HATS: Hierarchical Adaptive Transmission Scheduling for Multi-Application Adaptation

Eyal de Lara[a], Dan S. Wallach[b], and Willy Zwaenepoel[b]

[a]Electrical and Computer Engineering, Rice University, Houston, US
[b]Computer Science, Rice University, Houston, US

ABSTRACT

Applications running on a mobile and wireless device most be able to adapt gracefully to limited and fluctuating network resources. When multiple applications run on the same device, control over bandwidth scheduling becomes fundamental in order to coordinate the adaptation of the multiple applications. This paper presents the design and evaluation of our Hierarchical Adaptive Transmission Scheduler (HATS), which adds control over bandwidth scheduling to the Puppeteer adaptation system. HATS enables the implementation of system-wide adaptation policies that can drastically improve the user's perception of network performance.

1. INTRODUCTION

It is widely recognized that network bandwidth is the limiting resource for mobile and wireless devices,[1–4] and that applications running on these devices must be able to gracefully adapt to limited and fluctuating network resources. This paper describes a system capable of adapting 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. In our earlier work, we have introduced *component-based adaptation*,[5] a new approach to adaptation that takes advantage of these APIs and document formats. We have demonstrated that component-based adaptation is an effective technique that supports powerful adaptation policies without requiring modifications to the applications. In this paper, we demonstrate 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

Further author information: (Send correspondence to E.d.L.)
E.d.L.: E-mail: delara@rice.edu, WWW: www.cs.rice.edu/~delara
D.S.W.: E-mail: dwallach@rice.com, WWW: www.cs.rice.edu/~dwallach
W.Z.: E-mail: willy@rice.edu, WWW: www.cs.rice.edu/~willy

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.

In this paper we present the design, implementations, and evaluation of our Hierarchical Adaptive Transmission Scheduler (HATS). HATS is an extension of the Puppeteer[5] component-based adaptation system. HATS 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.

HATS is limited to scheduling data flowing into the bandwidth-limited client. At present, Puppeteer is limited to adapting applications for browsing over bandwidth-limited links. Accordingly, in the current Puppeteer implementation, the vast majority of the data flows into the client, and just a small amount of control data (mainly requests for components) flows out of the client. Our future plans, however, call for extending Puppeteer to support document editing, and for HATS to schedule outgoing flows.

While earlier systems have implemented individual facilities described here,[6–14] 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 paper is organized as follows. Section 2 describes the Puppeteer adaptation system. Section 3 presents our implementation of HATS. Section 4 evaluates the performance of three system-wide adaptation policies. Section 5 describes previous work in bandwidth scheduling and how HATS differs from these earlier efforts. Finally, Section 6 discusses our conclusions and directions for future work.

2. PUPPETEER

Puppeteer is a system that implements bandwidth adaptation policies in mobile environments. It uses the exported APIs of component-based applications, such as Microsoft Office or Sun's OpenOffice, and the structured nature of the documents they manipulate to implement adaptation without changing the code of the application. It supports subsetting adaptations (in which only a subset of the elements of a document are provided to the application, for example, the first page), and versioning adaptations (in which a different version of some of the elements of a document is provided to the application, for example, a low-fidelity version of an image). Puppeteer adapts applications by extracting subsets and versions from documents. Puppeteer uses the exported APIs of the applications to incrementally increase the subset of the document or improve the version of the elements available to the application. For example, it uses the exported API to insert additional pages or higher-fidelity images into the application.

Figure 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 unaware of Puppeteer. Data servers can be arbitrary repositories of data such as Web servers, file servers, or data bases. All communication between the application(s) and the data server(s) go through the Puppeteer local and remote proxies that work together to perform the adaptation. The Puppeteer local proxy runs on the bandwidth-limited device and manipulates the running application through a subset of the application's exported API. The Puppeteer remote proxy runs on the other side of the bandwidth-limited link, at or close to the access point, and is assumed to have high-bandwidth and low-latency connectivity (relative to the bandwidth-limited device) to the data servers.

Figure 2 shows the architecture of the Puppeteer local and remote proxies. It consists of application-specific adaptation policies, component-specific drivers, type-specific transcoders and an application-independent kernel.

