML, email, Presentation, and Writer documents in their entirety using the native application (*PowerPoint.native, Word.native, IE.native, Outlook.native, Presentation.native, Writer.native*) and the application with Puppeteer support (*PowerPoint.full, Word.full, IE.full, Outlook.full, Presentation.full, Writer.full*). In the latter configuration, Puppeteer loads the document's skeleton and all its components at their highest fidelity, using a separate control message for every loaded component. 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.

Figures 5.7 and 5.8 show the extra latency and Figure 5.9 shows extra data sent for PowerPoint.full, Word.full, IE.full, Outlook.full, Presentation.full, and Writer.full. Extra latency and data are normalized by the latencies and data traffic for loading the documents with the native applications. Overall, for all these applications, the Puppeteer extra latency 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 extra latency is small compared to the total document loading time. For example, the overhead for PowerPoint documents varies from 2% for large documents over 384 Kb/sec to 57% for small documents over 10 Mb/sec and for IE documents from 4.7% for large documents over 56 Kb/sec. to 305% for small document over 10 Mb/sec.

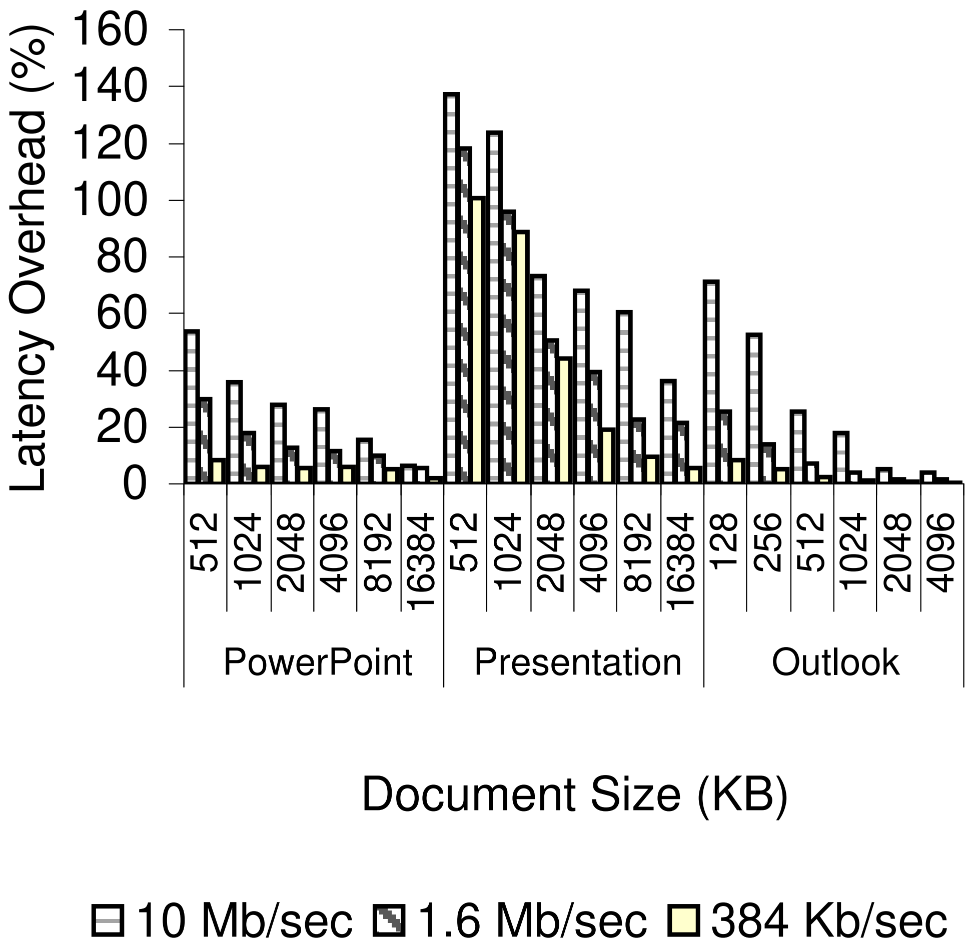


Figure 5.7 : Initial adaptation costs. Latency overhead for loading documents and emails with Puppeteer for PowerPoint, Presentation, and Outlook.

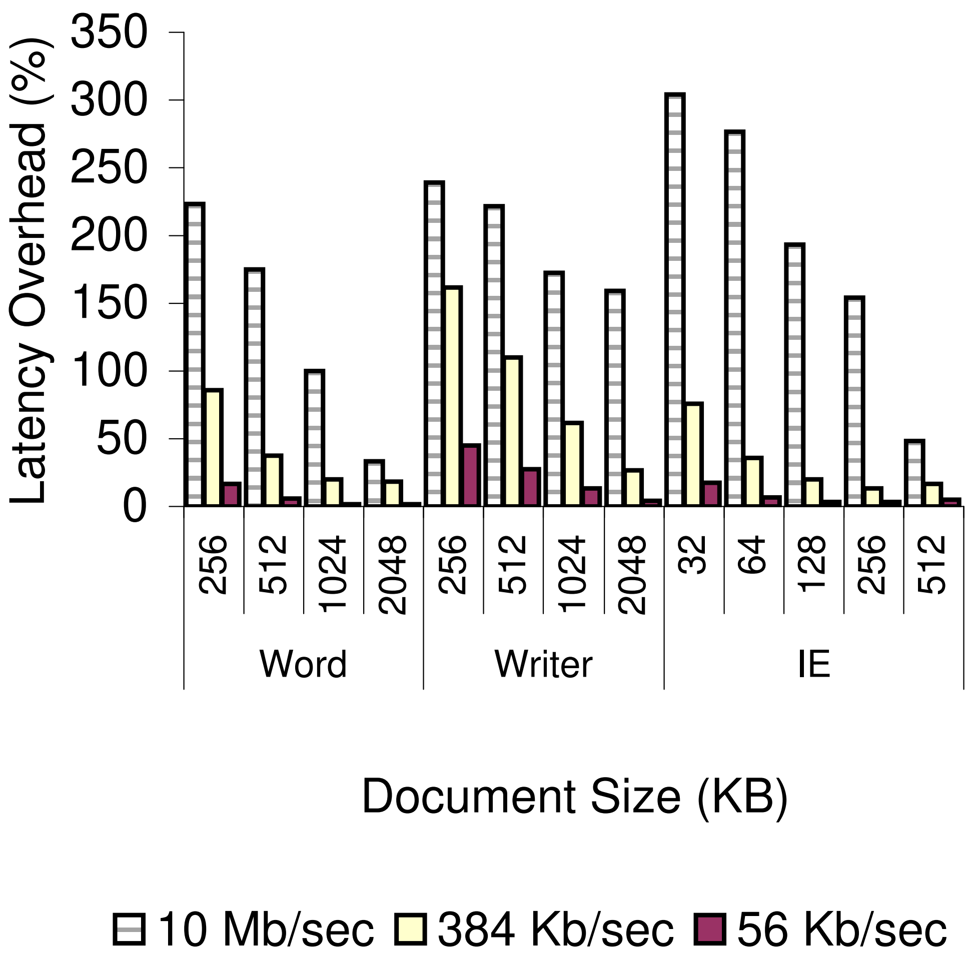


Figure 5.8 : Initial adaptation costs. Latency overhead for loading documents with Puppeteer for Word, Writer, and IE.

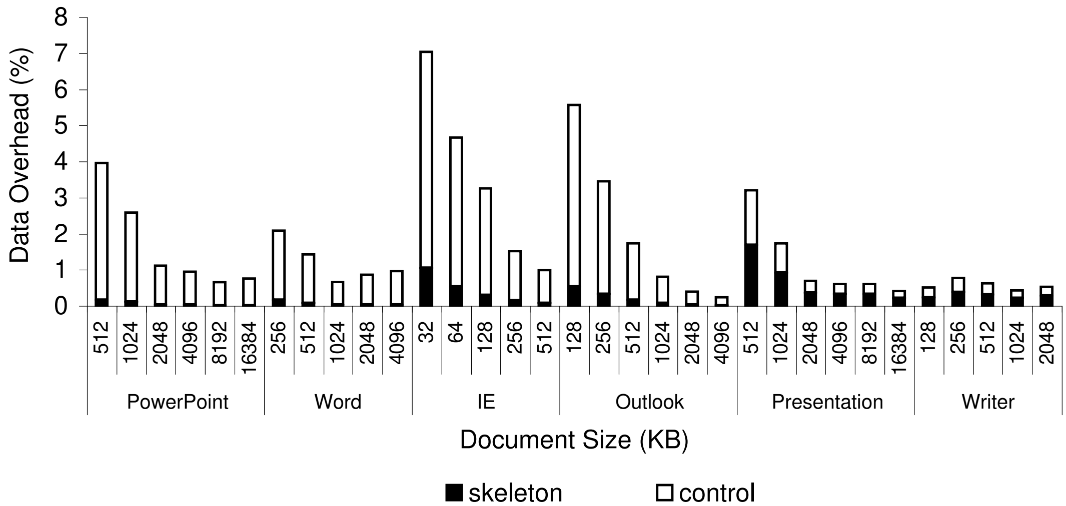


Figure 5.9 : Initial adaptation costs. Data overhead for loading documents and emails with Puppeteer for PowerPoint, Word, IE, Outlook, Presentation, and Writer.

Figure 5.9 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 Figures 5.7 and 5.8. The Puppeteer data overhead becomes less significant as document size increases. For example, the data overhead varies for PowerPoint documents from 2.9% on large documents to 34% on small documents and for HTML documents from 1.3% on large documents to 25% on small documents.

Continuing Adaptation Costs

The continuing costs of adaptation using the exported APIs are clearly dependent on the application and the policy. The purpose of this section 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.

This section presents results for 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 the measurements, the prototype makes sure that the data is present locally at the client before loading it into the application.

For PowerPoint I find that the average time to load a single slide in a presentation is 894 milliseconds with an 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 an 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 instance, 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.

5.3 Limitations

While it is possible 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 I encountered limitations in these applications, which prevented or limited my ability to adapt. Some limitations truly prevented me from performing certain adaptations I wanted to do. Others simply made it more difficult to add the desired adaptive support.

The most restrictive limitations I found were limits on the ability to recognize meaningful subsets and dependencies between different meaningful 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, then 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 the adaptation system 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 made the implementation of some adaptations for some applications more cumbersome than needed. Ideally, I would like to create or update any subset of the document directly from its persistent representation. For example, I 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. Instead, for instance in PowerPoint, a new temporary document needs to be created, containing the new slide or image, which 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 slides in the presentation (e.g., logos, background color, or font size). For this reason, all our PowerPoint policies, even those that load just 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, I was nonetheless able to implement a large number of adaptations for the different applications without much complexity. Part of our ongoing work is to define a comprehensive adaptation API that overcomes the above limitations.

5.4 Related Work

Much work has gone into extending the system architecture to better support mobile clients [80] and to creating programming models that incorporate adaptation into the design of the application [71, 72]. Transcoding has been used for some time now to customize information to the user's preferences [17, 34], reduce server load [4, 20], or support thin clients [53, 145, 146]. The project that most closely relates to Puppeteer is Odyssey [103], which splits the responsibility for adaptation between the application and the system. Puppeteer takes a similar approach, pushing common adaptation tasks into the system infrastructure and leaving the application-specific aspect of adaptation to application drivers. The main difference between the two systems lies in Puppeteer's use of existing run-time interfaces to adapt existing applications, whereas Odyssey requires applications to be modified to work with it.

Visual Proxies [123], an offspring of Odyssey, 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 to implement them.

The term "component adaptation" [47, 70, 82] is also used to describe efforts that support tunable RPC implementations, which monitor and adapt the communication path between distributed CORBA or DCOM components. These approaches enable client components to control their bindings to servers components. Instead of getting a default static stub, clients components can choose among available bindings, or adapt a binding by, for instance, 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.

Another project that uses similar ideas to Puppeteer is Dynamic Documents [75]. This instrumentation of the Mosaic Web browser uses Tcl scripts to set the policies for individual HTML documents. While Puppeteer uses the external interfaces provided by the application, Dynamic Documents use an internal script interpreter in the browser.

The Pebbles [100] project uses Automation interfaces to explore the use of Personal Digital Assistants (PDAs) to augment the functionality of applications.

Quint et. al [114] describes a structured editor that provides an API that enables extending the editor with new functionality. The API supports commands for navigating and modifying the three-like document structure, and for displaying these modifications on the user interface. In its current form, however, the editor's API can only be invoked within the same process (i.e., like a macro).

5.5 Summary and Conclusions

This chapter evaluated the extent to which existing APIs can be used for the purposes of adaptation. In particular, the chapter describes the implementation of a number of adaptation policies for popular applications from the Microsoft Office and the OpenOffice productivity suites and for Internet Explorer. Although I found some limitations in their APIs, I was able to implement a large number of adaptations policies without much complexity and with little overhead.

Chapter 6

Multi-Application Adaptation

Previous approaches to adaptation do not support system-wide bandwidth adaptation policies, which adapt multiple applications in concert. This chapter describes how, by extending component-based adaptation with dynamic control over bandwidth scheduling, it is possible to adapt multiple applications in a coordinated fashion. Some sample system-wide bandwidth adaptation policies include: prioritizing the transmission of text over image data across all applications; shifting most of the available bandwidth to the document that is currently in the foreground, while reducing the fidelity of the background documents; guaranteeing that all images across all applications are available at the same fidelity level before fetching higher fidelity refinements; and guaranteeing a minimal bandwidth allocation to an audio application independent of network bandwidth fluctuations, by dynamically adjusting the fidelity of the data downloaded by other applications running on the device.

Clearly, coordinated adaptation of multiple applications requires additional functionality beyond what is required to adapt individual applications. While an adaptation policy that adapts a single application can be fully contained within the application itself, adapting multiple applications in concert requires system-wide adaptation policies that run in a *centralized* location (i.e., outside any specific application). Implementing powerful adaptation policies, like the examples given above, requires understanding the semantics of data, controlling the behavior of the applications, and monitoring the user activity (e.g., determine what application is on the foreground). Doing so outside the application, requires that the application expose an API (Application Programming Interface) and a document format, through which the adaptation policies can be implemented. Chapters 3 and 5 introduced *component-based adaptation* and demonstrated that component-based adaptation is an effective technique that supports powerful adaptation policies without requiring modifications to the applications. This chapter demonstrates that system-wide adaptation policies can be implemented by extending component-based adaptation.

In addition to understanding the behavior and the data of the applications, system-wide adaptation policies also require coordinating between the adaptation policies for different applications, documents, and document components. In particular, dynamic control over *transmission scheduling* is a core requirement.

While for single-application adaptation, it suffices to specify what data to transmit and how this data is to be adapted, system-wide adaptation policies must be able to specify, for all applications and documents running on the device, *in what order* to transmit what data and how this data is adapted. If, for example, we want to implement an adaptation policy that specified "text first for all applications", then we need to be able to schedule the transmission of the text of all applications before any other document elements are transmitted.

This chapter presents the design, implementations, and evaluation of our Hierarchical Adaptive Transmission Scheduler (HATS) [37], which extends Puppeteer with fine grain control over the transmission of component data. HATS enables Puppeteer to enforce the network allocations determined by the adaptation policies, to react quickly to changes in network connectivity and user behavior (e.g., by shifting bandwidth to the higher priority components), and to implement sophisticated transmission policies that give different priorities to the hierarchy of applications, documents, and components running on the device. HATS even supports giving different priorities to various parts of a component's data. For example, fetching the first half of a component's data can have higher priority that fetching the rest of the data.

While earlier systems have implemented individual facilities described here [7, 53, 71, 82, 90, 103, 130, 123, 135], to our knowledge, no prior system supports scheduling and adaptation working in concert. HATS is thus more powerful than its predecessors. HATS enables the implementation of new powerful system-wide adaptation policies that allow bandwidth allocation where it will most likely minimize the time users wait for an application to complete its network operations.

The rest of this chapter is organized as follows. Section 6.1 presents our implementation of HATS. Section 6.2 evaluates the performance of three system-wide adaptation policies. Section 6.3 describes how HATS differs from previous approaches to resource scheduling. Finally, Section 6.4 summarizes and concludes the chapter.

6.1 HATS

This section first discusses possible implementations of HATS on Puppeteer. It then describes a proof-of-concept remote proxy-based HATS implementation.

6.1.1 Possible HATS Implementations

In Puppeteer, all communication between the applications on the mobile or wireless client and the data server(s) passes through the local and remote proxies. These proxies are therefore the natural places for implementing HATS. The rest of this section describes two possible implementations and the tradeoffs between them.

A local proxy-based HATS implementation follows most closely the conventional HTTP pull-based data transfer model: each component is specifically requested by the local proxy (based on the ordering determined by the system-wide adaptation policies), and the remote proxy simply fulfills each request as it arrives. To enable the concurrent transmission of multiple components, the local proxy requests only a small portion of the data of a given component at any given time. I refer to the amount of data requested at one time for a specific component as the *scheduling quantum*. The local proxy may, of course, request data for multiple components in a single batch message.

A remote proxy-based HATS implementation follows a different approach, using a push-based model of data transfer. The system-wide adaptation policies still run on the local proxy, but determine the order in which components get transferred by choosing a transmission strategy that orders the transmission of component data. In the most general case, this would require sending mobile code that implements specific transmission strategies. In practice, I expect that common transmission strategies will be available at the remote proxy, and that the system-wide adaptation policies will just parameterize them.

There are a variety of tradeoffs between the local and remote proxy approaches. On the plus side for the local proxy approach, it closely follows the well-understood pull-based Web transfer model. Additionally, client events, such as the user changing focus to a different window, can be handled locally (i.e., changing the order in which components are requested). On the minus side, the local proxy does not know which components have data ready to be sent on the remote proxy. This information is available at the remote proxy and would need to be transferred to the local proxy, requiring additional communication, in order to ensure that the local proxy would not waste scheduling cycles on components that do not have data ready to transmit. Moreover, the fact that every component must be requested separately requires that a fairly large scheduling quantum be used; smaller quanta require a large number of request packets. A small bandwidth scheduling quantum is desirable for streaming audio and video applications that need a continuing stream of data to avoid jitter.

On the plus side for the remote proxy approach, it does not require a request for every quantum and can, therefore, support a smaller quantum size. On the minus side, a remote proxy approach requires the system-wide policies to reconfigure the remote proxy transmission strategy when relevant user events occur.

Our current HATS implementation follows the remote proxy-based model. I chose the remote proxy-based implementation over the local proxy-based approach because I want to provide for bandwidth sharing with low jitter. While it is widely accepted that low jitter is important for streaming media (which I plan to support in the future), it is less apparent that fine granularity scheduling is also important for more traditional data types (e.g., text and images). Scheduling components that encode traditional media types atomically can seriously restrict the ability of the system to run multiple applications concurrently (which is, after all, the underlying purpose of HATS). Since traditional media types can be large (a few megabytes) [35], they can take a long time to propagate over the bandwidth-limited link. Scheduling large components atomically can therefore starve other applications and documents from network service. Finally, I felt that, for our initial implementation, a fixed number of transmission strategies, provided by the remote proxy, were sufficient. The following section discusses in detail a proof-of-concept remote proxy-based HATS implementation.

6.1.2 Remote-Proxy-Based HATS Implementation

The current HATS implementation is incorporated mainly in the Puppeteer remote proxy, and extends many of the facilities present in Puppeteer. Figure 6.1 shows the architecture of our implementation. It involves four key mechanisms: a *transmission hierarchy, schedulers,* and *multiplexing* and *demultiplexing facilities*. In short, the transmission hierarchy describes the hierarchical structure of the data to be transmitted. Schedulers determine the order in which nodes in the hierarchy are transmitted. The multiplexer and demultiplexer organize the concurrent transmission of data associated with multiple nodes in the hierarchy.

The following sections first describe the above mechanisms. I then discuss how our implementation supports dynamic reconfiguration. I conclude with a simple example that illustrates the use of these mechanisms.