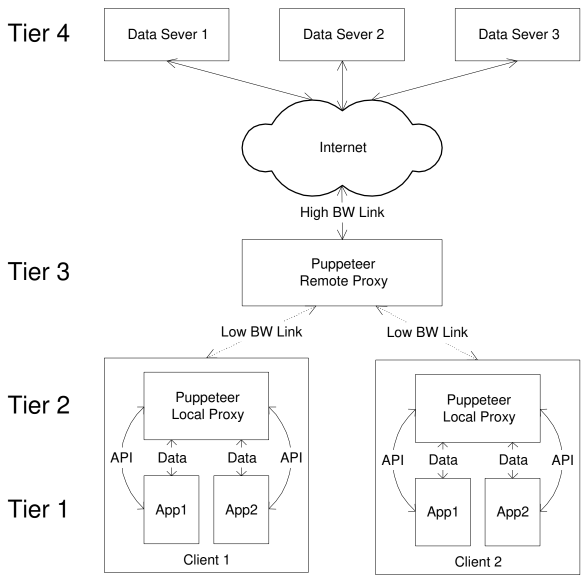


Figure 1: Puppeteer architecture.

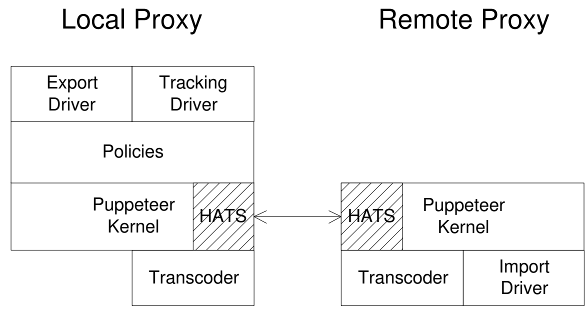


Figure 2. Puppeteer remote- and local-proxy architectures. HATS runs in the kernel of both proxies and manages all data transmission.

The Puppeteer local proxy is in charge of executing adaptation policies. The Puppeteer remote proxy is responsible for parsing documents, exposing their structure, and transcoding components as requested by the local proxy.

The adaptation process in Puppeteer is divided roughly into three stages: parsing the document to uncover the structure of the data, fetching selected components at specific fidelity levels, and updating the application with the newly fetched data.

When the user opens a document, the Puppeteer remote proxy instantiates an *import driver* for the appropriate document type. The import driver parses the document, and extracts its *skeleton* (component structure, a tree) and the data associated with the nodes in the tree. Puppeteer then transfers the document's skeleton to the Puppeteer local proxy. The adaptation policies running on the local proxy fetch an initial set of elements from within the skeleton at a specified fidelity. These adaptation policies may be static, or may depend on *tracking drivers* that detect the occurrence of certain events, such as moving the mouse over an image, causing the image to be loaded.

When the initial set of components is received by the local proxy, Puppeteer uses the *export driver* for the particular document type to supply this set of components to the application as though it had the full document at its highest fidelity level. The application, believing that it has finished loading the document, returns control to the user. Meanwhile, the adaptation policy running at the local proxy knows that only a fraction of the document has been loaded and uses the application's exported API to incrementally supply remaining components or upgrade their fidelity.

3. HATS

In this section we first discuss possible implementations of HATS on Puppeteer. We then describe our remote proxy-based HATS implementations.

3.1. 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. We next describe 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. We 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, we 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 waist 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. We chose the remote proxy-based implementation over the local proxy-based approach because we want to provide for bandwidth sharing with low jitter. While it is widely accepted that low jitter is important for streaming media (which we 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),[15] 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, we felt that, for our initial implementation, a fixed number of transmission strategies , provided by the remote proxy, were sufficient. We discuss our remote proxy-based HATS implementation in detail in the following section.

3.2. 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 3 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.

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

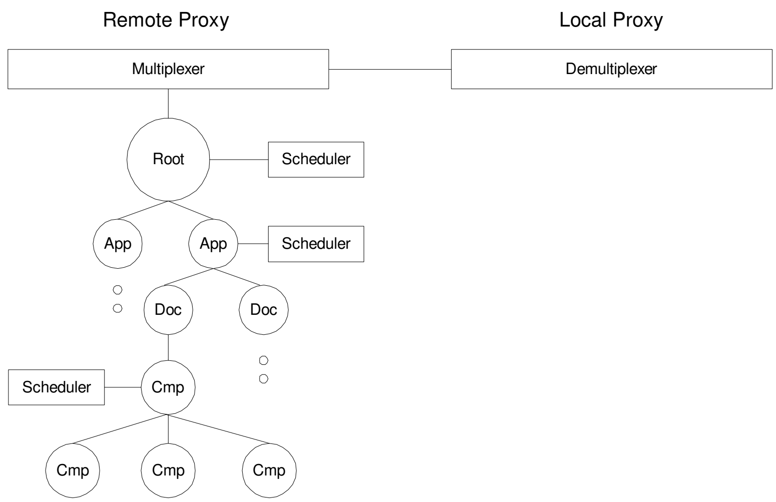


Figure 3. Remote proxy-based HATS implementation. The architecture consist of schedulers, multiplexing and demultiplexing facilities, and a hierarchy of applications (App), documents (Doc), and components (Cmp).

3.2.1. 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 (when they become leafs).

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.[6,10,14,16]

3.2.2. 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 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, we could 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.

3.2.3. 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 we can reallocate bandwidth is limited by the OS transmission buffers, which we 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 4.1.2.

3.2.4. 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.

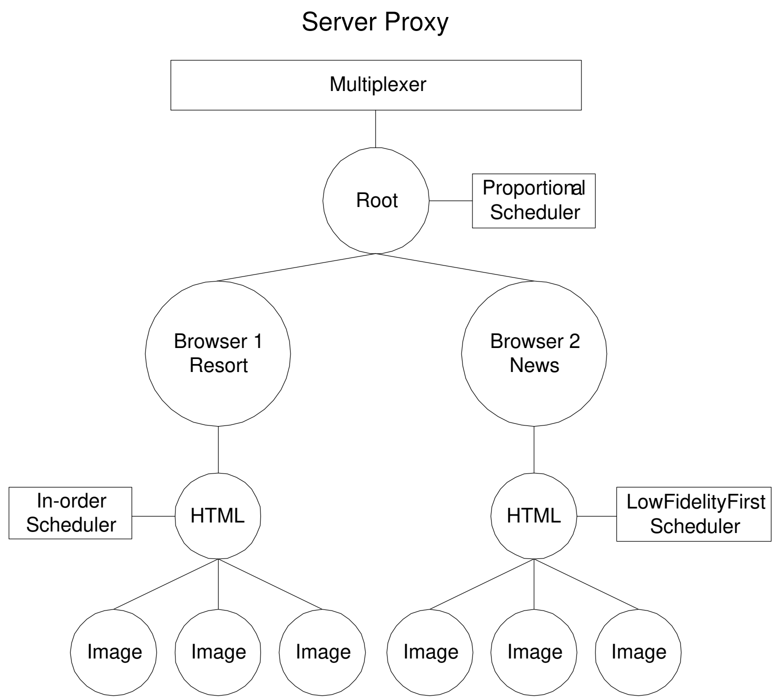


Figure 4: Transmission hierarchy for loading to web pages.

3.2.5. Example

To illustrate the HATS mechanisms let us consider a user simultaneously loading two Web pages in two browsers as depicted in Figure 4. 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, we use a system-wide adaptation policy that installs 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). We assume that images have 3 fidelity levels, which we 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 images 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.

4. EXPERIMENTAL EVALUATION

In this section we quantify 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. Our purpose is not to determine the best adaptation policies for the applications we adapt. Instead, we want to illustrate the variety of system-wide adaptation policies made possible by HATS.

We quantify the performance of our implementation by loading documents using Internet Explorer 5.5 (IE5) and PowerPoint (PPT) on a platform consisting of three Pentium III 500 MHz machines running Windows 2000. We configured these machines as: a data server running Apache 1.3, which stores all the documents we use in our 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.[17] This setup allows us 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.

4.1. HATS-based adaptation policies

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

We run these experiments using a 256 byte HATS scheduling quantum. We 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. We examine the overhead incurred by the small quantum size in Section 4.2.