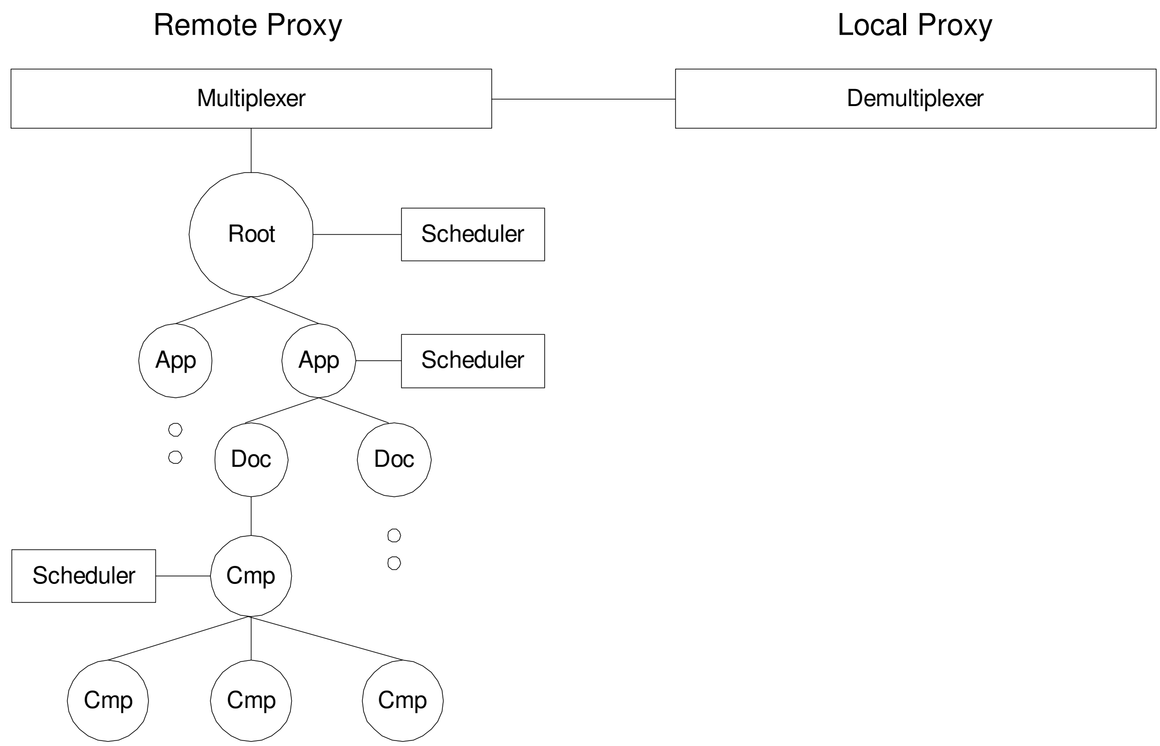


Figure 6.1 : Remote proxy-based HATS implementation. The architecture consist of schedulers, multiplexing and demultiplexing facilities, and a hierarchy of applications (App), documents (Doc), and components (Cmp).

Transmission Hierarchy

The *transmission hierarchy* is a transient structure composed of applications, documents, and components that are in the process of being transmitted. The transmission hierarchy is assembled in response to requests from adaptation policies running on the local proxy. Nodes remain in the transmission hierarchy only as long as data is being transferred to them. Specifically, leafs in the transmission hierarchy are removed as soon as their related data finishes transmission. Interior nodes of the hierarchy are removed when their related data finishes transmission and they do not have any children left with pending data.

Nodes in the transmission hierarchy can be in one of three states: *ready*, *waiting*, and *disabled*. A node in the ready state has data queued for transmission. A node in the waiting state has no data queued-up. When data is added to the node's queue, it transitions to the ready state. A node in the disabled state may have data queued for transmission but has been temporarily disabled as a result of the transmission strategy.

HATS keeps separate transmission hierarchies for each bandwidth-limited device serviced by the remote proxy. Each of the client hierarchies is serviced by a separate thread running on the remote proxy and receives a fair share of the remote proxy's resources. This implementation decision assumes that the client bandwidth is the bottleneck in our system. If, however, a Puppeteer remote proxy was to serve a large set of clients, we might see contention for the remote proxy resources. If so, we can adopt one of the several resource reservation strategies that have been proposed in the literature [7, 27, 90, 135].

Schedulers

Adaptation policies (running on the local proxy) determine the transmission strategy by setting a scheduler for every internal node of the transmission hierarchy. Specifically, HATS requires one scheduler for the root node (the *root scheduler*) to choose what application to serve, one scheduler for every application node (the *application scheduler*) to choose what document to serve, and one scheduler for every component node (the *component scheduler*) that has children.

The HATS scheduler architecture is modular and supports the addition of new schedulers. HATS supports static schedulers that implement simple transmission strategies that choose the next node to serve based on well known scheduling disciplines (e.g., round-robin, in-order, FIFO, packet fair queuing), and dynamic schedulers that reconfigure their scheduling discipline in response to events (e.g., changing the weight of a packet fair queuer after a drop in available bandwidth). Dynamic schedulers may register for events that signal changes in available network bandwidth, changes in the transmission hierarchy, or progress in the transmission of a component's data.

Bandwidth is distributed top-down. A parent scheduler has full control over how it allocates bandwidth to its children. A child scheduler can only distribute bandwidth within the bounds of the allocation provided to it by its parent. Schedulers of parent nodes can re-parameterize the schedulers of their descendents. A parent scheduler can also temporarily disable one of its descendents. Once a node has been disabled, it can only be re-enabled by one of its ancestors that is at least as high in the hierarchy as the scheduler that disabled it. These properties, ensure that transmission strategies implemented by ancestor schedulers take priority over transmission strategies implemented by child schedulers and enable the implementation of transmission strategies that operate across levels of the hierarchy.

For example, it is possible to implement a scheduler that prioritizes the transmission of text data over image data across applications and documents by installing a root scheduler that implements the following algorithm: (1) traverse the hierarchy to determine if there are any nodes with associated text and image data (since the root scheduler sits at the root of the hierarchy, it has access to all the nodes); (2) if there are any nodes with associated text data, disable all nodes with associated image data;

(3) monitor the transmissions of the nodes with associated text data and respond to completion events generated by these nodes; (4) once all nodes with text data have been transmitted, re-enable nodes with associated image data; and (5) restart this process when a node with associated text data is added to the hierarchy. This root scheduler implements the above system-wide transmission strategy independent of the specific schedulers running at the application and component levels of the hierarchy.

Multiplexing and Demultiplexing

The multiplexer and demultiplexer perform the actual transmission of component data. The multiplexer and demultiplexer operate in terms of a scheduling quantum. The size of the quantum, which determines the maximum amount of data of a single node of the hierarchy that is transferred at a time, can be configured by the system-wide adaptation policies to reflect changes in connectivity between the remote and local proxies, the types of the nodes being fetched, or any other criterion.

Our prototype implementation multiplexes all component data going to the same client over a single TCP connection. An important benefit of this approach, in addition to its simplicity, is that it avoids any issues stemming from TCP's slow-start or congestion control algorithms. TCP only sees one long-running and stable network connection, for which it tends to perform much better than short-lived "transactional" sessions. The downside of using TCP is that the frequency with which the prototype can reallocate bandwidth is limited by the OS transmission buffers, which the prototype must keep full to get good performance, and by TCP's own ability to respond to changes in the available bandwidth. Measurements of reconfiguration latencies are present in Section 6.2.1.

Dynamic Reconfiguration

HATS supports dynamic reconfiguration of the transmission hierarchy by either explicit requests for reconfiguration from adaptation policies running on the local proxy or by the schedulers running on the remote proxy.

On the local proxy, adaptation policies use tracking drivers to monitor the user, and reallocate resources to benefit the application, document, or components currently in use. Adaptation policies reconfigure the transmission hierarchy by sending requests, which replace or re-parameterize the schedulers attached to the transmission hierarchy.

On the remote proxy, schedulers can register for events and reconfigure the transmission hierarchy when they occur. For example, a scheduler can implement a transmission strategy, which ensures that all Progressive JPEG images in an HTML page are present with a given fidelity level before higher fidelity data is transferred. Initially, all Progressive JPEG objects are scheduled with equal priority. Once smaller images reach a given threshold, for example 30% of the file has been transmitted, the scheduler reconfigures the hierarchy ensuring that larger images that have not yet reached the 30% mark now have transmission priority over the smaller images.

Example

To illustrate the HATS mechanisms let us consider a user simultaneously loading two Web pages in two browsers as depicted in Figure 6.2. Let us consider further that one of the pages describes a resort that the user is considering for an upcoming vacation, while the other page comes from a major news site.

Obviously, our user values the images depicting the idyllic resort more than the images (mostly advertisements) that appear on the news site. Accordingly, our user would like to make significantly more bandwidth available to loading the resort images than to loading the ads. The user, however, would like to feel that both browsers are making similar progress with new images appearing in both browsers at similar rates, even if this requires that images from the news page be loaded initially at low fidelity and refined later as bandwidth becomes available.

To achieve the above goals, I use a system-wide adaptation policy that installs

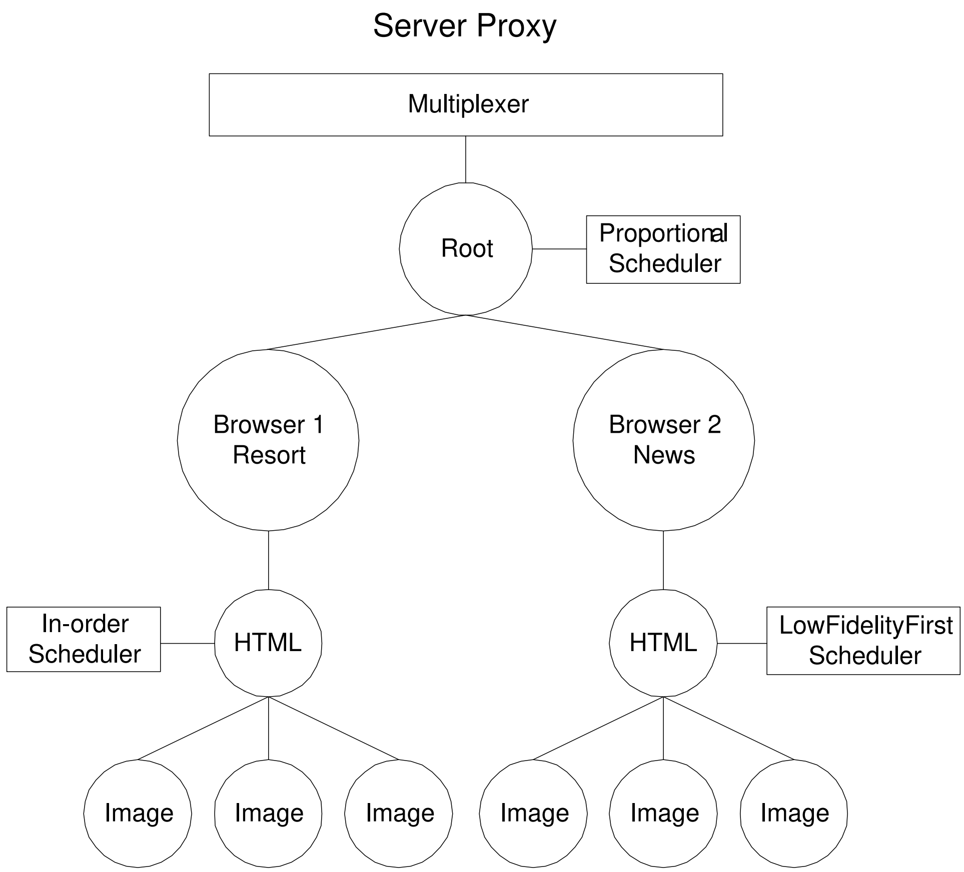


Figure 6.2 : Transmission hierarchy for loading to web pages.

a root-level proportional scheduler that gives 80% of the available bandwidth to the browser on the foreground (i.e., loading resort page) and the remaining 20% to the browser on the background (i.e., loading the news page).

Since it has enough bandwidth, the policy for the browser fetching the resort page decides to fetch all images at high fidelity and sets a component-level scheduler that transmits the image components in-order.

On the other hand, the policy for the browser fetching the news page converts images into a progressive representation (e.g., progressive JPEG). This policy assumes that images have 3 fidelity levels, which I will refer to as *low*, *medium*, and *high*. To make sure that all images in the news page are present with low fidelity before any image is refined, the policy sets the *LowFidelityFirst* dynamic scheduler as the component-level scheduler. LowFidelityFirst gives document-wide priority to low fidelity images, transmitting images sequentially. When an image reaches a target fidelity level, LowFidelityFirst start serving the next image in the page. When all images reach low fidelity, LowFidelityFirst starts sending medium fidelity data for the first image and so on.

Later, when an article catches the user's attention, the user can reverse the current policy by moving the browser fetching the news page to the foreground. The system-wide adaptation policy, traps the user event and reconfigures the proportional scheduler, shifting the majority of the bandwidth to the browser loading news pages. The system-wide policy then switches the news browser to fetching full fidelity images and the resort browser to fetching progressive images.

6.2 Experimental Evaluation

This section quantifies the advantages and overheads of our Puppeteer-based HATS implementations by measuring the effects of three HATS-enabled adaptation policies. The set of possible adaptation policies is, however, much larger. The purpose of this section is not to determine the best adaptation policies for the applications we adapt.

Instead, this section illustrates the variety of system-wide adaptation policies made possible by HATS.

This section quantifies the performance of the Puppeteer-HATS prototype by loading documents using Internet Explorer 5.5 (IE5) and PowerPoint (PPT) on a platform consisting of three Pentium III 500 MHz PCs running Windows 2000. These PC are configured as: a data server running Apache 1.3, which stores all the documents used in these experiments; a Puppeteer remote proxy; and a client that runs the user's applications and the Puppeteer local proxy. The local and remote Puppeteer proxies communicate via another PC running the DummyNet network simulator [117]. This setup allows me to emulate various network technologies, by controlling the bandwidth between the local and remote Puppeteer proxies. The Puppeteer remote proxy and the data server communicate over a high speed LAN.

6.2.1 HATS-based Adaptation Policies

This section explores the power of HATS by implementing adaptation policies that benefit from dynamic bandwidth reconfiguration. I first explore the power of dynamic schedulers running on the remote proxy that prioritize the transmission of text and low fidelity images. Then, I experiment with an adaptation policy running on the local proxy that re-allocates bandwidth to the document that has the user's focus.

I run these experiments using a 256-byte HATS scheduling quantum. I choose this small quantum size to stress the performance of HATS: If the implementation performs well with the small quantum size, we expect its performance would only improve with a larger quantum size. I examine the overhead incurred by the small quantum size in Section 6.2.2.

Remote-proxy-based Reconfiguration

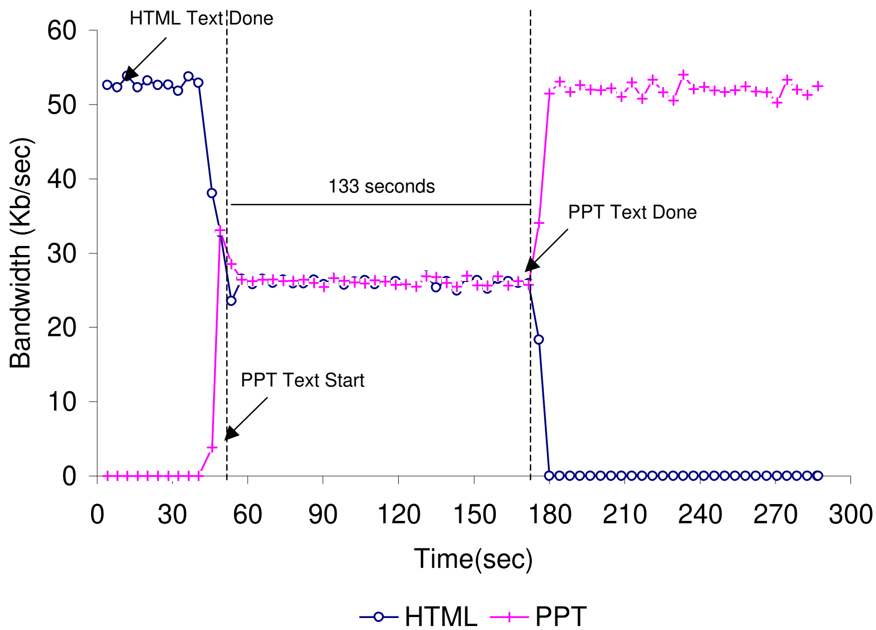
This section compares the performance of two system-wide adaptation policies running on Puppeteer with and without HATS support. Without HATS, Puppeteer assigns equal allocations to all applications and documents, and transfers components within a document in-order.

Text First This experiment compares the time to load text-only versions of a HTML page and a PPT presentation. For each application I use an adaptation policy that loads a document into the application as soon as all its text components are present at the local proxy (the HTML for IE, and the text of all slides for PPT), and displays images and other embedded components as they become available at the local proxy.

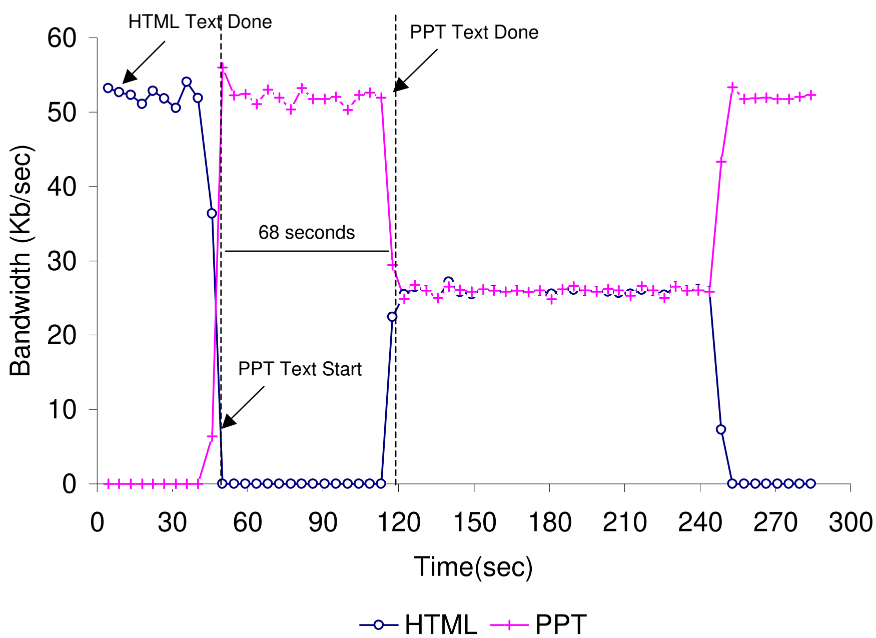
For Puppeteer with HATS support (Puppeteer-HATS), I also use a system-wide adaptation policy that sets the *TextFirst* dynamic scheduler as the root scheduler in the transmission hierarchy. TextFirst gives system-wide priority to the transmission of text components (HTML, PPT slides).

Figures 6.3(a) and (b) show the bandwidth allocations for loading an image-rich 668 KB HTML document and a 1.05 MB PowerPoint presentation over a 56 Kb/sec network link with Puppeteer and Puppeteer-HATS. For these documents, HATS achieves savings of 49%, reducing the time to load a text-only version of the PPT presentation from 133 to 68 seconds. The arrows in the figures show the start and finish times for loading the text components of the HTML and PPT presentation. HATS lowers the times to load a text-only version of the PPT presentation by re-allocating bandwidth from sending images embedded in the HTML document to sending PPT slides.

Low Fidelity First In this experiment, I use two browsers to load image-rich Web pages. Both browsers use an adaptation policy that loads HTML pages progressively. The adaptation policy, running on the local proxy, instructs the remote proxy to convert images embedded in the HTML pages into a progressive JPEG representation and transfer them to the local proxy. I assume that images have 3 fidelity levels, which I will refer to as *low*, *medium*, and *high*. I chose these levels to correspond



(a) Puppeteer



(b) Puppeteer-HATS

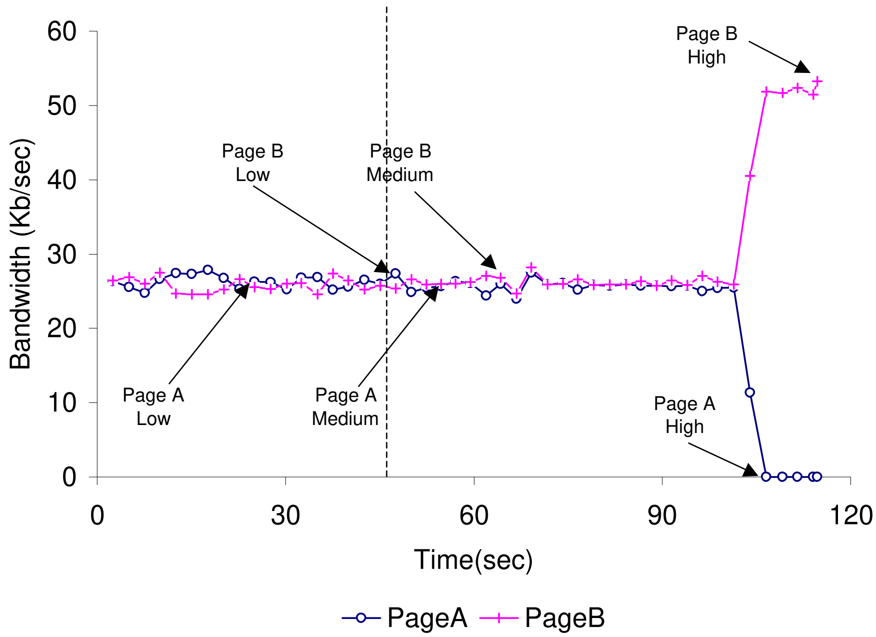
Figure 6.3 : Bandwidth allocations for transmitting a HTML document and a PowerPoint (PPT) presentation using (a) Puppeteer and (b) Puppeteer-HATS with *TextFirst*, a dynamic scheduler that prioritizes the transmission of text components. Puppeteer-HATS reduces the time to load a text-only version of a PPT presentation by 49%, from 133 to 68 seconds.

to the first 1/7, the next 2/7, and the last 4/7 of the bytes in the progressive JPEG representation. The adaptation policy then responds to events on the local proxy that signal progress in the transmission of the images' data. When the image data available at the client reaches predefined fidelity levels, the adaptation policy transcodes the image back to its original format (GIF, JPEG, etc.) and uses IE5's exported API to update the image in the browser.