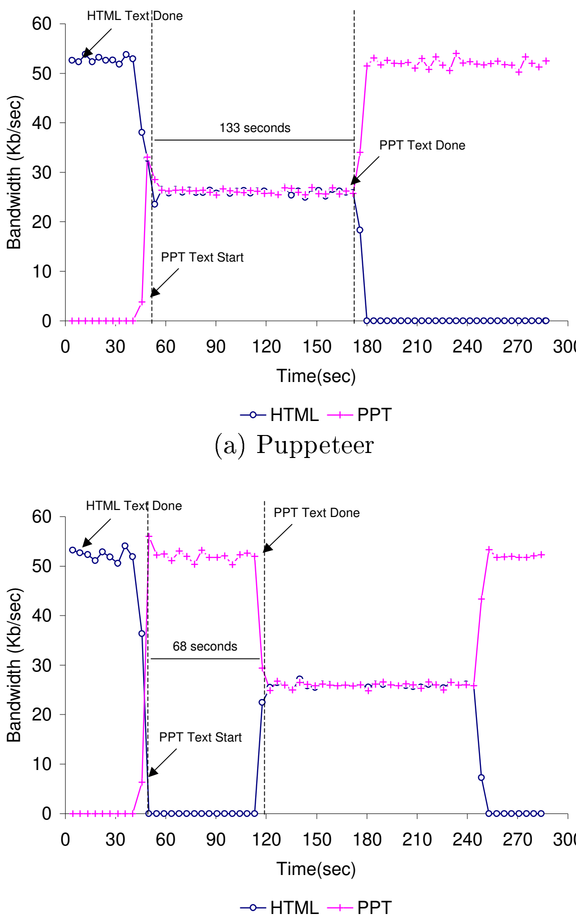
4.1.1. Remote-proxy-based reconfiguration

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 In this experiment we compare the time to load text-only versions of a HTML page and a PPT presentation. For each application we 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), we 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).

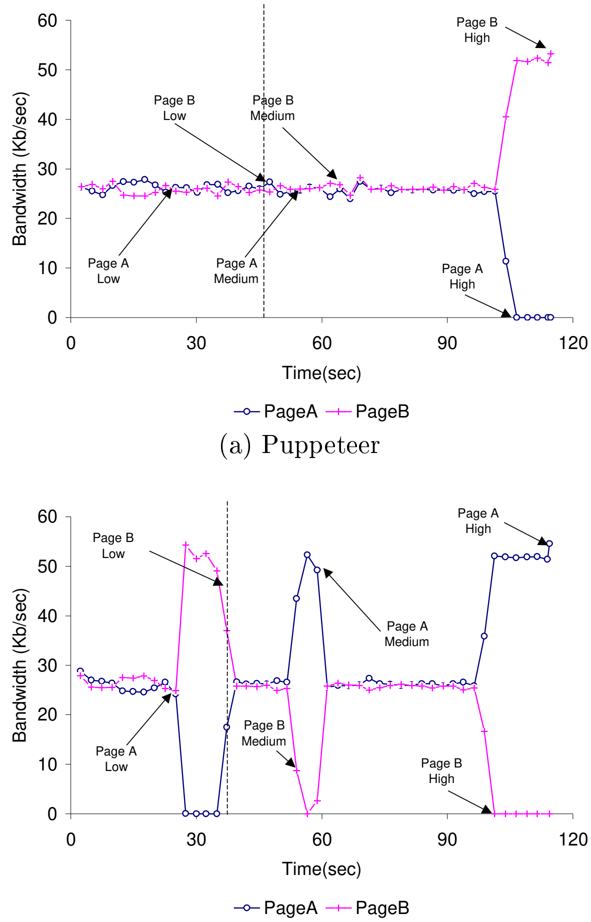
Figures 5(a) and (b) show the bandwidth allocations for loading an image-rich 668 KB HTML document and a 1.05 MB PowerPoint presentation over a 56 Kb/sec network link with Puppeteer and Puppeteer-HATS. For these documents, HATS achieves savings of 49%, reducing the time to load a text-only version of 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 reallocating bandwidth from sending images embedded in the HTML document to sending PPT slides.



(a) Puppeteer

(b) Puppeteer-HATS

Figure 5. 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.



(a) Puppeteer

(b) Puppeteer-HATS

Figure 6. 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.

Low fidelity first In this experiment, we 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. We assume that images have 3 fidelity levels, which we will refer to as *low*, *medium*, and *high*. We chose these levels to correspond to the first 1/7, next 2/7, and last 4/7 of the bytes in the progressive JPEG representation. 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 we 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.