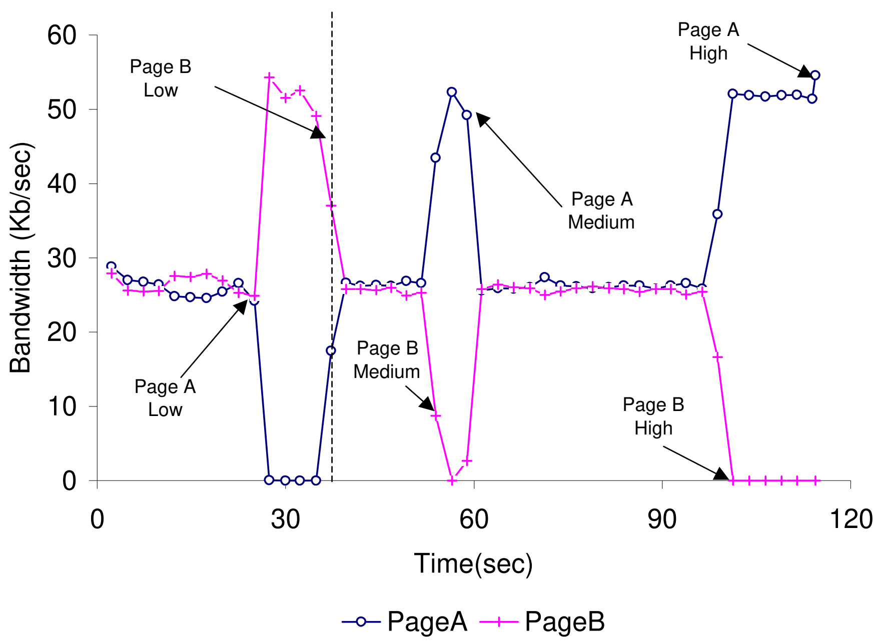
For Puppeteer-HATS I also use a system-wide adaptation policy that sets the *LowFidelityFirst* dynamic scheduler as the root scheduler in the transmission hierarchy. LowFidelityFirst gives application-wide and document-wide priority to low fidelity images. LowFidelityFirst ensures that all images, of both documents, have reached the same fidelity level before starting to transmit higher fidelity data.

I experimented with the order in which images within a page get serviced by using a round-robin and an in-order scheduler. With round-robin, all images are transferred in parallel. As smaller images reach a target fidelity level, their share of bandwidth is redistributed among the larger images that still have data to be sent. With in-order, the prototype transmits images sequentially. When an image reaches a target fidelity level, the prototype start serving the next image in the page.

I was expecting round-robin to be the better transmission strategy as small images would show quickly irrespective of their location on the page. In practice, however, our experience showed that many images have similar sizes and that servicing them all in parallel results in a situation where users do not see any progress for an extended amount of time, followed by brief periods of high activity where most images are updated simultaneously (the implementation only triggers an application update when it reaches a targeted fidelity level). Moreover, because most images reach the target fidelity level around the same time, there is little opportunity for overlapping transcoding with network transmission. For these reasons, I choose to present my results using an in-order transmission strategy, which achieves better overall performance.



(a) Puppeteer



(b) Puppeteer-HATS

Figure 6.4 : Bandwidth allocations for transmitting two HTML document with (a) Puppeteer and (b) Puppeteer-HATS with *LowFidelityFirst*, a dynamic scheduler that prioritizes the transmission of low fidelity images. Puppeteer-HATS reduces the time to load a low fidelity version of *PageB* from 46 to 35 seconds.

Figures 6.4(a) and (b) show the results of this experiment for loading two large HTML documents (*PageA* and *PageB*), with sizes 388 KB and 304 KB, over a 56 Kb/sec network link, with Puppeteer and Puppeteer-HATS. For these documents, Puppeteer-HATS reduces the time to load a low fidelity version of *PageB* from 46 to 35 seconds. The arrows in the figures show the times, in each run, at which the pages reach the three fidelity levels. With Puppeteer-HATS, when *PageA* reaches low fidelity, the LowFidelityFirst dynamic scheduler reconfigures the transmission hierarchy, transferring network resources to *PageB*, and lowering the time it takes *PageB* to reach low fidelity. In contrast, Figure 6.4(b) shows that with Puppeteer without HATS, the remote proxy keeps sending data for *PageA* even after it reaches low fidelity, limiting the bandwidth available to *PageB* and increasing the time it takes for *PageB* to reach low fidelity.

Figure 6.4(b) also shows that with the LowFidelityFirst dynamic scheduler *PageB* reaches medium and high fidelity before *PageA*; the precise opposite of what happens in Figure 6.4(a). The inversion of times to reach medium and high fidelity are an artifact of the adaptation policy and the pages used in this experiment. The LowFidelityFirst adaptation policy only transcodes images to progressive JPEG if they are bigger than 4KB in size. In its first state, LowFidelityFirst loads all images smaller than 4 KB and the first 1/7 of all transcoded images. In this experiment *PageB* has more images smaller than 4 KB, which are not transcoded and are sent in whole, than *PageA*. This is why it takes *PageB* more time to reach low fidelity, but less time to reach medium and high fidelity.

Local-proxy-based Reconfiguration

This experiment evaluates *Focus*, an adaptation policy that reconfigures the transmission strategy (by re-parameterizing the root scheduler of the transmission hierarchy) so that it re-allocates bandwidth to the components of the foreground application or document. This experiment loads two different HTML pages simultaneously in two separate instances of IE5. The experiment uses a system-wide adaptation policy that switches the browser in the background to fetching low fidelity images. This adaptation policy achieves small page download times, albeit at low fidelity, even when the browser in the background receives just a small fraction of the available bandwidth. Once the page is loaded in low fidelity, Puppeteer starts fetching higher fidelity refinements and uses IE5's exported API to upgrade the images in the browser.

Focus, which runs on the local proxy, uses explicit requests to reconfigure the transmission strategy and give the majority of the bandwidth (80%) to the browser of the foreground. This experiment uses a scheduler that implements the WF^2Q+ Packet Fair Queuing (PFQ) algorithm [19], which distributes bandwidth among the nodes it services according to their rate allocations.

Figure 6.5 shows the results of this experiment for loading two image-rich HTML pages (*PageA* and *PageB*), with sizes 668 KB and 193 KB, on two separate IE5 windows over a 384 Kb/sec network link. The transmission strategy starts by giving equal bandwidth to both documents. Seconds later, *Focus* detects that the user has selected the browser loading *PageB* (moving it to the foreground), and reconfigures the transmission strategy giving *PageB* 80% of the link bandwidth. Later, when the user selects the browser loading *PageA*, *Focus* reconfigures the transmission strategy again.

I measured the time it takes for a bandwidth re-allocation to take effect, from the time it is requested by the local proxy until the bandwidth allocation for the documents stabilizes again. I refer to this time as re-allocation latency. I measured an average re-allocation latency of 1.38 and 0.24 seconds for runs on 56 Kb/sec and 384 Kb/sec network links, respectively. The re-allocation latency is dominated primarily by the time it takes the system to send any packets that were already in the network transmission queue by the time the remote proxy processed the re-allocation request. We experimented with increasing the network latency in DummyNet. Unsurprisingly, the re-allocation latency increased by exactly the amount of extra latency

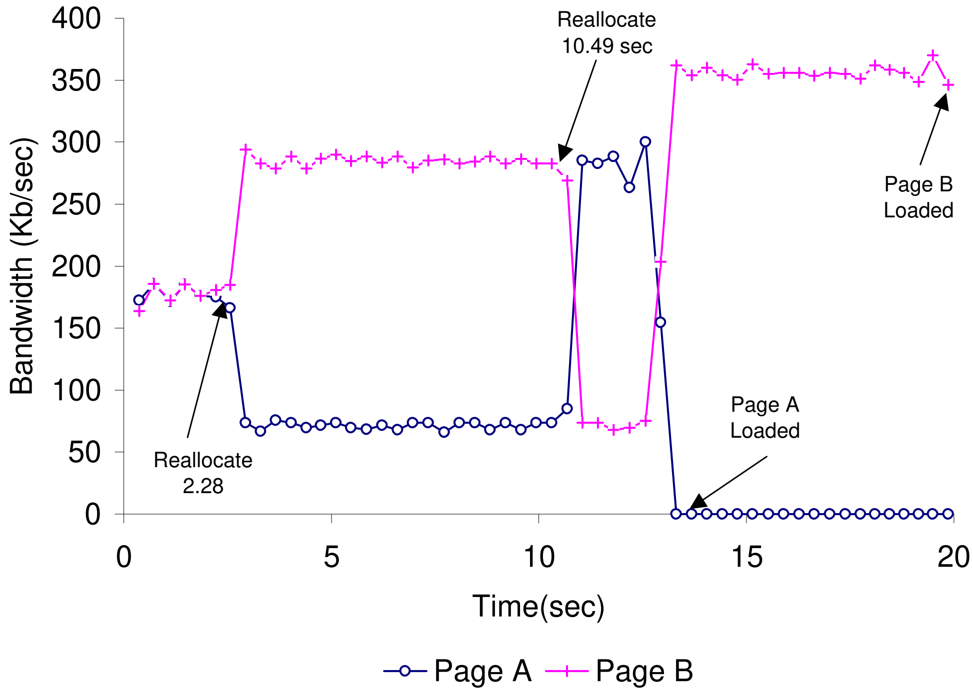


Figure 6.5 : Bandwidth allocations for transmitting two HTML documents with *Focus*, an adaptation policy that re-allocates 80% of the available bandwidth to the foreground document.

introduced by DummyNet (e.g., 2.38 seconds over 56 Kb/sec when I set the latency to 0.5 sec). This result meets our expectation, as the greater latency only affects the single message that is sent from the local proxy to the remote proxy to reconfigure the transmission hierarchy. The low reconfiguration latencies show good promise for the prospects of client-driven transmission scheduling reconfiguration even on networks with limited bandwidth and high latencies.

6.2.2 System Overhead

I measured the worst case HATS latency overhead by using IE5 to load a set of HTML documents downloaded from the Web. I load the documents without doing any adaptations or using any dynamic transmission strategy, using Puppeteer and Puppeteer-HATS. Figure 6.6 shows the results of this experiment for HTML documents of sizes ranging from 64 KB to 773 KB over network links of 56 Kb/sec, 128 Kb/sec, and 384 Kb/sec. The values in the plot show the overhead over loading the documents using native IE5 without any adaptive support.

This experiment represents the worst possible case for both Puppeteer and Puppeteer-HATS as it incurs the full overhead of parsing the document and transmitting the skeleton but do not benefit from any adaptation or transmission strategy (i.e., we only load one document at a time). In this experiment, Puppeteer and Puppeteer-HATS differ only in the mechanism they use to ship data to the local proxy. Puppeteer uses the conventional pull mechanism, while Puppeteer-HATS implements the push mechanism described in Section 6.1. For this experiment I configured the Puppeteer-HATS scheduling quantum to be 256 bytes.

Figure 6.6 demonstrates that, while Puppeteer-HATS overhead is slightly bigger than Puppeteer (an average of 11.17% over all documents and network speeds vs. an average of 7.75%), it is still small compared to total document loading time, especially for large documents (512 KB and over) on slow links, where adaptation and transmission scheduling are the most important.

The extra overhead in Puppeteer-HATS over Puppeteer results from the packet headers Puppeteer-HATS uses along with the transmission of component data (for multiplexing component data over a single TCP stream). For this experiment, with a quantum size of 256 bytes, sending the headers requires transmitting 4.72% more data, which is roughly similar to the ratio of header size over payload size (12/256=4.68%), and close to the difference in performance between Puppeteer and Puppeteer-HATS. Increasing the Puppeteer-HATS quantum size to 1024 bytes results in just 1.25% overhead for sending the packet headers and eliminates the performance gap between Puppeteer-HATS and Puppeteer. The smaller overhead comes, however, at the expense of increasing jitter.

The three dynamic HATS transmission strategies described in Section 6.2.1 had similar data overheads ranging between 4.6% to 4.8% extra data. The data overhead

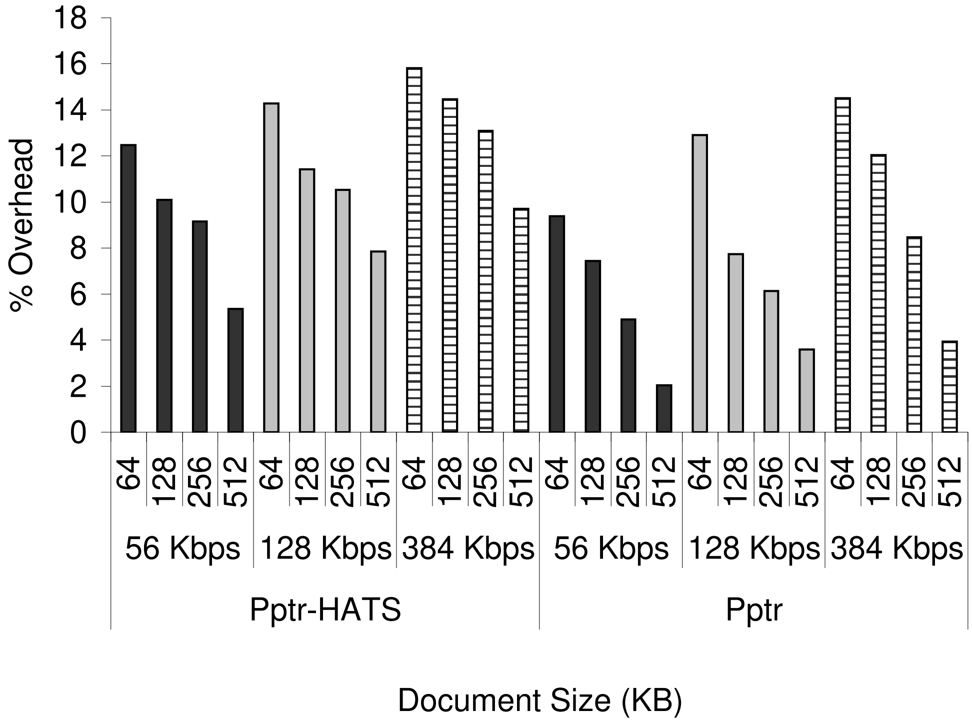


Figure 6.6 : Overhead for loading HTML documents of various sizes on Puppeteer (Pptr), and a Puppeteer-based HATS implementation (Pptr-HATS) over 56 Kb/sec, 128 Kb/sec, and 384 Kb/sec network links. Pptr-HATS quantum is set to 256 bytes. All times are normalized by time to load documents in native IE5, without Puppeteer support. Pptr-HATS overhead is small, especially for large documents (512 KB and over) on slow network speeds, where adaptation matters the most.

consists of data headers that identify data packets (*Headers*), skeleton data (*Skeleton*), and control messages for requesting objects and setting bandwidth allocations (*Control*). The *Skeleton* and *Control* components account for little overhead. The *Headers* account for the majority of the overhead with an average of 4.45% extra data. Although the overhead is not large, it can be reduced significantly by switching to a larger quantum size as explained above.

Finally, all three dynamic HATS transmission strategies achieve a high degree of network throughput. *TextFirst* and *LowFidelityFirst* measured an average of 93% and 92.6% network utilization over 56 Kb/sec, while *Focus* averaged a 93.08% network utilization over 384 Kb/sec.

6.3 Related Work

There is a large body of related work in this area. In this section I summarize a few of the projects I believe are most related to Puppeteer and HATS. I group these works depending on whether their main focus is on arguing for system-wide adaptation, implementing server and network support for scheduling, or enabling client-centric scheduling.

6.3.1 System-wide Adaptation

Several groups [45, 89, 104] 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. Moreover, none of these efforts provide support for scheduling data transmission for applications and documents running on the client.

6.3.2 Server and Network Support for Scheduling

Several projects [7, 27, 90, 135] have added differentiated services to general purpose operating systems. While most of these systems support hierarchical bandwidth scheduling, they do not understand the semantics of the applications they run, which prevents them from implementing more sophisticated system-wide adaption policies. They are typically deployed to manage server resources and do not provide system support for adapting to changes in user behavior or network connectivity. In contrast, the main objective of HATS has been to manage client bandwidth, while adapting to changes in connectivity and user behavior.

Significant effort has been invested into providing differentiated service in network hierarchies [19, 50, 92, 108, 109, 134]. HATS can use existing network facilities, where available, to reserve bandwidth between end-points. In cases where the network does not support bandwidth reservations, HATS measures the throughput of its connections and adapts to variations in available bandwidth.

Packeteer [3] provides monitoring and control of network traffic for ISPs. It enables setting quotas for different applications, network protocols, users, etc. Packeteer differs from HATS in that the focus is on rationing server-side resources and as such, it does not take any input form the client. Moreover, Packeteer is not dynamic and does not perform adaptation.

The Stream Control Transmission Protocol (SCTP) [133] supports the concurrent transmission of component data over a single stream.

6.3.3 Client-Centric Scheduling

Several projects enable clients to schedule their resources. Most of these efforts, however, are limited to scheduling the bandwidth, disk, or CPU usage of a single application and do no support adaptation.

Several projects have used application specific information to provide better support for file and media servers [93, 128, 129, 127]. Systems have been developed to enable clients to specify their QoS network requirements [29, 46, 144, 147].

Rialto [69], to the best of our knowledge, was the first system to introduce the notion of user-centric scheduling, where the system dynamically schedules resources in a way that dynamically maximizes the user's perceived utility of the system, rather than the performance of individual applications.

Ford and Susarla [51] introduced CPU inheritance scheduling, a novel processor scheduling framework in which arbitrary threads can act as schedulers for other threads. Goyal et al. [58] implemented a hierarchical CPU scheduler for a multimedia operating system. Their system, however, assigns applications to well defined service classes and does not support adaptation.

Eclipse/BSD [21] provides existing applications the ability to set disk resource reservations without modifying their source code. In Eclipse/BSD, the application is linked with a customized version of libc and the user sets reservations by adding a "needed-resources" property to the each file.

The WebTP [60] protocol allows Web browser to schedule the order in which objects of an HTML page are downloaded based on the client's hardware, network conditions, and user preferences. WebTP, however, does not schedule across documents or applications (i.e., two browsers downloading two separate pages).

Spring *et al.* [130] implemented a receiver-based system that manipulates the TCP receiver window size to prioritize between flows of three different service classes: interactive, short download and long download. Their system, however, does not implement a hierarchical scheduler and does not schedule between flows of the same class. Finally, its limited knowledge of data semantics, prevents it from implementing complex system-wide adaptation policies.

SCUBA [9] enables users of a distributed teleconferencing application to adjust their sending rates based on receiver interest. The collaboration group has an assigned capacity. Multiple sources in the group compete for a share of this capacity. Each source receives a part of the capacity depending on how much interest receivers show for the source. Similarly, RCTP [22] enables video clients to signal servers to reduce their transmission rate to save network bandwidth and reduce congestion.

Finally, recent efforts use resource scheduling to extend battery life for mobile devices by adaptively scaling the CPU's clock rate [94], or selectively disabling the network interface [25].

6.4 Summary and Conclusions

This chapter demonstrated a system that is capable of adapting multiple applications running concurrently on a bandwidth-limited device. Specifically, it described the design and evaluated the performance of the Hierarchical Adaptive Transmission Scheduling (HATS), which I implemented as an extension of the Puppeteer adaptation system. Together, Puppeteer and HATS support centralized transmission scheduling for bandwidth-limited devices and support dynamic system-wide adaptation policies based on the semantics of the applications, documents, and components running on the mobile or wireless device.

HATS enables Puppeteer to enforce the network allocations determined by the adaptation policies, to react quickly to changes in network connectivity and user behavior (e.g., by reallocating bandwidth to the higher priority components), and to implement sophisticated hierarchical transmission scheduling strategies.

The goal of this chapter was not to perform a comprehensive study of transmission strategies. Instead, I have implemented a small number of transmission strategies to demonstrate the benefits of the infrastructure and the fact that the overheads involved are small. For example, I have shown that even simple policies that prioritize text over other component types can achieve substantial savings in the time it takes to open text-only versions of large documents.

Chapter 7

Writing and Collaborative Work

There is much to be gained from enabling users of bandwidth-limited devices to edit documents and to share their modifications with their peers. For example, consider a team member on a wireless device that wants to collaborate with the rest of his team on a presentation. Since bandwidth limitations make it cumbersome to download or upload the entire presentation in full fidelity, the user downloads, instead, an adapted version of the presentation, perhaps containing the text and low-fidelity versions of the images. Alternatively, the user may know which slides he wants to change, and download only those slides. The user then modifies his version of the presentation, and uploads his changes back to the server. One of the items that the user added to the presentation is an image that the user just took with his digital camera. Transmission of such a large data item may take considerable time over a wireless link, so the adaptation system first sends a reduced-fidelity version of the image, and may upgrade it later with higher-fidelity versions.

While clearly useful, the above scenario has not been supported, to the best of our knowledge, by any previous adaptation system. This chapter introduces CoFi, a new architecture that supports document editing and collaborative work on bandwidth-limited devices.

I identify three factors that hinder document editing and collaborative work over bandwidth-limited links:

1. **The techniques used by adaptation systems to lower download latencies**. These systems reduce network traffic by use of *subsetting* and *versioning*. In subsetting adaptations, only a subset of the components of the original docu-

ment, for example the first page, is transferred. In versioning adaptations some of the components are transcoded into lower fidelity representations, for example low-resolution images. In either case, the documents present at the bandwidth-limited device are only partially loaded, and may be significantly different from the documents stored at the server. Naively storing user modifications to a partially loaded document may result in the deletion of components that were not included in the subset, or in the replacement of high-fidelity components with the lower-fidelity versions available at the bandwidth-limited device (even in cases where the user did not modify the transcoded components).

2. **Large updates**. Users can produce large multimedia content (e.g., photographs, drawings, audio notes) that may incur long upload latencies over the bandwidth-limited link.

3. **Conflicts**. The use of optimistic consistency models [80, 112] allows concurrent modifications, which may conflict with each other. Conflicts can occur in other circumstances as well, but low bandwidth and the possibility of frequent disconnection make their occurrence more likely.

This chapter introduces CoFi, a unified architecture that combines the notions of *co*sistency and *fi*delity, and supports document editing and collaborative work on bandwidth-limited devices. CoFi consists of two novel component-based mechanisms that enable document editing and collaborative work over bandwidth-limited links: *adaptation-aware editing* and *progressive update propagation*. These mechanisms extend traditional replication models with concepts of fidelity. Both mechanisms decompose documents into their component structure (e.g., pages, images, paragraphs, sounds), and keep track of consistency and fidelity at a component granularity. Adaptation-aware editing enables editing partially-loaded documents by differentiating between modifications made by the user and those which result from adaptation. Progressive update propagation shortens the propagation time of components created or modified at the bandwidth-limited device by transmitting subsets of the modified components or transcoded versions of those modifications. Adaptation-aware editing and progressive update propagation also reduce the likelihood of update conflicts. First, by working at the component level rather than the whole-document level, they reduce the sharing granularity. Second, because both mechanisms lower the cost to upload component data, they encourage clients to communicate more frequently, increasing the awareness that users have of their collaborators' activities.

I have found that these two techniques can be added without much difficulty to both optimistic and pessimistic replication models. Consistency maintenance methods for replicated objects are typically represented by state diagrams, and I follow this general paradigm. I present state diagrams that incorporate the presence of low-fidelity versions of components, for use with both optimistic and pessimistic replication. The introduction of low-fidelity component versions is orthogonal to the maintenance of consistency between replicas. More specifically, new states are added to represent low-fidelity versions, but the semantics of the existing states and the transitions between them remain unchanged. I therefore argue that the techniques presented in this chapter can be applied to any replication method without undue complication.

The CoFi mechanisms are general and allow for several possible implementations. One possibility is to write or modify applications to implement CoFi natively within the application. Another option, is to implement CoFi as part of a centralized adaptation system. I believe that the later is a more profitable proposition. Most parts of the CoFi architecture are bound to be common for most applications; by implementing CoFi in the general adaptation infrastructure I get to leverage the coding effort across a wide set of applications.

I have implemented a prototype that supports both application-aware editing and progressive update propagation in an optimistic replication protocol. The implementation is done by extending Puppeteer. I demonstrate the implementation with examples using progressive propagation of messages in Outlook and adaptation-aware editing and progressive update propagation for PowerPoint presentations. The implementation effort was modest (a total of 5700 lines of code), and the performance of the system is good, even though no changes were made to the applications.

This chapter is organized as follows. Section 7.1 introduces adaptation-aware editing and progressive update propagation and explores the implications of extending traditional replication models to support these mechanisms. Section 7.2 presents a prototype implementation of adaptation-aware editing and progressive update propagation on top of the Puppeteer adaptation system. Section 7.3 presents experimental results for document editing and collaborative work on bandwidth-limited clients using Microsoft Outlook and PowerPoint. Section 7.4 discusses related work. Finally, Section 7.5 summarizes and concludes the chapter.

7.1 Mechanisms

This section describes the implications of extending traditional replication models to provide various degrees of support for adaptation-aware editing and progressive update propagations.

I assume that multiple views of a component can co-exist in different replicas. Two views of a component may differ because they have different creation times, and hence reflect different stages in the development of the component, or because they have different fidelity levels. I consider two fidelity classes: *full* and *partial*. For a given creation time, a component can have only one full-fidelity view but many partial-fidelity views. A component is present at full fidelity when its view contains data that is equal to the data when the view was created. Conversely, a component is present with partial fidelity if its view was lossily transcoded from the component's original view. Fidelity is by nature a type-specific notion, and hence there can be a type-specific number of different partial-fidelity views. I assume, however, that all the views of a component can be arranged into a monotonically increasing order according to their fidelity, the last one being a full-fidelity view.

I consider both pessimistic and optimistic replication models. A pessimistic replication model guarantees that at most one replica modifies a component at any given time, and that a replica does not modify a component while it is being read by some other replica. The mutual exclusion guarantee can be realized by various mechanisms such as locks or invalidation messages. With optimistic replication, replicas can read and write components without any synchronization, and communication occurs only when two replicas reconcile.

I also consider both primary replica and serverless approaches. In a primary replica approach, a server holds the primary replica of the document. Clients can replicate subsets or all of the document's components by reading components from the server's primary replica. Client modifications are sent to the server, and there is no direct communication between clients. I also discuss the implications of a serverless configuration, in which there is no centralized server or primary replica and there is direct communication between replicas.

I first describe the implications of supporting adaptation-aware editing and progressive update propagation in isolation. I then describe replication models that support both mechanisms. The initial discussion assumes a primary replica. Serverless systems are discussed in Section 7.1.4.

7.1.1 Adaptation-Aware Editing

The simplest form of adaptation-aware editing limits users to modifying only components that are loaded with full fidelity at the bandwidth-limited device. Such an implementation requires the system to keep track of which components are available at the bandwidth-limited device and whether these components have been transcoded into partial-fidelity versions. This information is normally already present in the adaptation system or easily added. The system then prevents users from modifying any component that is not present with full fidelity.

A simple extension to the previous model is to allow users to (completely) overwrite or delete components present at partial-fidelity or components that were not included in the client's replica subset. In this scenario, the user can (completely) replace the content of a component that was not loaded, or that was loaded at partial fidelity, with new full-fidelity content generated at the bandwidth-limited device. The user can also remove a component from the document altogether. Adding this functionality does not require keeping extra state, but the system may wish to warn the user that he is about to overwrite or delete a component for which a full-fidelity view is not locally available.

Pessimistic Replication

Figure 7.1(A) shows the state transition diagram for individual components of a client replica for a pessimistic replication model that supports modifying full-fidelity component views and overwriting partial-fidelity component views. In our state diagrams, I represent new low-fidelity states and the new transitions in and out of these states with gray ovals and dotted lines, respectively. In contrast, I represent the states present in traditional replication models and their transitions with clear ovals and full lines, respectively. The state diagram for the primary replica stays the same as without support for application-aware editing, containing two states, Empty and Clean, with the obvious meanings, and is not shown.

In the client replica state diagram, a component can be in one of four states: Empty, Partial-Clean, Clean, and Dirty. A component is in Empty when it is being edited by some other client replica or when the client chose not to read it. A component transitions into Partial-Clean when the client replica reads a partial-fidelity view. This view can be further refined by reading higher-fidelity partial-fidelity views (i.e., Read-Partial) or the component can transition into Clean by reading a full-fidelity view. The component transitions into Dirty when the client replica either modifies a full-fidelity view (i.e., a component in the Clean state) or overwrites an

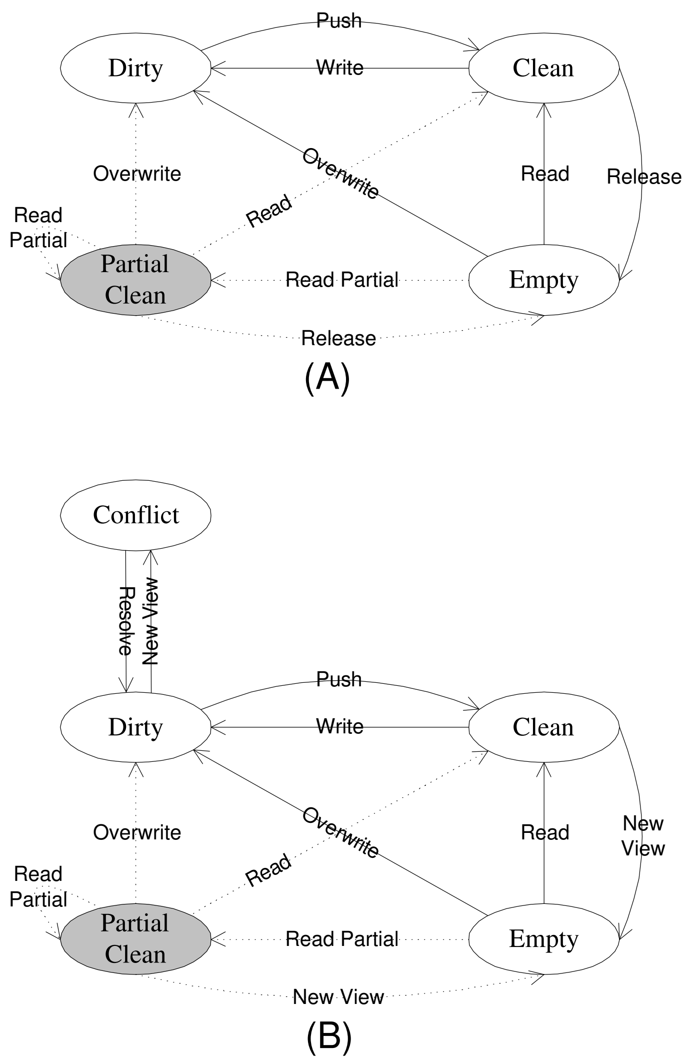


Figure 7.1 : State transition diagram for a pessimistic (A) and optimistic (B) replication model with support for adaptation-aware editing. Low-fidelity states and the transitions in and out of these states are represented with gray ovals and dotted lines, respectively. In contrast, states present in traditional replication models and their transitions are represented with clear ovals and full lines, respectively.

unloaded component or a partial-fidelity view (i.e., a component in Empty or Partial-Clean). The component transitions back to Clean when the client replica propagates a full-fidelity view to the primary replica. Finally, a component transitions back to Empty when the client replica no longer wishes to read the component. Transitions to Empty depend on the specific mechanisms used to guarantee mutual exclusion, and can occur, for example, when the client replica releases a lock or receives and invalidation.

Optimistic Replication

Figure 7.1(B) shows the state transition diagram for an optimistic replication model. The optimistic replication diagram differs from the pessimistic diagram (Figure 7.1(A)) in two ways. First, it has an extra state for conflict resolution. The component transitions to the Conflict state when the replica detects the primary replica view and the client replica view are concurrent (i.e., it is not possible to determine a partial ordering for the two views) [86]. The component transitions back to the Dirty state once the client replica reads the conflicting view and resolves the conflict. Second, transitions between the Clean and Partial-Clean states to the Empty state occur when the client replica learns that the primary replica has a more recent view for the component.

Modification of Partially Loaded Components

A more ambitious form of application-aware editing and one that results in significant complications is to allow users to modify just a portion of a partial-fidelity view, for example, replacing parts of transcoded images, audio recordings, or video streams. Such modifications result in views that contain a mixture of partial and full-fidelity data, which contravenes our initial assumption that the system keeps track of fidelity at the component granularity. While the semantics of some data types, such as images may support modifications to just parts of the component's view, these semantics are

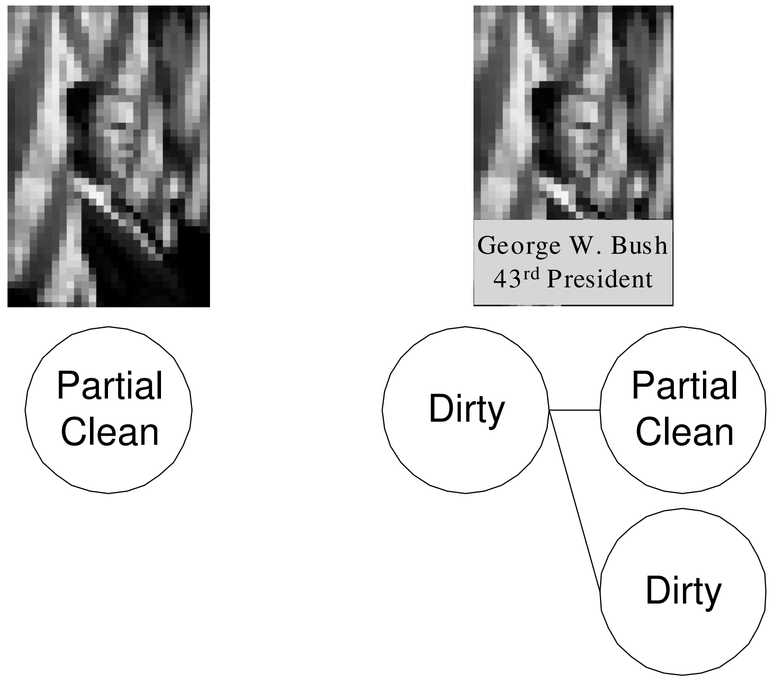


Figure 7.2 : A component is split into two subcomponents when a user modifies a partial-fidelity view.

not visible to the replication system. To make the changes visible to the system, the component has to be split into two subcomponents as shown on Figure 7.2. The first subcomponent holds the partial-fidelity data, which was not modified by the user, and the second subcomponent holds the user's full-fidelity modifications. The original component (now turned into a container for the two subcomponents) transitions to the Dirty state, to reflect the change in the document's component structure. The subcomponent holding unmodified partial fidelity data remains in the Partial-Clean state. In contrast, the subcomponent holding new full-fidelity data transitions to the Dirty state.

The partial-fidelity subcomponent is further subdivided when the user modifies it again. As Figure 7.3 shows, however, a client replica may have at most two subcomponents for each of the original components, as two full-fidelity subcomponents can be merged into a larger subcomponent. The figure shows that when the partial-

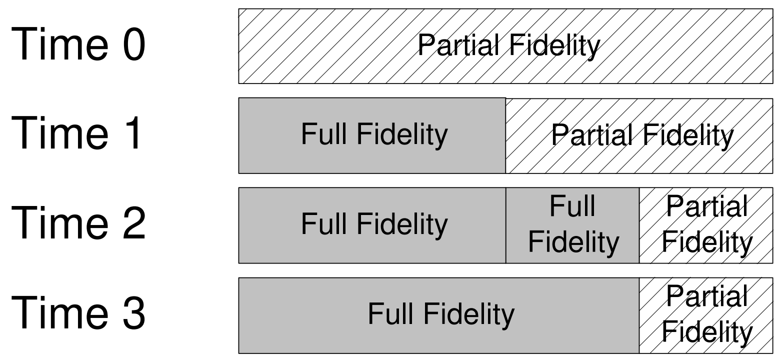


Figure 7.3 : When a partial-fidelity subcomponent is subdivided, the resulting full-fidelity sub-subcomponent can be merged with an existing full-fidelity subcomponent.

fidelity subcomponent is subdivided for a second time, the resulting full-fidelity sub-subcomponent can be merged with the full-fidelity subcomponent created in the first subdivision. For a similar reason, a component at the primary replica does not have to be subdivided, even when it was subdivided at a client replica. Conceptually, when a client replica propagates modifications to a component that was split, it has to split the component in the primary replica. In this section, however, since I limit client replicas to propagate only full-fidelity data, both subcomponents at the primary replica (the one the client replica changed and the one it did not change) will have full-fidelity data and can be merged into a single full-fidelity component. In Section 7.1.3, I describe systems that require subdividing components at the primary replica and enable subdividing a component into an arbitrary number of subcomponents.

A partial-fidelity view can also be changed by an operation that does not produce any full-fidelity data, for instance by applying a gray-scale filter. In some cases, the operation rather than the resulting data can be propagated to the primary replica, and applied there. Additional complications result from the fact that the adaptation system may receive a higher-fidelity version of the component while it is being modified by the user.

7.1.2 Progressive Update Propagation

A system can support progressive update propagation by propagating a subset of the modified components and/or by propagating partial-fidelity views of modified components.

I first consider the implications of an implementation that supports progressive update propagation but does not support transcoding components on read or editing partial-fidelity components. In such an implementation, client replicas have by default full-fidelity views of the components they replicate. A client replica has a partial-fidelity view for a component only when the component is being updated by some other client and the updates are being progressively propagated. In other words, the decision to propagate partial-fidelity data is made by the replica that is writing the component and not by the reader, as was the case in the previous section. Moreover, independent of whether I implement a pessimistic or optimistic approach to replication, once a partial-fidelity view has been propagated to the primary replica, it can only be replaced with another view created by the same writer (i.e., a higher-fidelity view or a more recent view). Replacing a partial-fidelity view with a view created by a different writer would violate the restriction over editing partial-fidelity components.

Pessimistic Replication

Supporting progressive fidelity propagation requires adding one new state to the primary replica's state transition diagram (Partial-Clean) and two new states to the client replica's state transition diagram (Pseudo-Dirty and Partial-Clean).

Figures 7.4 (A) and (B) show the state transition diagram for an individual component in a pessimistic replication model at a client replica and at the primary replica, respectively. The transition diagram for the primary replica is simple. A component at the primary replica can be in one of three states: Empty, Partial-Clean, and Clean. A component is in the Empty state while it is being edited by a client replica. The

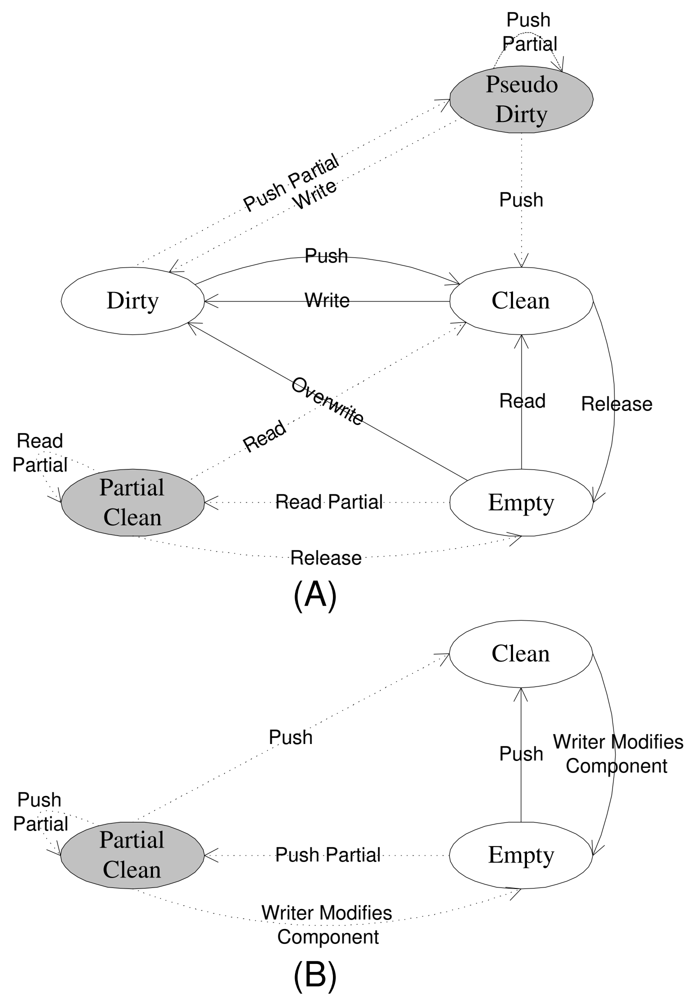


Figure 7.4 : Pessimistic replication state transition diagram for components of the client (A) and primary (B) replicas.

component transitions to the Partial-Clean and Clean states when the writer pushes a partial-fidelity view or a full-fidelity view of the components, respectively.