We 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, we transmit images sequentially. When an image reaches a target fidelity level, we start serving the next image in the page.

We were 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 (our implementation only triggers an application update when we reach 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, we choose to present our results using an in-order transmission strategy, which achieves better overall performance.

Figures 6(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, on 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(b) shows that with Puppeteer, 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(b) also show that with the LowFidelityFirst dynamic scheduler *PageB* reaches medium and high fidelity before *PageA*. The precise opposite of what happens in Figure 6(a). The inversion of times to reach medium and high fidelity are an artifact of the adaptation policy and the pages we use in this experiment. The adaptation policy only transcode images to progressive JPEG if they are bigger than 4KB in size. Loading low fidelity, loads all images smaller than 4KB and the first 1/7 of all transcoded. In this experiment *PageB* has more small images that are not transcoded than *PageA*. This is why it takes *PageB* more time to reach low fidelity, but less time to reach medium and high fidelity.

4.1.2. 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 reallocates bandwidth to the components of foreground application or document. In this experiment we load two different HTML pages simultaneously in two separate instances of IE5. We use 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

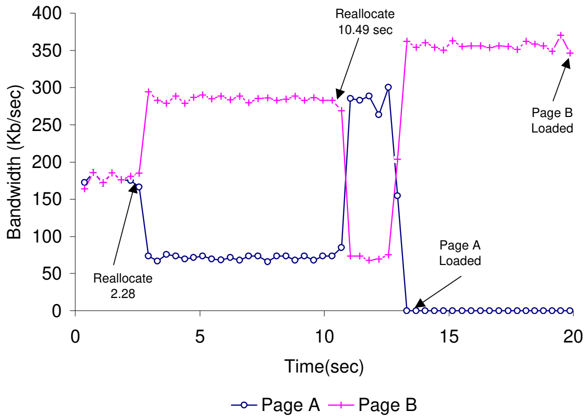


Figure 7. Bandwidth allocations for transmitting two HTML documents with *Focus*, an adaptation policy that reallocates 80% of the available bandwidth to the foreground document.

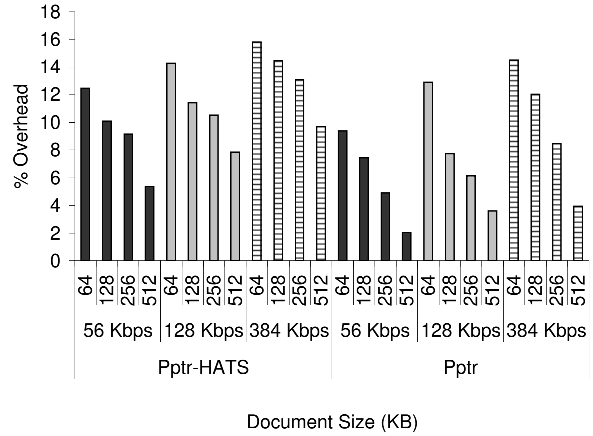


Figure 8. Overhead for loading HTML documents of various sizes on Puppeteer (Pptr), and a Puppeteer-based HATS implementation (Pptr-HATS) over 56 Kb/sec, 128 Kb/sec, and 384 Kb/sec network links. Pptr-HATS quantum is set to 256 bytes. All times are normalized by time to load documents in native IE5, 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.

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. For this experiment we use a scheduler that implements the WF^2Q+ Packet Fair Queuing (PFQ) algorithm,[18] which distributes bandwidth among the nodes it service according to their rate allocations.

Figure 7 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.

We measured the time it takes for a bandwidth reallocation to take effect, from the time it is requested by the local proxy until the bandwidth allocation for the documents stabilizes again. We refer to this time as reallocation latency. We measured an average reallocation latency of 1.38 and 0.24 seconds for runs on 56 Kb/sec and 384 Kb/sec network links, respectively. The reallocation 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 reallocation request. We experimented with increasing the network latency in DummyNet. Unsurprisingly, the reallocation latency increased by exactly the amount of extra latency introduced by DummyNet (e.g., 2.38 seconds over 56 Kb/sec when we 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.