A component in a client replica can be in one of five states: Empty, Clean, Partial-Clean, Dirty, and Pseudo-Dirty. A component is in the Empty state while it is being modified by some other client replica. A component transitions to Clean by reading a full-fidelity view. If only a partial-fidelity view is available at the primary replica because the last writer has not propagated a full fidelity view yet, the client replica can read this view and transition Partial-Clean. The component transitions from Clean to

Dirty after the client modifies its content. The client can then propagate modifications to the primary replica in two ways. First, the writer can push a full-fidelity view of the modifications, forcing the component at the primary and the writer's replica to transition to Clean. Second, the writer can propagate a partial-fidelity view of the component, forcing the component to transition to Partial-Clean in the primary replica, and to Pseudo-Dirty in the writer's replica. The various replicas will remain in these states until the writer pushes a full-fidelity view and the component at both the primary and the writer's replica transition to Clean. At this time, other client replicas can read the full-fidelity view and transition to Clean. Alternatively, a writer in Pseudo-Dirty can modify the component for a second time and transition to Dirty. This can only happen, of course, if the last writer can regain exclusive access to the component by, for example, acquiring a write lock or invalidating all other copies.

Optimistic Replication

In an optimistic replication scheme, before propagating modifications to a component, the writer has to determine if these modifications conflict with other modifications previously reflected at the primary replica. If there is a conflict, the client replica will have to resolve it by merging (in a type-specific way) the full-fidelity views of the conflicting modifications. After resolution, the client replica can propagate a full- or partial-fidelity view of the component to the primary replica. If, however, the primary replica has only a partial-fidelity view for a conflicting component (i.e., the concurrent writer has not propagated a full-fidelity view of its modifications), the two views can not be merged as this would violate the restriction on editing partially-loaded components, as described in Section 7.1.2. In this case, conflict resolution has to be delayed until the client replica, which propagated the conflicting partial-fidelity view, propagates a full-fidelity view of its modifications.

7.1.3 Combining Adaptation-Aware Editing and Progressive Update Propagation

This section explores the implications of extending pessimistic and optimistic replication models to support both partial document editing and progressive update propagation. I consider systems that support all the features presented in Sections 7.1.1 and 7.1.2.

Pessimistic Replication

Figure 7.5 (A) shows the state transition diagram for components at the client replica for a pessimistic replication system that supports adaptation-aware editing and progressive update propagation. The state transition diagram for components at the primary replica is the same as the one shown in Figure 7.4 (B).

The diagram in Figure 7.5 (A) is similar to that of Figure 7.4 (A) and most states have similar semantics. The semantics of Partial-Clear are, however, a little different. A component may be in the Partial-Clean state because the client requested a partial-fidelity view to reduce its network usage, or because only a partial-fidelity view of the component is available at the primary replica.

Client replicas can also modify partially-loaded components (i.e., component in Partial-Clean). If the client overwrites the partial-fidelity data with new client-generated full-fidelity data, the component transitions to Dirty. In contrast, if the client only modifies a portion of the component's view, the component is split in two as described in Section 7.1.1. The subcomponent that contains the new high-fidelity data transitions to Dirty, and the subcomponent with the unmodified partial-fidelity data remains in Partial-Clean. In contrast to Section 7.1.1, the client replica has to split the component at the primary replica when it starts propagating partial-fidelity views of its modifications.

Moreover, when the client replica releases the partial-fidelity subcomponent, other client replicas can read and modify partial-fidelity versions of this subcomponent.

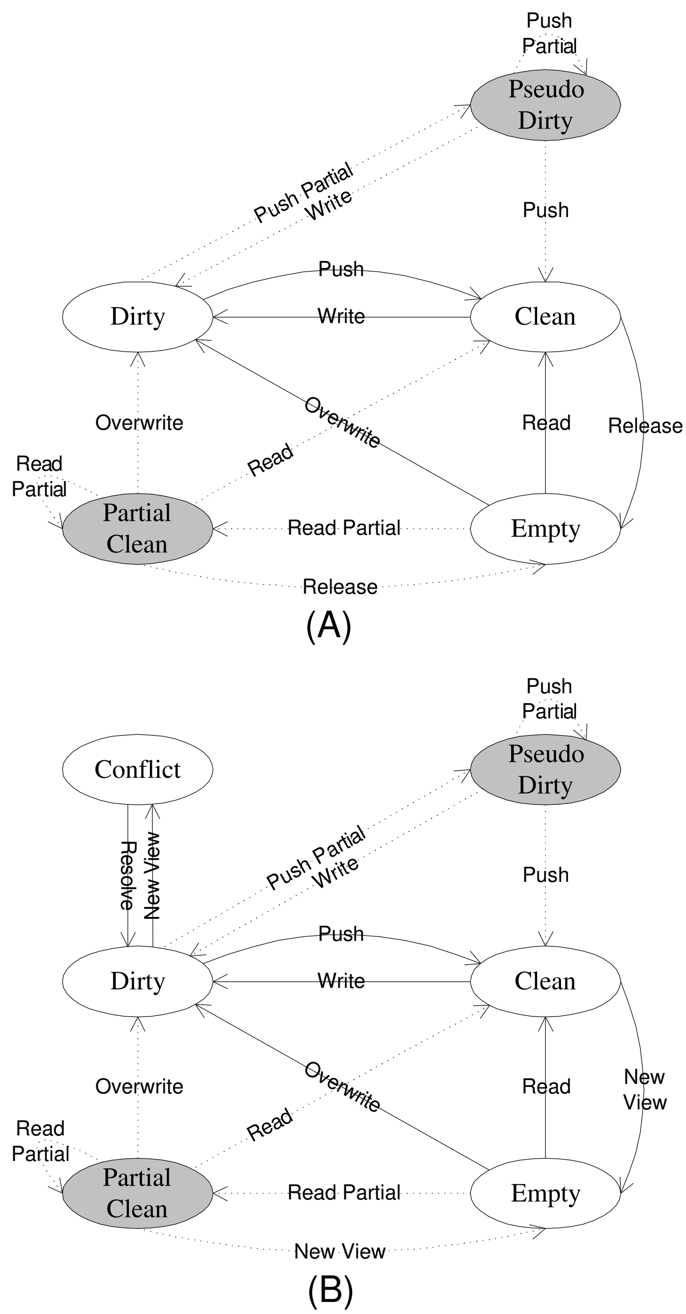


Figure 7.5 : State transition diagrams for two implementation that support partial document editing and progressive update propagation based on pessimistic (A) and optimistic (B) replication models.

Since these other client replicas can also propagate partial-fidelity versions of their own modifications, the subcomponent may be subdivided into an arbitrary number of sub-subcomponents. However, even when multiple client replicas modify different subcomponents of the same component, the document's component structure always remains consistent as client replicas need to achieve exclusive access before modifying a partial-fidelity view and further subdividing the subcomponent.

Optimistic Replication

Figure 7.5 (B) shows the state transition diagram for an optimistic replication model that supports partial document editing and progressive update propagation. The state transition diagram is similar to that of the pessimistic replication model discussed in the previous section, with states and transitions with the same names having equivalent semantics. The optimistic replication diagram differs however in two ways. First, it has an extra state for conflict resolution. Second, it transitions from Clean and Partial-Clean to Empty and from Dirty to Conflict when the client replica reconciles with the primary replica and learns about a more recent or concurrent component view.

The optimistic replication model also enables client replicas to resolve conflicts even when the server's primary-replica just has a partial-fidelity view for the component. In such case, as in any other conflict, the client replica would read the conflicting partial-fidelity view, resolve the conflict, and choose whether to propagate a full- or partial-fidelity view of the modifications.

Finally, an optimistic replication system that enables the propagation of partial-fidelity modifications made to partially-fidelity views also has to be able to resolve conflicts that may arise in the document component structure. Figure 7.6 illustrates a simple component structure conflict, which results from concurrent partial modifications to partial-fidelity views. Initially, both client replicas read partial-fidelity views of the same component. Then both client replicas split the component by mod-

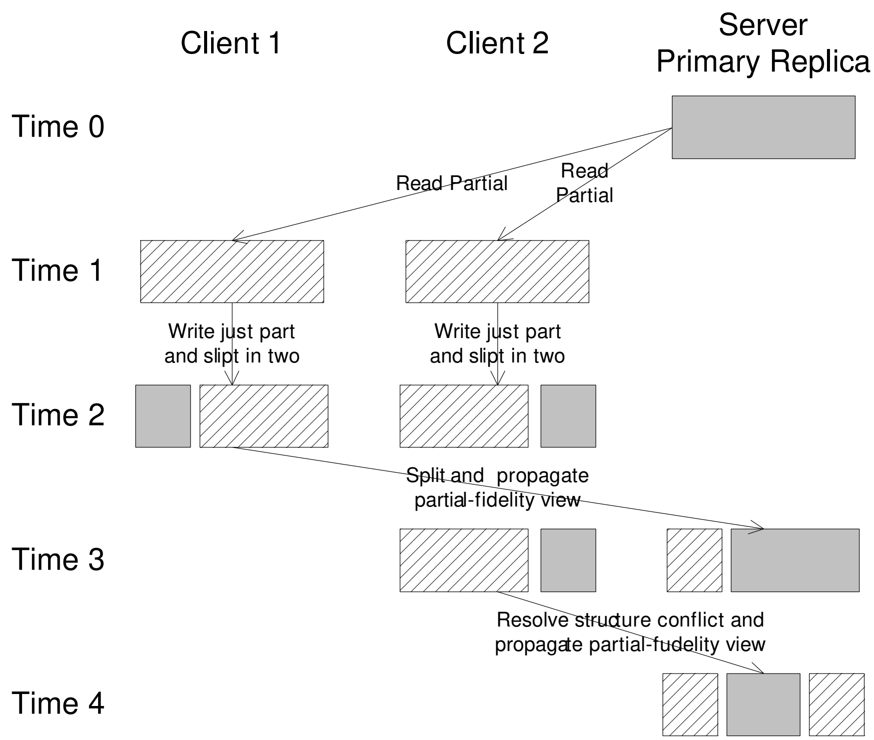


Figure 7.6 : Example of subcomponent component structure conflict resolutions in an optimistic replication system.

ifying different sections of their partial-fidelity view. When Client1 reconciles with the primary replica, it partitions the primary replica's component and propagates a partial-fidelity view of its modifications. Later, when Client2 reconciles with the primary replica it detects the concurrent modifications to the component. Client2 also realizes that its subcomponent structure differs from the primary replica's subcomponent structure. Client2 resolves the conflict by reading the primary replica's subcomponent views and repartitioning the subcomponent structure. After resolving the conflict, Client2 propagates a partial-fidelity view of its modifications.

7.1.4 Serverless Replication

The earlier state diagrams can be carried over from a primary replica configuration to a serverless configurations without any change. In a serverless configuration, when a replica modifies a component it becomes the source for distributing these modifications to other replicas. In practice, however, not all replicas have to read the modifications directly from the source replica and replicas can get these modifications from some other replica that in turn got the modifications from the source replica.

Independent of how the modifications are propagated, the last writer has a full-fidelity view of the component and is perceived by other replicas as the source for this view. Hence, the replica that writes the component last becomes effectively a temporary “primary replica” for the component that it modified. The states and state transitions otherwise remain the same. If a replica, other than the primary, wants to progressively read the component from the primary replica, it needs to maintain a Partial-Clean state. If the temporary primary replica for a particular component wishes to progressively update its modifications, it needs to maintain a Partial-Dirty state. If it wants to do that concurrently to more than one replica, it needs to maintain the progress of each individual transmission as part of that state.

7.1.5 Summary

I have described the changes necessary to the state diagrams for pessimistic and optimistic replication in order to support adaptation-aware editing and progressive update propagation. In general, the changes involve adding states and transitions. The existing states and transitions remain, with their original semantics. No other changes are necessary if we disallow modifications to partially-loaded components. Significant complications arise if this restriction lifted, requiring components to be split to reflect old partial-fidelity data and new full-fidelity data.

7.2 CoFi Prototype

The mechanisms presented in the previous section allow for several possible implementations. One possibility is to modify applications to implement adaptation-aware editing and progressive update propagation natively. An alternative is to implement

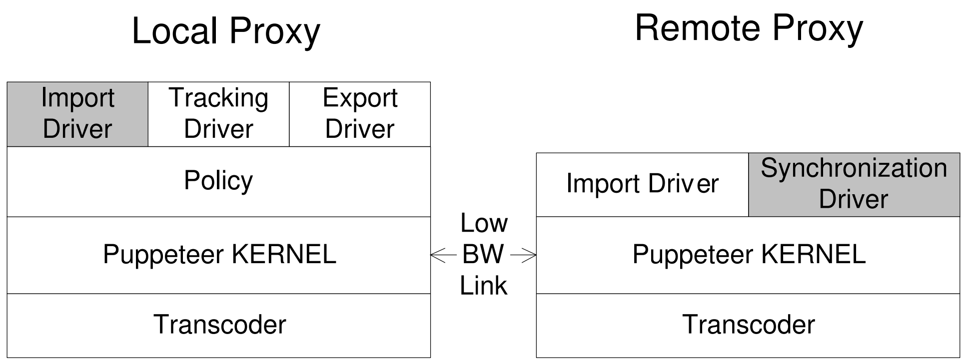


Figure 7.7 : Puppeteer remote and local proxy architectures. Shaded components show extension made to Puppeteer to implement CoFi.

these mechanisms as part of a centralized adaptation system. I believe that the latter is a more profitable proposition, as it leverages the coding effort across a wider set of applications.

This section presents a CoFi prototype implemented on top of the Puppeteer adaptation system. The CoFi-enabled Puppeteer prototype (Puppeteer-CoFi) supports editing partially loaded documents, incremental update propagation, and conflict resolution.

7.2.1 Puppeteer-CoFi

Puppeteer-CoFi implements an optimistic client-server replication model that supports adaptation-aware editing and progressive update propagation. The prototype consists of a group of bandwidth-limited nodes, each running a Puppeteer-CoFi local proxy, that collaborate by exchanging component data over a single Puppeteer-CoFi remote proxy, which stores the primary replica of the document. The Puppeteer-CoFi remote proxy, in contrast to the Puppeteer remote proxy, stores hard state. When a document is first opened, it is imported from its native data store into the Puppeteer-CoFi remote proxy. Further accesses to the document, both reads and writes, are then served from the Puppeteer-CoFi remote proxy's copy. To enable communication with

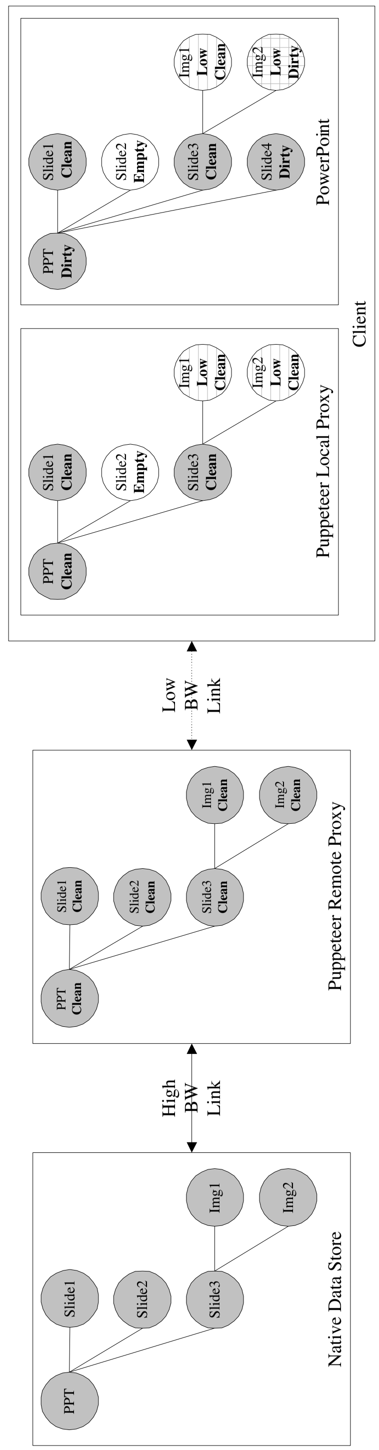


Figure 7.8 : At the start of the update propagation process, the native data store, Puppeteer remote and local proxies, and the application the can have copies of the document that differ in their component subsets and views.

non Puppeteer-CoFi-enabled programs, Puppeteer-CoFi exports document modifications back to their native data store, where it is assumed that no clients will directly modify the data.

In Puppeteer-CoFi, several copies of a document can co-exist in various parts of the system. Figure 7.8 exemplifies the state of the system for a single client editing a PowerPoint document. The figure shows that there is one copy of the document in each of the native data store, the Puppeteer-CoFi remote and local proxies, and the application running on the client. Moreover, the figure shows that these copies differ in their component subsets and component fidelities. In the example, the native data store and the Puppeteer-CoFi remote proxy have full copies of the document. In contrast, both the Puppeteer-CoFi local proxy and the application have just partial copies: the first slide is empty and the images of the second slide are only present in low fidelity. Finally, the application copy has an extra slide component.

The differences between the remote and local proxies copies result from subsetting and versioning adaptations. In contrast, differences between the local proxy and application copies result from user modifications. Finally, the native store and remote proxy copies can differ because the two copies have not been synchronized (i.e., modifications have not been propagated to the native store) or when the native store copy is being updated using out-of-band mechanisms (i.e., outside of the Puppeteer-CoFi system).

The rest of this section describes how copies of the document converge by exchanging component data. First, I describe how Puppeteer-CoFi propagates user modifications from the bandwidth-limited device to the Puppeteer remote proxy and the data store. Second, I describe how the Puppeteer local proxy refreshes the application's document copy with new and higher-fidelity view.

User modifications propagation

Application-specific adaptation policies running on the Puppeteer local proxy (as described in Chapter 4) control the propagation of user modifications to the Puppeteer remote proxy and the native data store. Update propagation involves four stages: acquiring user modifications, detecting and resolving conflicts, transmitting modifications to the remote proxy, and synchronizing the modifications with the document's native store system (SMTP, NFS, a relational database, etc). While these stages need to be executed in order for a given component, they do not have to be executed simultaneously or uniformally for all components. Specifically, the adaptation policy determines for each component of the document when to resolve conflicts (if there are any), when to propagate modifications and with what fidelity, and when to reflect modification to the native data store. For example, if the adaptation policy determines that modifications to a set of components conflict with the remote proxy's copy, it could choose to delay resolving the conflicts and propagate just modifications to non-conflicting components; leaving conflict resolution for a later time (e.g., on reconnection over high bandwidth).

Acquire modifications Users modify documents by changing their component structure or by modifying individual components. Structure modifications include adding, deleting, and reordering components. Changes to individual components involve modifications to the component's view (i.e., its native data). Finally, since it is common for a component to encode the component structure of its children, any changes in the children component structure usually implies a change in the parent's view (i.e., when a document rearranges its parts, the document itself has changed).

Puppeteer-CoFi uncovers user modifications by comparing the local proxy's document copy to the application's copy. To compare the document copies, Puppeteer needs to access the data that is currently loaded inside the application. This can be done in two ways. The simplest one, although the least available one, involves using the application's exported API to retrieve any user modifications. The alternative is to instruct the application to serialize its data into a temporary document (e.g., a file or an email). The temporary document is then parsed using an import driver (as described in Chapter 4) and compared to the local proxy copy. For the comparison to be valid, it must be possible to uniquely identify each document component independently of the data stored in its view. If I had to rely on the component view (i.e., a hash of the view data or some other finger printing techniques) to determine its identity, I would not be able to differentiate between a newly created component and one that just changed. The easiest way to determine identity, of course, is for the application to assign unique and immutable component IDs to components at creation time. Unfortunately, many applications do not preserve the IDs of components as the document evolves. For these applications, Puppeteer-CoFi requires extra component-specific driver code that (while executing outside the application) fixes the component IDs, ensuring that they remain immutable over the lifetime of the document. I describe in Section 7.3.2 the steps I take to ensure that PowerPoint preserves unique identifiers for slides and images.