4.2. System overhead

We measured the worst case HATS's latency overhead by using IE5 to load a set of HTML documents downloaded from the Web. We load the documents without doing any adaptations or using any dynamic transmission strategy, using Puppeteer and our Puppeteer-based HATS implementation. Figure 8 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 proxy support.

This experiment represents the worst possible case for both Puppeteer and Puppeteer-HATS as we incur 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 3. For this experiment we configured the Puppeteer-HATS scheduling quantum to be 256 bytes.

Figure 8 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 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. Finally, 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 we implemented in the previous section had similar data overheads ranging between 4.6% to 4.8% extra data. The data overhead 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.

5. RELATED WORK

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

5.1. System-wide adaptation

Much work has gone into extending support for mobile clients. For example, Coda[19] and Odyssey[11] add system support for adaptation; Rover,[8] Visual Proxy,[13] and Fox *et al.*[7] create programming models that incorporate adaptation into the design of the application; LBFS[20] provides an optimized file system for bandwidth-limited clients; and Smart Proxies,[9] Jorgensen *et al.*,[21] and Fitzpatrick *et al.*[22] support tunable RPC implementations. Several groups[23–25] have suggested the need for centralized adaptation systems that implement system-wide adaptation policies. None of the these efforts, however, provide support for scheduling data transmission for applications and documents running on the client.

5.2. Network and server support for scheduling

Systems have been developed to enable clients to specify their QoS network requirements[26,27] and to provide differentiated service in network hierarchies.[18,28–31] 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.

Several projects[6,10,14,16] 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.

5.3. Client-centric scheduling

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

Rialto,[32] 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.

The WebTP[33] 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 browser downloading two separate pages).

Spring *et al.*[12] 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.

6. CONCLUSIONS

We demonstrated a system that is capable of adapting multiple applications running concurrently on a bandwidth-limited device. Specifically, we described the design and evaluated the performance of our Hierarchical Adaptive Transmission Scheduling (HATS), which we 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.

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.

Our goal in this paper was not to perform a comprehensive study of transmission strategies. Instead, we 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, we have shown that even simple policies that prioritize text over other component types can achieve savings of 49% in the time it takes to open text-only versions of large documents.

In future work, we intend to investigate scheduling of two-way traffic between the remote and local proxies, extending beyond our current support for read-only browsing to other services such as read-write network file systems and e-mail systems. We also plan to investigate using protocols other than TCP to communicate between the local and remote proxies, as other protocols may have better support for vertical hand-off and dynamic, low-latency, reprioritization of data streams.

REFERENCES

1. R. Bagrodia, W. W. Chu, L. Kleinrock, and G. Popek, "Vision, issues, and architecture for nomadic computing," *IEEE Personal Communications* **2**, pp. 14–27, Dec. 1995.
2. D. Duchamp, "Issues in wireless mobile computing," in *Proceedings of Third Workshop on Workstation Operating Systems*, pp. 1–7, (Key Biscayne, Florida), Apr. 1992.
3. R. H. Katz, "Adaptation and mobility in wireless information systems," *IEEE Personal Communications* **1**(1), pp. 6–17, 1994.
4. M. Satyanarayanan, "Fundamental challenges in mobile computing," in *Fifteenth ACM Symposium on Principles of Distributed Computing*, (Philadelphia, Pennsylvania), May 1996.
5. E. de Lara, D. S. Wallach, and W. Zwaenepoel, "Puppeteer: Component-based adaptation for mobile computing," in *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems*, (San Francisco, California), Mar. 2001.
6. W. Almesberger, "Linux network traffic control - implementation overview," Apr. 1999. `http://lrcwww.epfl.ch/linux-diffserv/`.
7. A. Fox, S. D. Gribble, E. A. Brewer, and E. Amir, "Adapting to network and client variability via on-demand dynamic distillation," *SIGPLAN Notices* **31**, pp. 160–170, Sept. 1996.
8. A. D. Joseph, A. F. deLespinasse, J. A. Tauber, D. K. Gifford, and M. F. Kaashoek, "Rover: a toolkit for mobile information access," in *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, (Copper Mountain, Colorado), Dec. 1995.
9. R. Kosner and T. Kramp, "Structuring QoS-supporting services with smart proxies," in *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, (New York, New York), Apr. 2000.
10. K. Li, J. Walpole, D. McNamee, C. Pu, and D. C. Steere, "A rate-matching packet scheduler for real-rate applications," in *Proceedings of Multimedia Computing and Networking 2001*, (San Jose, California), Jan. 2001.
11. B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker, "Agile application-aware adaptation for mobility," *Operating Systems Review (ACM)* **51**, pp. 276–287, Dec. 1997.
12. 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), Mar. 2000.
13. M. Satyanarayanan, J. Flinn, and K. R. Walker, "Visual proxy: Exploiting OS customizations without application source code," *Operating Systems Review* **33**, pp. 14–18, July 1999.
14. V. Sundaram, A. Chandra, P. Goyal, P. Shenoy, J. Sahni, and H. Vin, "Application performance in the QLinux multimedia operating system," in *Proceedings of the Eighth ACM Conference on Multimedia*, (Los Angeles, California), Nov. 2000.
15. E. de Lara, D. S. Wallach, and W. Zwaenepoel, "Opportunities for bandwidth adaptation in Microsoft Office documents," in *Proceedings of the Fourth USENIX Windows Symposium*, (Seattle, Washington), Aug. 2000.
16. S. Chen and K. 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.
17. L. Rizzo, "DummyNet: a simple approach to the evaluation of network protocols," *ACM Computer Communication Review* **27**, pp. 13–41, Jan. 1997.
18. J. Bennett and H. Zhang, "Hierarchical packet fair queuing algorithms," in *Proceedings of SIGCOMM'96*, (Stanford, California), Aug. 1996.
19. L. Mummert, M. Ebling, and M. Satyanarayanan, "Exploiting weak connectivity for mobile file access," in *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, (Copper Mountain, Colorado), Dec. 1995.
20. A. Muthitacharoen, B. Chen, and D. Mazieres, "A low-bandwidth network file system," in *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, (Banff, Canada), Oct. 2001.