Conflict detection and resolution Puppeteer-CoFi detects conflicting modifications by tagging component views with logical timestamps [78], which determine the partial order of modifications in the system. Timestamps consist of the client IDs of nodes that modified the component and the logical times of the modifications. We say that timestamp A dominates timestamp B (i.e., is more recent) if A contains all the client IDs of B and all logical times of A are equal or greater than in B. In contrast, we say that timestamps A and B are concurrent if neither dominates the other.

Puppeteer-CoFi implements both client- and server-based conflict resolution. In client-based resolution, the local proxy fetches the conflicting view from the remote proxy, resolves the conflict and creates a new view that dominates the two conflicting views. In server-based resolution, the client pushes its view to the remote proxy, and a transcoder executing in the remote proxy creates a new view that merges the conflicting modifications.

I argue that most conflicts can be resolved automatically by application-specific adaptation policies without requiring user intervention. I base this argument in the observation that most conflicts detected by Puppeteer-CoFi result from a mismatch between the application's semantic consistency unit and the component structure exposed to Puppeteer. In other words, modifications within a component usually occur in non-overlapping parts, prompting Puppeteer-CoFi to detect a conflict that can be resolved automatically. Moreover, since the sub-component structure of a component is usually encoded in its view, non-overlapping changes to the sub-structure (e.g., adding new components in non-overlapping sections of the document) result in an update conflict for the parent component.

When user intervention is necessary to resolve a conflict, i expect conflict resolution to be client-based. The application-specific adaptation policy can use an export driver to present the conflicting component views in the context of the applications environment. The application-specific adaptation policy can then ask the user to fix the conflict by either selecting one of the views or by merging the conflicting views into a new view.

Modification transmission The Puppeteer-CoFi remote proxy implements a simple consistency model, where the remote proxy accepts only views that are either more recent or have higher fidelity than the current view of the component. This limits the Puppeteer local proxy to propagating only conflict-free modifications. Server-based conflict resolution, as explained above, is somewhat of an exception to this rule. While the client does propagate a conflicting view, a transcoder executing in the remote proxy creates a new view by merging the user modifications with the remote proxy view. At the end of this process, the remote proxy has a view that dominates both conflicting views.

As mentioned earlier, the application-specific adaptation policies select the subset of component for which to propagate modifications to the server. For components that encode their sub-component structure in their view, the adaptation policy needs to take into consideration the following two constraints:

1. Adding, removing, or re-arranging a component also requires updating its parent.

2. Any update to a component that has children, requires updating the structure of its children (adding and removing).

These constraints ensure that the remote proxy does not service a view that references a component for which the remote proxy does not yet have a view. It also ensures that if a component has been removed, it will be removed from the document subset of any client that reads the new view of the parent component, which no longer references the deleted component.

The above constraints require that the transmission of views for components that have a parent-child relationship appear as atomic to the remote proxy. Puppeteer-CoFi provides atomicity by keeping shadow copies of the new views and re-evaluating the constraints whenever a new view is transmitted to the remote proxy. When both constraints hold for all related components, the remote proxy atomically declares the shadow view to be the active views for all related components.

The fidelity with which to propagate a view is set by application-specific policies running on the client, which can request transcoding transformations for the view. To increase the fidelity of a previously propagated modifications, the application-specific policy can re-select the component and push a higher-fidelity version of its view.

Synchronization with native storage I extended the Puppeteer proxy architecture to include synchronization drivers that can export documents back to their native storage system. I export documents to their native storage to enable information sharing with client outside of the Puppeteer system and to leverage from the mechanisms that these storage systems implement (e.g., availability, fault tolerance, security, etc).

Refreshing the application's document copy

Application-specific adaptation policies may choose to refresh the application's document copy to reflect changes made by other users to the document or to increase the fidelity of component present at low fidelity.

Refreshing the application's document copy involves the following steps: fetching the new component views, detecting and resolving conflicts, and reflecting the new views in the application's document copy. Fetching component view proceeds as explained in Chapter 4 — the adaptation policy just chooses what components to fetch and with what fidelity. Detecting and resolving conflicts is necessary to make sure the system does example, if the user edits a low fidelity image that the prototype is trying to upgrade, it is necessary to merge the user modifications with the higher fidelity data fetched from the remote proxy. The prototype detects and resolves conflicts in a similar manner as described did in Section 7.2.1. Finally, the adaptation policy reflects the modifications in the application's document copy by using the export drivers.

7.3 Implementation Experience

This section reports on our experience using Puppeteer-CoFi to adapt the Outlook email client and PowerPoint presentation system for document editing and collaboration over bandwidth-limited links. I implemented progressive update propagation for both applications, while for PowerPoint, I also support adaptation-aware editing. Table 7.1 shows the lines of code written to support these two applications.

The rest of this section is structured as follows. Section 7.3.1 describes the programming effort involved in writing Puppeteer-CoFi drivers and adaptation policies for the Outlook email client. Section 7.3.2 describes the programming effort

	PowerPoint	Outlook
Import Driver	1790	1674
Export Driver	697	220
Tracking Driver	148	85
Policy	680	386
Total	3315	2365

Table 7.1 : Line counts for Puppeteer-CoFi drivers and policies.

involved in writing Puppeteer-CoFi drivers and adaptation policies for PowerPoint. Section 7.3.3 presents experimental results that quantify the performance of the implementation.

7.3.1 Outlook

In collaboration with Rajnish Kumar, I implemented an email service that supports the progressive propagation of images embedded in or attached to emails. Figure 7.9 shows the architecture of the email service. On the sender side, the progressive email service uses Outlook's email client to generate emails. On the receiving end, the email service supports both Puppeteer-CoFi-enabled clients running Outlook and standard third-party email readers.

The email service implements emails as Puppeteer-CoFi shared documents that are written only by the email's sender but are read by one or more recipients. We implemented a sender adaptation policy that propagates the text content of new emails and transcodes image attachments into a progressive JPEG representation and sends just portions of an image's data. The sender can propagate fidelity refinements for the attached images by selecting the email from a special Outlook folder and re-sending it. The image fidelity refinements are available to Puppeteer-CoFi-enabled recipients as soon as they reach the Puppeteer-CoFi remote proxy. For Puppeteer-CoFi-enabled recipients, we implemented an adaptation policy that fetches the email's text content and transcoded versions of its image attachments. Readers request fidelity refinements

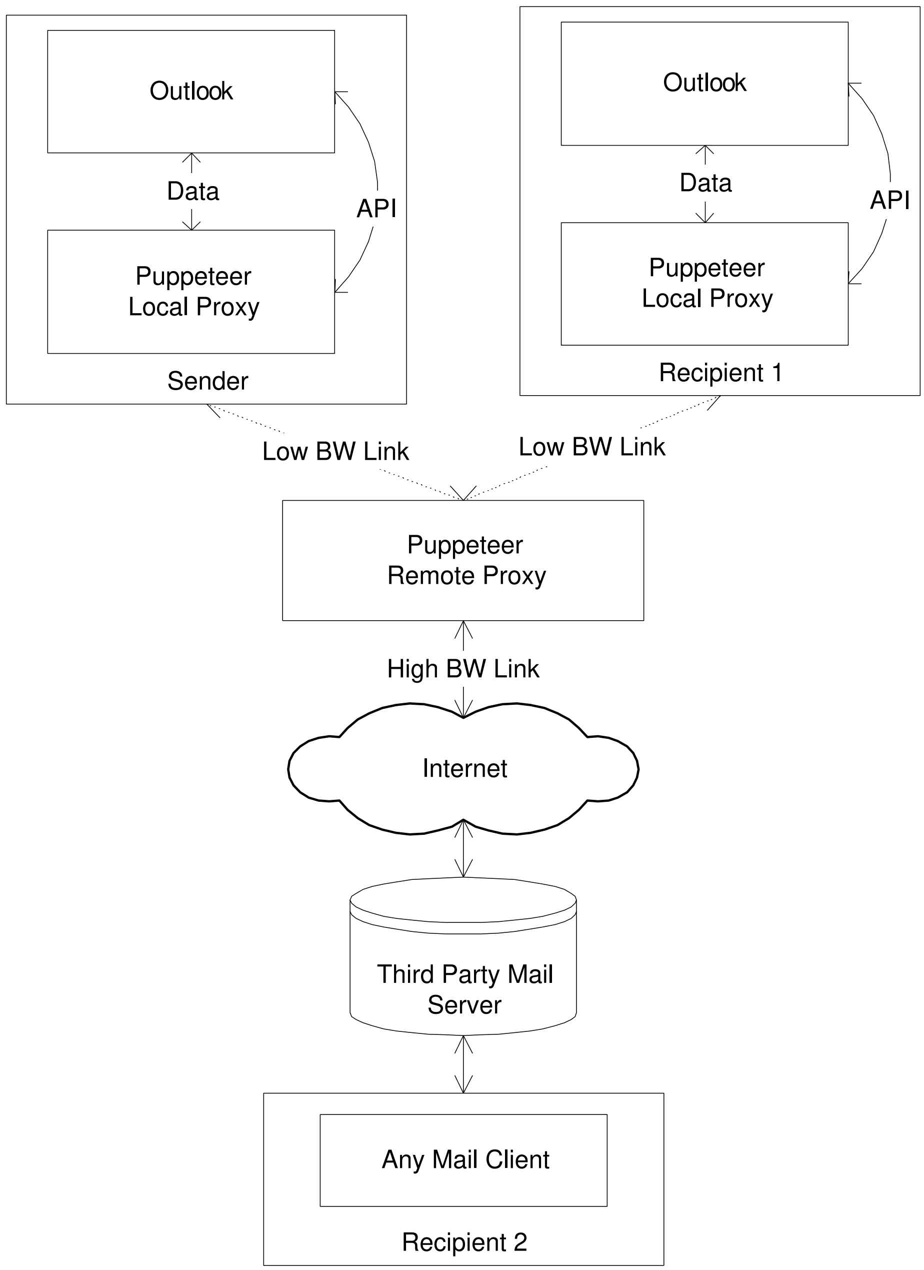


Figure 7.9 : Progressive email service architecture.

by clicking a refresh button added on Outlook's toolbar. Finally, the progressive email service supports third-party email readers by composing a new email message every time all of the email's image attachments reach a predetermined fidelity level. For example, if the progressive email service is configured to assume that images have two fidelity levels (one-seventh and full), the service will send each third-party email reader two versions of the same email; each containing the same text body but with images of progressively refined fidelities.

7.3.2 PowerPoint

This section describes how the prototype propagates user modifications and updates PowerPoint with more recent or higher-fidelity data.

User Modification Propagation

Acquire Modifications PowerPoint's exported API does not provide facilities for determining which presentation objects have been modified or for extracting these modifications. Instead, the prototype serializes the state of the presentation by instructing PowerPoint to save a temporary copy of the presentation in the client's local file system. The prototype saves the temporary copy in the new XML-based PowerPoint file format [95] (i.e., SaveAs Web Page) because the human-readable nature on XML simplifies the task of writing import drivers to parse the presentation.

Before the prototype can acquire the user modifications by parsing the temporary copy and comparing it with the local proxy's copy, it needs, however, to perform an intermediate step to account for the fact that the techniques used to update slides and to increase the fidelity of embedded images change the Ids of these components.

The PowerPoint exported API does not provide a mechanism for directly replacing the contents of presentation components (e.g., slides, images). The Puppeteer-CoFi PowerPoint adaptation policy, instead, upgrades slides and images by loading an auxiliary presentation that contains the components with their more recent or higher-fidelity data and uses the clipboard to copy slides and the images to the user's presentation. The PowerPoint policy then deletes the old versions of the slides and images from the user's presentation.

To the user, the above process is transparent. The presentation looks mostly the same, it just shows more recent or higher-fidelity data. To PowerPoint, however, the update and refinement of slides and images modifies the component structure of the presentation. From PowerPoint's perspective, the older or lower-fidelity components were deleted from the presentation and the more recent or higher-fidelity components were added to the presentation as new components. This results in a situation where an updated component, while appearing to be the same component to the user, is actually assigned a different component Id by PowerPoint.

Fortunately, PowerPoint allows querying the Ids of components pasted into the user's presentation. The prototype uses this functionality to record the new Ids of updated or upgraded components. The prototype then pre-processes the temporary copy of the presentation stored in the client's local file system and rename all components to their original Ids. At this stage, the prototype can parse the temporary copy and compare it with the local proxy's copy to acquire user modifications.

As Table 7.1 shows, implementing the import drivers for PowerPoint required 1790 lines of Java code. Of this line count, an estimated 250 lines implement the code that renames the component Ids.

Conflict Detection and Resolution A conflict occurs when: one user modifies a slide while another user deletes it, two users move a slide to different positions in the presentation, or two users concurrently modify the same slide. I refer to these conflicts as *edit-delete*, *move-move*, and *edit-edit*, respectively. For simplicity, for the rest of this section, I will refer to the copy of the presentation available at the remote and local proxies as the remote and local copies, respectively.

I now describe how the PowerPoint policy I implemented resolves the above conflicts. This is, however, just one of the possible ways to resolve conflicts, and I can easily envision multiple variations or extensions to the simple policy I implemented. The PowerPoint policy resolves edit-delete conflicts by giving priority to slide editing over deletion. If the remote copy has a new view for a slide, while the local copy shows the slide as deleted, the PowerPoint policy fetches the new view for the slide and inserts it into the running application, effectively canceling the slide deletion. If the situation is the opposite and the remote copy shows the slide as deleted and the local copy shows a new view for the slide, the policy will just undelete the slide from the remote copy.

The PowerPoint policy resolves move-move conflicts by giving priority to the local copy. This is achieved by simply updating the root component of the remote copy (i.e., the document) to reflect the new position of the slide.

The PowerPoint policy resolves edit-edit conflicts by using the PowerPoint export driver to present to the user the two conflicting slides. Once the user has resolved these conflicts, either by choosing one of the slides or by merging their content, the PowerPoint policy propagates the new slide view (which dominates both the old local and remote views) to the remote copy.

Modification Transmission I experimented with incremental propagation by implementing a policy that transcodes new images into a progressive JPEG representation. I assume that images have 3 fidelity levels, which I will refer to as *low*, *medium*, and *high*. I chose these levels to correspond to the first one-seventh, next two-sevenths, and last four-sevenths of the bytes in the progressive JPEG representation.

The first time the user requests to save the changes, the policy propagates the new images to the remote proxy at low fidelity. On subsequent save requests, the policy upgrades the fidelity of the images available at the remote proxy to medium and high fidelity.

Synchronization with Native Storage Our present implementation assumes that the PowerPoint presentation is only edited by Puppeteer-CoFi-enabled clients. This assumption affords the implementation of a simple synchronization driver that recreates in the native storage the PowerPoint presentation in its XML-based format. If the assumption were to be violated, and users were to modify the PowerPoint presentation outside of Puppeteer-CoFi, the synchronization driver would need to acquire any modifications made to the native copy, and resolve any conflict before regenerating the document in the native store.

Refreshing the Application's Document Copy

The PowerPoint policy uses the copy-and-paste mechanism described earlier to add new components to the presentation and to replace components with more recent or higher-fidelity data. The PowerPoint policy also uses the PowerPoint exported API to delete and reorder slides to reflect changes to the presentation by other users.

Before any changes or upgrades can be reflected in the user's presentation, it is necessary to detect if the components we are about to add, delete, move, or upgrade have not been modified by the user, resulting in a conflict. The prototype detects modifications and resolves conflict by following the *acquire modifications* and *conflict detection and resolution* steps described in the Section 7.2.1.

7.3.3 Experimental Results

I measure the performance of our adaptations on an experimental platform consisting of three 500 MHz Pentium III PCs running Windows 2000. Two of the machines are configured as clients and one as a server. Client machines run the user application and the Puppeteer local proxy. The server machine runs the Puppeteer remove proxy. Clients and the Puppeteer server communicate via a fourth PC running the DummyNet network simulator [117]. This setup allows control over the bandwidth between clients and server to emulate various network technologies. I use our departmental

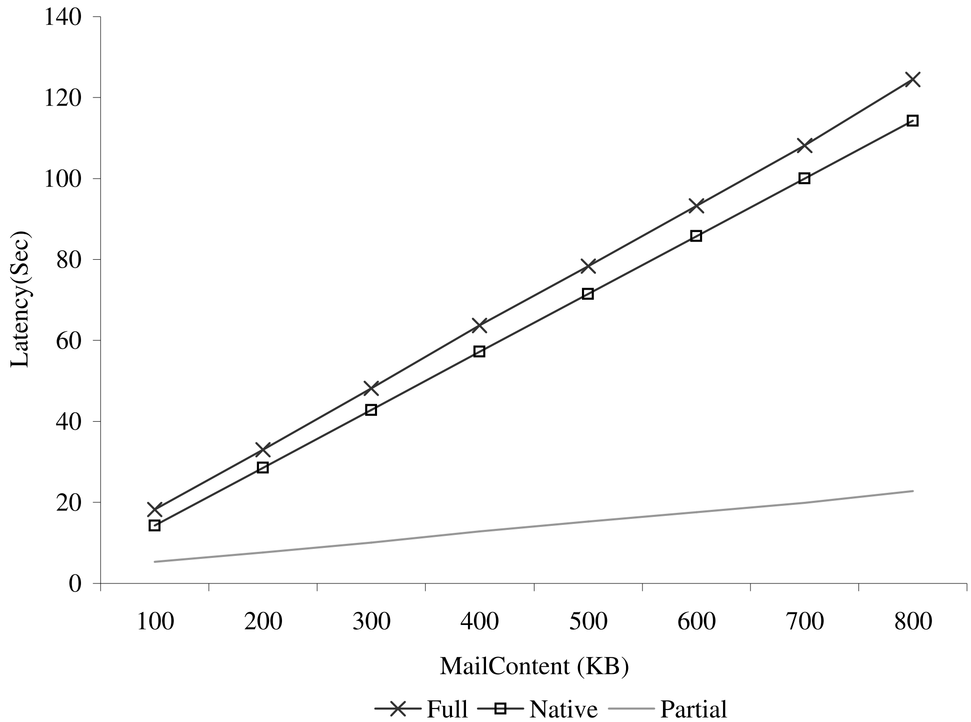


Figure 7.10 : Latency for sending emails with and without progressive update propagation over 56 Kb/sec link.

NFS and IMAP servers as the native stores for our experiments with PowerPoint and Outlook, respectively.

Outlook

Figure 7.10 plots the latencies for transmitting a set of synthetic emails consisting of a few text paragraphs and a variable number of image attachments each of size 100 KB over 56 Kb/sec link. The figure shows results for a run that uses Outlook without any adaptation support (Native), and two Puppeteer-CoFi runs, one that sends the full attachments (Full) and another that uses versioning to convert images into a progressive JPEG representation and transfers only the initial one-seventh of the images' data (Partial). For the Native run, I measure only the time it takes to transmit the emails between the mobile client running Outlook and an SMTP server on the other end of the bandwidth-limited link. This accounts for the time that the mobile client has to wait before disconnection in order to propagate the email. I do not include the time it takes the SMTP server to deliver the email to the recipients, as these operations can be done asynchronously and do not require the mobile client to remain connected. Similarly, for Puppeteer-CoFi runs, I measure only the time it takes to transmit the emails between the Puppeteer-CoFi local and remote proxies and do not include the time needed to compose and send emails to third-party email recipients, or the time it takes Puppeteer-CoFi-enabled recipients to adaptively read the email. The Full run demonstrates that the Puppeteer-CoFi overhead is small, averaging less than 5% over all emails. In contrast, the Partial run shows, as was to be expected, that progressive propagation of the attachments reduces the latency by roughly 80%.

I measured the time it takes a Puppeteer-CoFi-enabled recipient to upgrade the fidelity of an image attachment from one-seventh of the image's content to full-fidelity. This time was less than 12% and 2% of the transmission time for 10KB image and 100KB image attachment, respectively, on 56 Kb/sec link. This time is proportional to the size of the image attachment, and partly it is also proportional to the size of text body if the image is embedded in the body. On average, the time taken to upgrade the fidelity of an embedded image increased only by 8 milliseconds for every 1 KB increase in text body, and only by 2 milliseconds for every 10 KB increase in size of embedded images, while the increase in transmission time is 1.2 seconds for every 10KB increase in the email size.

PowerPoint

I adapted PowerPoint to support adaptation-aware editing and progressive update propagation. Adaptation-aware editing reduces download time by enabling mobile clients to edit PowerPoint presentations that have been aggressively adapted. For example, in Chapter 5, I demonstrated that loading text-only versions of PowerPoint

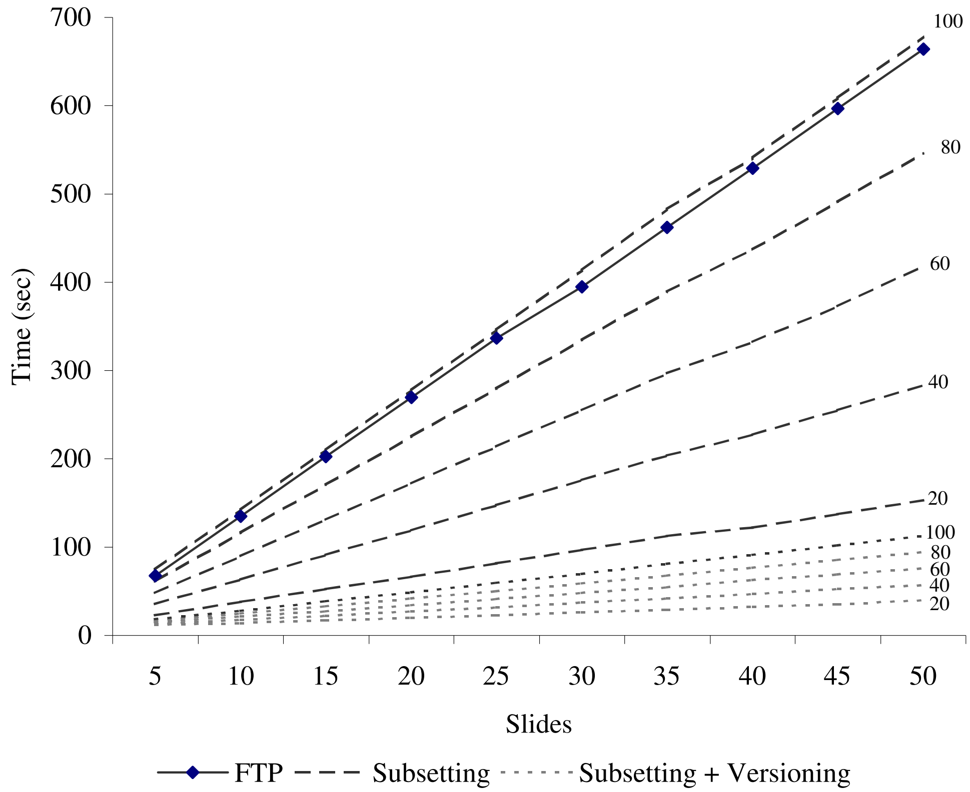


Figure 7.11 : Latency for saving modifications to PowerPoint presentations with and without adaptation over 56 Kb/sec link.

presentations can reduce download latency by over 90% for large presentations over bandwidth-limited links. The rest of this section evaluates the benefits of progressive update propagation.

Puppeteer-CoFi-enabled PowerPoint propagates modifications progressively by saving subsets of the modified slides or embedded objects, or by transcoding embedded images into a progressive JPEG representation and saving just portions of the images' data. The adaptation policy I have implemented propagates modifications every time the user saves the document. The policy propagates the text content of any new or modified slide and transcoded versions of new or modified embedded images. Image fidelity refinements are then propagated on every subsequent save request untill all images at the Puppeteer-CoFi remote proxy reach full fidelity.

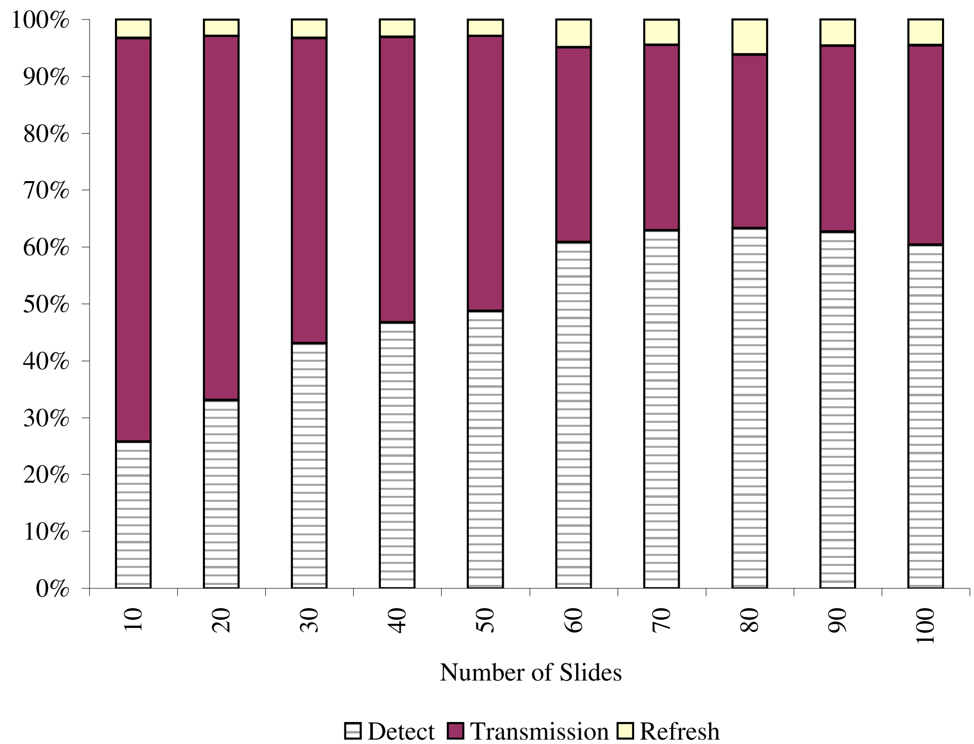


Figure 7.12 : Latency breakdown for replacing one slide in PowerPoint documents of various sizes.The figure shows that as the document size increases, the time to acquire modifications becomes the dominating factor of the update process.

I evaluated the effectiveness of progressive update propagation by measuring the latency for saving modifications to a set of synthetic PowerPoint documents. I constructed our synthetic documents by replicating a single slide that contained 4 KB of text and a 80 KB image.

Figure 7.11 shows latency measurements for saving PowerPoint documents with up to 50 slides over a 56 Kb/sec network link. The figure shows latency results for transferring the documents over FTP and adaptation policies that use subsetting and versioning to reduce the data traffic. The FTP measurements provide a baseline for the time it takes to transfer the full document without any adaptation. I use this baseline to determine the effectiveness of our adaptation policies.

Figure 7.11 plots the results of 5 experiments that use subsetting to reduce latency. The numbers on the right hand side of the figure, next to each line, show the proportion of document slides that was saved to the remote proxy. In this manner, the topmost line corresponds to documents that were saved in their entirety, while the lowest subsetting line corresponds to documents where modifications to only 20% of the slides were saved. In all experiments, I assume that the text and the single image in each slide were modified and had to be saved. The topmost subsetting line, which corresponds to saving the full document, shows that the Puppeteer-CoFi overhead is small, averaging less than 5% over all documents. In contrast, as was to be expected, all other subsetting experiments show significant reductions in upload latency. The last five lines in Figure 7.11 show the results for an adaptation policy that uses subsetting and versioning of images to further reduce upload latency. This policy converts images embedded in slides into a progressive JPEG representation and transfers only the initial one-seventh of the image's data. As was to be expected, the versioning policies achieve even larger reductions in upload latency.

Figure 7.12 shows the breakdown of the execution time for updating a single slide to reflect changes made by some other user. The figure shows that the API calls to replace the content of the slide (Refresh) account for a small portion of the overall latency. More significant, the figure shows that for large documents, roughly 60% of the reconciliation time is spent making sure the user did not change the slide the prototype is about to update (Detect). Detecting modifications is time consuming because PowerPoint's API does not support querying if a components has been changed. Instead, the prototype detects modifications by saving a temporary copy of the presentation, parsing this copy, and comparing it with the local proxy copy of the presentation.

The time to detect modifications places a hard bound on how frequently the prototype can save modifications or refresh the application's copy with new or higher-fidelity data. While the 23 seconds that it takes to acquire modifications for the 100-slide synthetic presentation will probably not bother the user when saving changes (a rather infrequent operation), it limits the ability to load the document incrementally and the speed and frequency with which the system can reflect modifications made by other users. Moreover, this result exposes the inefficiency of detecting user modifications by serializing documents into temporary files. Since serialization time grows roughly linearly with the size of the presentation and is independent of how much the document was changed, it is to be expected that the proportion of time spent detecting modification will increase as the presentation grows. It is important to note that this is a limitation of the PowerPoint API and not of the CoFi mechanisms.

7.4 Related Work

While several adaptation systems [10, 36, 45, 53, 72, 80, 89, 98, 103, 123] use subsetting and versioning to reduce document download time, to the best of our knowledge, CoFi is the first to provide adaptation support for document editing and collaborative work over bandwidth-limited devices.

WebDAV [143], and LBFS [99] implement file systems for wide-area and low-bandwidth networks. Coda [85], Locus [111, 142], Ficus [116], and Bayou [137] provide support for document editing on disconnected devices. CoFi differs from these previous efforts in that it assumes that update propagation and conflict resolution can occur over bandwidth-limited connections and that do not have to wait for the device to be strongly connected.

Other systems have experimented with reducing the size of the consistency unit to prevent concurrent modifications that result from a mismatch between the consistency units of the application and the system. Some examples are Shasta [125], Cashmere [132] and Millipage [68].

Several efforts [28, 55, 91] have used component-based technologies to implement collaborative applications that adapt to variations on network connectivity, or implemented collaborative applications that use the document's component structure to reduce conflicts or limit the amount of data that needs to be present at the device [24, 38, 39, 81, 106, 120]. These efforts, however, do not allow the propagation of low-fidelity versions of modifications. MASSIVE-3 [59] uses transcoding to reduce data traffic necessary to keep users of a collaborative virtual world aware of each other. MASSIVE-3, however, implements a pessimistic single-writer consistency model.

An alternative to shipping data between replicas to achieve consistency is operation shipping. In operation shipping, the set of operations applied to the local replica is logged and later replayed in every replica in order to bring all replicas into an equivalent state. Some systems that implement operation shipping are Orca [15, 16], Globus [141], and a variant of Coda [87].

7.5 Summary and Conclusions

I described CoFi, a novel architecture for supporting document editing and collaborative work on bandwidth-limited devices. CoFi consists of two component-based mechanisms: adaptation-aware editing and progressive update propagation. Both mechanisms decompose documents into their components structures (e.g., pages, images, paragraphs, sounds) and keep track of consistency and fidelity at a component granularity. Adaptation-aware editing enables editing partially-loaded documents by differentiating between modifications made by the user and those which result from adaptation. Progressive update propagation shortens the propagation time of components created or modified at the bandwidth-limited device by transmitting subsets of the modified components or transcoded versions of the modifications.

I demonstrated that support for adaptation-aware editing and progressive update propagation can be added to optimistic and pessimistic replication protocols in an orthogonal fashion. Specifically, new states are added to the state machines that describe the replication protocols, but the existing states and transitions remain unaffected.

I implemented adaptation-aware editing and progressive update propagation on top of the Puppeteer component-based adaptation system and experimented with document editing and collaboration in two real world applications. For these applications, the ability to read and edit partially loaded documents and progressively propagate fidelity refinements of modifications substantially reduces upload and download latencies, improving application responsiveness and increasing awareness about concurrent changes to documents. Finally, the programming effort to support each additional application is moderate, and the system overhead is low.

Chapter 8

Conclusions and Future Work

In this thesis, I presented the concept of component-based adaptation. Underlying component-based adaptation is the idea that the adaptation policies control applications by calling methods in the APIs that the applications already export. This approach has a number of advantages, allowing for a wide variety of adaptation policies to be implemented with popular office productivity applications, and achieving significant latency reductions over low-bandwidth links, yet requiring no modifications to the applications.

I implemented the concept of component-based adaptation in the Puppeteer system and evaluate the extent to which existing APIs can be used for the purposes of adaptation. Although I found some limitations in the APIs of existing applications, I was able to implement a large number of adaptations policies without much complexity and with little overhead.

Puppeteer's modular architecture is specifically designed to limit the amount of development for integrating new applications and new adaptations. Overall, I 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 for a new application. Writing new adaptation polices proved much easier.

In experimenting with adapting existing applications, I identified limitations in these application's interfaces and file formats that either complicate the task of adding adaptive behavior or limit the ways in which applications can be adapted.

Investigating how to maximize the adaptation potential of applications and proposing design guidelines for adaptation-friendly run-time APIs and document formats, remain open research questions. Specifically, more research is needed in order to answer the following questions. What is the best way to structure an application in order to maximize its adaptation potential? What is the functionality that an application's API should provide in order to support the broadest set of adaptations? What are the characteristics of an application's document format which promote or prevent adaptation?

In this thesis, I also described a system that is capable of adapting multiple applications running concurrently on a bandwidth-limited device. Specifically, I described the design and evaluated the performance of the Hierarchical Adaptive Transmission Scheduling (HATS), which I implemented as an extension of the Puppeteer adaptation system. Together, Puppeteer and HATS support centralized transmission scheduling for bandwidth-limited devices and supports dynamic system-wide adaptation policies based on the semantics of the applications, documents, and components running on the mobile or wireless device.

Wireless networks are by nature broadcast channels. When multiple clients share a wireless access point, they compete for the available network resources. When clients are being adapted, there is a correlation between adaptation decisions made for one client and the bandwidth available to other clients. Developing scheduling techniques that support adaptation policies that span across clients remains an open question. More research in necessary in order to implement policies that provide fairness to multiple wireless clients, while delivering bandwidth where it will be most effective. In particular, more research is necessary to determine how to achieve QoS guarantees by doing adaptation.

Finally, I described CoFi, a novel architecture for supporting document editing and collaborative work on bandwidth-limited devices. CoFi consists of two component-based mechanisms: adaptation-aware editing and progressive update propagation. Adaptation-aware editing enables editing partially-loaded documents by differentiating between modifications made by the user and those which result from adaptation. Progressive update propagation shortens the propagation time of components created or modified at the bandwidth-limited device by transmitting subsets of the modified components or transcoded versions of the modifications. I demonstrated that support for adaptation-aware editing and progressive update propagation can be added to optimistic and pessimistic replication protocols in an orthogonal fashion. Specifically, new states are added to the state machines that describe the replication protocols, but the existing states and transitions remain unaffected.

While the focus of this thesis has been on providing mechanisms that enable adaptation, these mechanisms are only as effective as the adaptation policies that drive them. To be effective, adaptation policies need to be tailored to the capabilities of the device and the user's access patterns and preferences. Requiring users to configure adaptation policies is unrealistic, as few users are willing to spend time reconfiguring their system. More research is necessary to develop techniques that monitor the user's behavior and automatically generate adaptation policies by deducing the user's intent and the relationships between components.

Bibliography

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

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

[3] Four steps to application performance across the network. Technical report, Packeteer, Inc., 2000.

[4] Tarek F. Abdelzaher and Nina T. Bhatti. Web content adaptation to improve server overload behavior. In *Proceedings of The 8th International World Wide Web Conference*, Torondo, Canada, May 1999.

[5] Sarita V. Adve and Mark D. Hill. Implementing sequential consistency in cache-based systems. In *Proceedings of the 1990 International Conference on Parallel Processing*, pages 47–50, August 1990.

[6] Paulo Sergio Almeida, Carlos Baquero, and Victor Fonte Minho. Panasync: Dependency tracking among file copies. In *Proceedings of the 9th ACM SIGOPS European Workshop*, Kolding, Denmark, September 2000.

[7] Werner Almesberger. Linux network traffic control - implementation overview, April 1999. `http://lrcwww.epfl.ch/linux-diffserv/`.

[8] Peter A. Alsberg and John D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 562–570, San Francisco, California, October 1976.

[9] Elan Amir, Steve McCanne, and Randy Katz. Receiver-driven bandwidth adaptation for light-weight sessions. In *Proceedings of the Fifth ACM International Conference on Multimedia '97*, Seattle, WA, November 1997.

[10] David Andersen, Deepak Basal, Dorothy Curtis, Srinivasan Srinivasan, and Hari Balakrishnan. System support for bandwidth management and content adaptation in Internet applications. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, California, October 2000.

[11] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless network file systems. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Copper Mountain, Colorado, December 1995.

[12] Kamal Ayad, Mark Day, Steve Foley, Dan Gruen, Steve Rohall, and Quinton Zondervan. Pagers, Pilots, and Prairie Dog: Recent work with mobile devices at Lotus Research. In *Proceedings of Workshop on Handheld CSCW at CSCW'98*, Seattle, WA, November 1998.

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

[14] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. In *Proceedings of the 13th Symposium on Operating Systems Principles*, pages 198–211, Pacific Grove, California, October 1991.

[15] Henri E. Bal, Raoul Bhoedjang, Rutger Hofman, Ceriel Jacobs, Koen Langendoen, Tim Ruhl, and M. Frans Kaashoek. Performance evaluation of the Orca shared object system. *ACM Transactions on Computer Systems*, 16(1), February 1998.

[16] Henri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.

[17] Rob Barrett, Paul P. Maglio, and Daniel C. Kellem. How to personalize the Web. In *Proceedings of the Conference on Human Factors In Computing Systems (CHI 95)*, Denver, Colorado, May 1995.

[18] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 168–176, March 1990.

[19] Jon C. R. Bennett and Hui Zhang. Hierarchical packet fair queuing algorithms. In *Proceedings of SIGCOMM'96*, Stanford, California, August 1996.

[20] Ninna Bhatti, Anna Bouch, and Allan J. Kuchinsky. Integrating user-perceived quality into Web server design. In *Proceedings of The Nineth International World Wide Web Conference*, Amsterdam, Netherlands, May 2000.

[21] Jose Brustoloni, Eran Gabber, Avi Silberschatz, and Amit Singh. Quality of service support for legacy applications. In *Proceedings of the International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'99)*, pages 3–11, Basking Ridge, New Jersey, June 1999.

[22] Ingo Busse, Bernd Deffner, and Henning Schulzrinne. Dynamic QoS control of multimedia applications based on RTP. *Computer Communications*, 19(1):49–58, January 1996.

[23] Michael J. Carey and Miron Livny. Conflict detection tradeoffs for replicated data. *ACM Transactions on Database Systems (TODS)*, 16(4):703–746, December 1991.

[24] Rhonda Chambers, Dean Crockett, Greg Griffing, and Jehan-Francois Paris. A Java tool for collaborative editing over the Internet. In *Proceedings of the 1998 Energy Sources Technology Conference and Exhibition (ETCE '98)*, Houston, TX, February 1998.

[25] Surendar Chandra. Wireless network interface energy consumption implications

of popular streaming formats. In *Proceedings of the 2002 Conference on Multimedia Computing and Natworking (MMCN'02)*, San Jose, California, January 2002.

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

[27] Shigang Chen and Klara Nahrstedt. Hierarchical scheduling for multiple classes of applicatons in connection-oriented integrated-service networks. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems (ICMCS)*, Florence, Italy, June 1999.

[28] Keith Cheverst, Gordon Blair, Nigel Davies, and Adrian Friday. Supporting collaboration in mobile-aware groupware. *Personal Technologies*, 3(1):33–42, March 1999.

[29] David D. Clark and David L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of ACM SIGCOMM '90*, pages 201–208, Philadelphia, Pennsylvania, September 1990.

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

[31] John M. Danskin, Geoffrey M. Davis, and Xiyong Song. Fast lossy internet image transmission. In *1995 ACM Multimedia Conference*, San Francisco, California, November 1995. `http://www.cs.dartmouth.edu/~jmd/fliit.ps.Z`.

[32] Susan B. Davidson. Optimism and consistency in partitioned distributed database systems. *ACM Transactions on Database Systems*, 9(3):456–481, 1984.

[33] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen. Consistency in partitioned networks. *ACM Computing Surveys*, 17(3):341–370, September 1985.

[34] Eyal de Lara, Karin Petersen, Douglas B. Terry, Anthony LaMarca, Jim Thornton, Mike Salisbury, Paul Dourish, Keith Edwards, and John Lamping. Caching

documents with active properties. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems (HotOS-VII)*, Rio Rico, Arizona, March 1999.

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

[36] Eyal de Lara, Dan S. Wallach, and Willy Zwaenepoel. Puppeteer: Component-based adaptation for mobile computing. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS)*, San Francisco, California, March 2001.