21. B. N. Jorgensen, E. Truyen, F. Matthijs, and W. 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), Apr. 2000.
22. T. Fitzpatrick, G. Blair, G. Coulson, N. Davies, and P. Robin, "Supporting adaptive multimedia applications through open bindings," in *Proceedings of the International Conference on Configurable Distributed Systems (ICCDS '98)*, (Annapolis, Maryland), May 1998.
23. C. Efstratio, K. Cheverst, N. Davies, and A. Friday, "Architectural requirements for the effective support of adaptive mobile applications," in *Proceedings of the Second International Conference on Mobile Data Management*, (Hong Kong), Jan. 2001.
24. B. Li and K. Nahrstedt, "Qualprobes: Middlewate QoS profiling services for configuring adaptive applications," in *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, (New York, New York), Apr. 2000.
25. G. J. Nutt, S. Brandt, A. J. Griff, S. Siewert, M. Humphrey, and T. Berk, "Dynamically negotiated resource management for data intensive application suites," *Knowledge and Data Engineering* **12**(1), pp. 78–95, 2000.
26. D. Ferrari, J. Ramaekers, and G. Vente, "Client-network interactions in quality of service communication environments," in *Proceedings of the 4th IFIP Conference on High Performance Networking*, (Liege, Belgium), Dec. 1992.
27. L. Zhang, S. Deering, and D. Estrin, "RSVP: A new resource ReSerVation Protocol," *IEEE Network* **7**, pp. 8–18, Sept. 1993.
28. S. Floyd and V. Jacobson, "Link-sharing and resource management models for packet networks," *IEEE/ACM Transactions on Networking* **3**, pp. 365–386, Aug. 1995.
29. S. Lu, V. Bharghavan, and R. Srikant, "Fair scheduling in wireless packet networks," *IEEE/ACM Transactions on Networking* **7**(4), pp. 473–489, 1999.
30. A. Parekh and R. 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.
31. I. Stoica, H. Zhang, and T. Ng, "A hierarchical fair service curve algorithm for link-sharing, real-time and priority services," in *Proceedings of ACM SIGCOMM '97*, (Cannes, France), Sept. 1997.
32. M. B. Jones, P. Leach, R. Draves, and J. B. 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 (NOSSDAV)*, (Durham, New Hampshire), Apr. 1995.
33. R. Gupta, "WebTP: A receiver-driven web transport protocol," Master's thesis, University of California, at Berkeley, 1998. `http://www.path.berkeley.edu/~guptar/webtp/`.