[37] Eyal de Lara, Dan S. Wallach, and Willy Zwaenepoel. HATS: Hierarchical adaptive transmission scheduling for multi-application adaptation. In *Proceedings of the 2002 Multimedia Computing and Networking Conference (MMCN'02)*, San Jose, California, January 2002.

[38] Dominique Decouchant, Vincent Quint, and Manuel Romero Salcedo. Structured cooperative authoring on the World Wide Web. In *Proceedings of the Fourth International World Wide Web Conference*, Boston, Massachusetts, December 1995.

[39] Dominique Decouchant and Manuel Romero Salcedo. Alliance: A structured cooperative editor on the Web. In *Proceedings of the ERCIM workshop on CSCW and the Web*, Sant Augustin, Germany, February 1996.

[40] Alan F. deLespinasse. Rover Mosaic: E-mail communication for a full-function Web browser. Master's thesis, Massachusetts Institute of Technology, June 1995.

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

[42] Maria R. Ebling, Lily B. Mummert, and David C. Steere. Overcoming the

network bottleneck in mobile computing. In *Proceedings of the 1st Workshop on Mobile Computing Systems and Applications*, Santa Cruz, California, December 1994.

[43] Keith W. Edwards. Flexible conflict detection and management in collaborative applications. In *Proceedings UIST'97: ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 139–148, Banff, Alberta, Canada, 1997.

[44] Keith W. Edwards, Elizabeth D. Mynatt, Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, and Marvin M. Theimer. Designing and implementing asynchronous collaborative applications with Bayou. In *Proceedings of the Tenth ACM Symposium on User Interface Software and Technology (UIST)*, pages 119–128, Banff, Alberta, Canada, October 1997.

[45] Christos Efstratio, Keith Cheverst, Nigel Davies, and Adrian Friday. Architectural requirements for the effective support of adaptive mobile applications. In *Proceedings of the Second International Conference on Mobile Data Management*, Hong Kong, January 2001.

[46] Domenico Ferrari, Jean Ramaekers, and Girgio Vente. Client-network interactions in quality of service communication environments. In *Proceedings of the 4th IFIP Conference on High Performance Networking*, Liege, Belgium, December 1992.

[47] Tom Fitzpatrick, Gordon Blair, Geoff Coulson, Nigel Davies, and Philippe Robin. Supporting adaptive multimedia applications through open bindings. In *Proceedings of the International Conference on Configurable Distributed Systems (ICCDS '98)*, Annapolis, Maryland, May 1998.

[48] Jason Flinn, Eyal de Lara, Mahadev Satyanarayanan, Dan. S. Wallach, and Willy Zwaenepoel. Reducing the energy usage of Office applications. In *Proceedings of IFIP/ACM International Conference on Distributed Systems Platforms*

(Middleware), Heidelberg, Germany, November 2001.

[49] Jason Flinn and Mahadev Satyanarayanan. Energy-aware adaptation for mobile applications. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, Kiawah Island Resort, South Carolina, December 1999.

[50] Sally Floyd and Van Jacobson. Link-sharing and resource management models for packet networks. *IEEE/ACM Transactions on Networking*, 3(4):365–386, August 1995.

[51] Brian Ford and Sai Susarla. CPU inheritance scheduling. In *Proceedings of the Second USENIX Symposium on Operating System Design and Implementation (OSDI)*, Berkeley, California, November 1996.

[52] George H. Forman and John Zahorjan. The challenges of mobile computing. *IEEE Computer*, pages 38–47, April 1994.

[53] Armando Fox, Steven D. Gribble, Eric A. Brewer, and Elan Amir. Adapting to network and client variability via on-demand dynamic distillation. *SIGPLAN Notices*, 31(9):160–170, September 1996.

[54] Armando Fox, Steven D. Gribble, Yatin Chawathe, and Eric A. Brewer. Adapting to network and client variation using infrastructural proxies: Lessons and perspectives. *IEEE Personal Communications*, 5(4):10–19, August 1998.

[55] Julian Gallop, Christopher Cooper, Ian Johnson, David Duce, Gordon Blair, Geoff Coulson, and Tom Fitzpatrick. Structuring for extensibility - adapting the past to fit the future. In *Proceedings of The CSCW2000 workshop on Component-based Groupware*, Philadelphia, Pennsylvania, December 2000.

[56] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Performance evaluations of memory consistency models for shared-memory multiprocessors. In *Proceedings of the 4th Symposium on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, California, April 1991.

[57] David K. Gifford. Weighted voting for replicated data. In *Proceedings of the 7th Symposium on Operating System Principles*, Pacific Grove, California, December 1979.

[58] Pawan Goyal, Xingang Guo, and Harrick M. Vin. A hierarchical CPU scheduler for multimedia operating system. In *Proceedings of the Second USENIX Symposium on Operating System Design and Implementation (OSDI)*, Seattle, Washington, October 1996.

[59] C. Greenhalgh, J. Purbrick, and D. Snowdon. Inside Massive-3: Flexible support for data consistency and world structuring. In *Proceedings of the Third International Conference on Collaborative Virtual Environments*, San Francisco, California, September 2000.

[60] Rajarshi Gupta. WebTP: A receiver-driven Web transport protocol. Master's thesis, University of California, at Berkeley, 1998. `http://www.path.berkeley.edu/~guptar/webtp/`.

[61] Richard G. Guy, John S. Heidemann, Wai Mak, Thomas W. Page, Gerald J. Popek, and Dieter Rothmeier. Implementation of the Ficus replicated file system. In *USENIX Conference Proceedings*, pages 63–71, Anaheim, California, June 1990. USENIX.

[62] J. S. Heidemann, T. W. Page, R. G. Guy, and G. J. Popek. Primarily disconnected operation: Experiences with Ficus. In *Proceedings of the Second Workshop on Management of Replicated Data*, pages 2–5, Monterey, California, November 1992.

[63] Maurice Herlihy. A quorum-consensus replication method for abstract data types. *ACM Transactions on Computer Systems*, 4(1):32–53, February 1986.

[64] John H. Howard. An overview of the Andrew file system. In *Proceedings of the USENIX Winter 1988 Technical Conference*, pages 23–26, Berkeley, California, 1988.

[65] John H. Howard. Using reconciliation to share files between occasionally connected computers. In *In Fourth Workshop on Workstation Operating Systems (WWOS-IV)*, Napa, California, October 1993.

[66] Larry B. Huston and Peter Honeyman. Disconnected operation for AFS. In *Proceedings of the USENIX Mobile and Location- Independent Computing Symposium*, Cambridge, Massachusetts, August 1993.

[67] Larry B. Huston and Peter Honeyman. Partially connected operation. *Computing Systems*, 8(4):365–379, 1995.

[68] Ayal Itzkovitz and Assaf Schuster. Multiview and Millipage – fine-grain sharing in page-based DSMs. In *Proceedings of the Third USENIX Symposium on Operating System Design and Implementation*, New Orleans, Louisiana, February 1999.

[69] Michael B. Jones, Paul J. Leach, Richard P. Draves, and Joseph S. Barbera III. Support for user-centric modular real-time resource management in the Rialto operating system. In *Proceedings of the 5th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSS-DAV)*, Durham, New Hampshire, April 1995.

[70] Bo Norregaard Jorgensen, Eddy Truyen, Frank Matthijs, and Wouter Joosen. Customization of object request brokers by application specific policies. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, New York, New York, April 2000.

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

[72] Anthony D. Joseph, Joshua A. Tauber, and M. Frans Kaashoek. Building reliable mobile-aware applications using the Rover toolkit. In *Proceedings of*

the 2nd ACM International Conference on Mobile Computing and Networking (MobiCom '96), Rye, New York, November 1996.

[73] Anthony D. Joseph, Joshua A. Tauber, and M. Frans Kaashoek. Mobile computing with the Rover toolkit. *IEEE Transactions on Computers*, 46(3):337–352, March 1997.

[74] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, 1988.

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

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

[77] Pete Keleher. *Distributed Shared Memory Using Lazy Release Consistency*. PhD thesis, Rice University, December 1994. Appeared as Rice Technical Report RICE COMP-TR-240 and available by anonymous ftp from cs.rice.edu under public/TreadMarks/papers.

[78] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, Queensland, Australia, May 1992.

[79] Pete Keleher, Sandhya Dwarkadas, Alan L. Cox, and Willy Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the 1994 Winter Usenix Conference*, pages 115–131, San Francisco, California, January 1994.

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

[81] Michael Koch and Jurgen Koch. Application of frameworks in groupware - The Iris group editor environment. *ACM Computing Surveys*, 32(1es), March 2000.

[82] Rainer Kosner and Thorsten Kramp. Structuring QoS-supporting services with smart proxies. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, New York, New York, April 2000.

[83] Puneet Kumar. Coping with conflicts in an optimistically-replicated file system. In *Proceedings of the IEEE Workshop on Management of Replicated Data*, Houston, Texas, November 1990.

[84] Puneet Kumar and Mahadev Satyanarayanan. Supporting application-specific resolution in an optimistically replicated file system. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*, Napa, California, October 1993.

[85] Puneet Kumar and Mahadev Satyanarayanan. Flexible and safe resolution of file conflicts. In *Proceedings of the USENIX Winter 1995 Technical Conference*, New Orleans, Louisiana, January 1995.

[86] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[87] Yui-Wah Lee, Kwong-Sak Leung, and Mahadev Satyanarayanan. Operation-based update propagation in a mobile file system. In *Proceedings of the USENIX Annual Technical Conference*, Monterrey, California, June 1999.

[88] Eliezer Levy and Abraham Silberschatz. Distributed file systems: Concepts and examples. *ACM Computing Surveys*, 22(4):321–374, December 1990.

[89] Baochun Li and Klara Nahrstedt. Qualprobes: Middlewate QoS profiling services for configuring adaptive applications. In *IFIP/ACM International Con-*

ference on Distributed Systems Platforms (Middleware), New York, New York, April 2000.

[90] Kang Li, Jonathan Walpole, Dylan McNamee, Calton Pu, and David C. Steere. A rate-matching packet scheduler for real-rate applications. In *Proceedings of the 2001 Conference on Multimedia Computing and Networking (MMCN'01)*, San Jose, California, January 2001.

[91] Radu Litiu and Atul Prakash. Developing adaptive groupware applications using a mobile component framework. In *Proceedings of the ACM 2000 Conference on Computer Supported Cooperative Work (CSCW 2000)*, Philadelphia, Pennsylvania, December 2000.

[92] Songwu Lu, Vaduvur Bharghavan, and Rayadurgam Srikant. Fair scheduling in wireless packet networks. *IEEE/ACM Transactions on Networking*, 7(4):473–489, 1999.

[93] Cliff Martin, P.S. Narayanan, Banu Ozden, Rajeev Rastogi, and Avi Silberschatz. The Fellini multimedia storage server. In S.M. Chung, editor, *Multimedia Information Storage and Management*. Kluwer Academic Publishers, 1996.

[94] Malena Mesarina and Yoshio Turner. Reduced energy decoding of MPEG streams. In *Proceedings of the 2002 Conference on Multimedia Computing and Natworking (MMCN'02)*, San Jose, California, January 2002.

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

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

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

[98] Lily B. Mummert, Maria R. Ebling, and Mahadev Satyanarayanan. Exploiting weak connectivity for mobile file access. In *Proceedings of the 15th ACM Sym-*

posium on Operating Systems Principles (SOSP), Copper Mountain, Colorado, December 1995.

[99] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, Banff, Canada, October 2001.

[100] Brad A. Myers, Robert C. Miller, Benjamin Bostwick, and Carl Evankovich. Extending the Windows desktop interface with connected handheld computers. In *Proceedings of the 4th USENIX Windows Systems Symposium*, Seattle, Washington, August 2000.

[101] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the Sprite network file system. *ACM Transctions on Computer Systems*, 6(1), February 1988.

[102] Brian D. Noble and Mahadev Satyanarayanan. A research status report on adaptation for mobile data access. In *SIGMOD Record*, volume 24, December 1995.

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

[104] Gary J. Nutt, Scott Brandt, Adam J. Griff, Sam Siewert, Marly Humphrey, and Toby Berk. Dynamically negotiated resource management for data intensive application suites. *Knowledge and Data Engineering*, 12(1):78–95, 2000.

[105] John K. Ousterhout, Herve Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A trace-driven analysis of the Unix 4.2 BSD file system. In *Proceedings of the 10th ACM Symposium on Operating System Principles*, Orcas Island, Washington, December 1985.

[106] Francois Pacull, Alain Sandoz, and Andre Schiper. Duplex: A distributed collaborative editing environment in large scale. In *Proceedings of the ACM*

Conference on Computer Supported Cooperative Work (CSCW), pages 165–173, Chapel Hill, North Carolina, October 1994.

[107] T. W. Page, R. G. Guy, J. S. Heidemann, D. H. Ratner, P. L. Reiher, A. Goel, G. H. kuenning, and G. J. Popek. Perspectives on optimistically replicated, peer-to-peer filing. *Software–Practice and Experience*, 2(28):155–180, February 1998.

[108] Abhay K. Parekh and Robert G. Gallager. A generalized processor sharing approach to flow control - the single node case. In *Proceedings of the IEEE INFOCOM'92 Conference on Computer Communications*, Florence, Italy, May 1992.

[109] Abhay K. Parekh and Robert G. Gallager. Processor sharing approach to flow control in integrated services networks - the multiple node case. *IEEE/ACM Transactions on Networking*, 2(2):137–150, April 1994.

[110] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. *Operating Systems Review (ACM)*, 31(5):288–301, December 1997.

[111] Gerald Popek, Bruce Walker, Johanna Chow, David Edwards, Charles Kline, Gerald Rudisin, and Greg Thiel. LOCUS a network transparent, high reliability distributed system. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, Pacific Grove, California, December 1981.

[112] Gerald J. Popek, Richard G. Guy, Thomas W. Page, Jr., and John S. Heidemann. Replication in Ficus distributed file systems. In *Proceedings of the Workshop on Management of Replicated Data*, pages 20–25, Houston, Texas, November 1990.

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

[114] Vincent Quint and Irène Vatton. Making structured documents active. *Electronic Publishing*, 7(2):55–74, June 1994.

[115] Suchitra Raman, Hari Balakrishnan, and Murari Srinivasan. An image transport protocol for the Internet. In *Proceedings of the 8th International Conference on Network Protocols (ICNP)*, Osaka, Japan, November 2000.

[116] Peter Reiher, John Heidemann, David Ratner, Gregory Skenner, and Gerald J. Popek. Resolving file conflicts in the Ficus file system. In *Proceedings of the Summer USENIX Conference*, pages 183–195, Boston, Massachusetts, June 1994.

[117] Luigi Rizzo. DummyNet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1):13–41, January 1997.

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

[119] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the SUN network file system. In *Proceedings of the 1985 Summer Usenix Conference*, pages 119–130, Portland, Oregon, June 1985.

[120] Martina Angela Sasse, Mark James Handley, and Shaw Cheng Chuang. Support for collaborative authoring via email: The MESSIE environment. In *Proceedings of 3rd European Conference on Computer Supported Cooperative Work*, pages 249–264, Milan, Italy, sep 1993.

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

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

[123] Mahadev Satyanarayanan, Jason Flinn, and Kevin R. Walker. Visual proxy: Exploiting OS customizations without application source code. *Operating Systems Review*, 33(3):14–18, July 1999.

[124] Mahadev Satyanarayanan, James J. Kistler, Lily B. Mummert, Maria R. Ebling, Puneet Kumar, and Qi Lu. Experience with disconnected operation in a mobile environment. In *Proceedings USENIX Symposium on Mobile and Location-Independent Computing*, pages 11–28, Boston, Massachusetts, August 1993.

[125] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. Shasta: A low overhead software-only approach for supporting fine-grain shared memory. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, Cambridge, Massachusetts, October 1996.

[126] Marc Shapiro, Antony Rowstron, and Anne-Marie Kermarrec. Application-independent reconciliation for nomadic applications. In *Proceedings of the 9th ACM SIGOPS European Workshop*, Kolding, Denmark, September 2000.

[127] Prashant Shenoy and Harrik M. Vin. Cello: A disk scheduling framework for next generation operating systems. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Madison, Wisconsin, June 1998.

[128] Prashant J. Shenoy, Pawan Goyal, Sriram S. Rao, and Harrik M. Vin. Symphony: An integrated multimedia file system. Technical report, Department of Computer Sciences, Univ. of Texas at Austin, March 1997.

[129] Prashant J. Shenoy, Pawan Goyal, and Harrick M. Vin. Architectural considerations for next generation file systems. In *Proceedings of the ACM Multimedia'99*, pages 457–468, Orlando, Florida, November 1999.

[130] N. T. Spring, M. Chesire, M. Berryman, V. Sahasranaman, T. Anderson, and B. N. Bershad. Receiver based management of low bandwidth access links. In *Proceedings of IEEE INFOCOM*, Tel-Aviv, Israel, March 2000.

[131] Peter Steenkiste. Adaptation models for network-aware distributed computations. In *Proceedings of 3rd Workshop on Communication, Architecture,*

and Applications for Network-based Parallel Computing (CANPC'99), Orlando, Florida, January 1999.

[132] Robert Stets, Sandhya Dwarkadas, Nikolaos Hardavellas, Galen C. Hunt, Leonidas I. Kontothanassis, Srinivasan Parthasarathy, and Michael L. Scott. Cashmere-2L: Software coherent shared memory on a clustered remote write network. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 170–183, Saint-Malo, France, October 1997.

[133] Randall R. Stewart, Qiaobing Xie, Ken Morneault, Chip Sharp, Hanns J. Schwarzbauer, Tom Taylor, Ian Rytina, Malleswar Kalla, Lixia Zhang, and Vern Paxson. Stream control transmission protocol. RFC 2960, October 2000. `http://www.ietf.org/rfc/rfc2960.txt`.

[134] Ion Stoica, Hui Zhang, and T.S. Eugene Ng. A hierarchical fair service curve algorithm for link-sharing, real-time and priority services. In *Proceedings of ACM SIGCOMM '97*, Cannes, France, September 1997.

[135] Vijay Sundaram, Abhishek Chandra, Pawan Goyal, Prashant J. Shenoy, Jasleen Sahni, and Harrick M. Vin. Application performance in the QLinux multimedia operating system. In *Proceedings of the Eighth ACM Conference on Multimedia*, pages 127–136, Los Angeles, California, November 2000.

[136] Eduardo Takahashi, Peter Steenkiste, Jun Gao, and Allan Fisher. A programming interface for network resource management. In *Proceedings of the Second IEEE Conference on Open Architectures and Network Programming (OPENARCH'99)*, New York, New York, March 1999.

[137] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, pages 172–183, Cooper Mountain, Colorado, December 1995.

[138] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A scalable distributed file system. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 224–237, Saint-Malo, France, October 1997.

[139] Robert H. Thomas. A majority consensus approach for concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, June 1979.

[140] Eddy Truyen, Wouter Joosen, Bart Vanhaute, Pierre Verbaeten, and Bo N. Jorgensen. A dynamic customization model for distributed component-based systems. In *Proceedings of the International Workshop on Distributed Dynamic Multiservice Architectures*, Phoeniz, Arizona, April 2001.

[141] Maarten van Steen, Philip Homburg, and Andrew S. Tanenbaum. Globe: A wide-area distributed system. *IEEE Concurrency*, 7(1):70–78, January 1999.

[142] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The LOCUS distributed operating system. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, Bretton Woods, New Hampshire, October 1983.

[143] E. James Whitehead and Yaron Y. Goland. WebDAV: A network protocol for remote collaborative authoring on the Web. In *Proceedings of the Sixth European Conference on Computer Supported Cooperative Work (ECSCW'99)*, pages 291–310, Copenhagen, Denmark, September 1999.

[144] Geoffrey G. Xie and Simon S. Lam. An efficient network architecture motivated by application-level QoS. *Journal of High Speed Networking*, 6(3):165–179, January 1998.

[145] Mark Yarvis, Peter Reiher, and Gerald Popek. Conductor: A framework for distributed adaptation. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems (HotOS-VII)*, Rio Rico, Arizona, March 1999.

[146] Bruce Zenel and Dan Duchamp. General purpose proxies: Solved and unsolved problems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, Cape Cod, Massachusetts, May 1997.

[147] Lixia Zhang, Steve Deering, Deborah Estrin, Scott Shenker, and Daniel Zappala. RSVP: A new resource ReSerVation Protocol. *IEEE Network*, 7(5):8–18, September 1993.

[148] Yuanyuan Zhou, Liviu Iftode, and Kai Li. Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems. In *Proceedings of the 2nd USENIX Symposium on OS Design and Implementation (OSDI '96)*, pages 75–88, Seattle, Washington, November 1996.