RICE UNIVERSITY

Extensible Adaptation via Constraint Solving

by

Yuri Dotsenko

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Master of Science

APPROVED, THESIS COMMITTEE:

Dr. Dan S. Wallach, Chair,
Assistant Professor of Computer Science

Dr. Willy Zwaenepoel,
Karl F. Hasselman Professor of Computer Science and Professor of Electrical and Computer Engineering

Dr. David B. Johnson,
Associate Professor of Computer Science and of Electrical and Computer Engineering

HOUSTON, TEXAS

APRIL, 2002

Extensible Adaptation via Constraint Solving

Yuri Dotsenko

Abstract

This work presents the design, implementation, and evaluation of a simple programming language for expressing scheduling policies for transmission of multiple objects across a shared network connection. A key design component of the language is the ability to express constraints among the objects to be transmitted. Policies can: make ordering constraints, such as "all text objects are transmitted before any image objects"; express rules on the relative bandwidth allocations across objects of different types; reserve a certain amount of bandwidth or restrict transmission of a subset of objects. Because it is possible to express contradictory constraints, the system finds suitable approximate solutions when no precise solution is available.

Acknowledgments

I would like to thank my advisors, Dr. Dan S. Wallach and Dr. Willy Zwaenepoel, and Dr. David B. Johnson for fruitful discussions and helpful advice. I'm grateful to Eyal de Lara for providing help with the Puppeteer system.

Contents

Illustrations

Tables

Chapter 1

Introduction

The ability to adapt to limited and varying resources has long been considered a fundamental requirement for mobile computing [BCKP95, Kat94, Sat96]. The advent of ubiquitous and pervasive computing [Wei93, Sat01] will only increase the need for adaptation as the services available to a pervasive client depend on the resources that are carried by the client and those that are provided by the smart space in which the client happens to be.

In the last 10 years, researchers have devised several mechanisms for adapting to variation in network connectivity [dWZ01, FGBA96, KK00, NSN+97], energy supply [FS99], and device heterogeneity [CSWK01, TMT01]. While these mechanisms have proven powerful, there has been little work on how to specify effective *policies* for using these mechanisms. Policy definition is a particularly hard problem, as users may require different behavior based on a variety of criteria, such as resource availability, physical location, cost, time of day, the mix of applications running on the device or smart space, or the presence or absence of other users in the smart space. Moreover, the space of possible adaptations is combinatorial in the number of services provided by the smart space, the number of applications concurrently running, and the number of possible configuration options of the applications and their data.

The large size of the adaptation space precludes the implementation of all-encompassing adaptation policies that cover all possible scenarios. Instead, it can be expected that users will have to be active participants in the adaptation process, teaching the pervasive system how to adapt when it encounters a new situation, or when the user is unhappy with the

system's current behavior.

For ordinary users to become policy designers, the process of defining policies must be simple and scalable. Unfortunately, there is a mismatch between the way users think about the tasks they want the pervasive system to perform and the inputs that current adaptation mechanisms expect. Whereas the user may think in terms of behavior or desirable results (e.g., "get the text of a document fast", "I care most about my MP3 player"), current adaptation mechanisms function in terms of low level configuration parameters (e.g., fidelity levels, scheduling shares, and priorities). Likewise, it would be desirable to provide mechanisms that might allow unsophisticated users, who cannot be expected to write even simple programs, to be able to compose and manipulate policies written by others.

This thesis presents Extensible Adaptation via Constraint Solving (EACS), a novel approach that significantly simplifies policy design by distancing policy writers and ordinary users from the intricacies of the adaptation mechanisms. In EACS, policy writers specify adaptation policies by defining subsets of data objects to transmit and defining constraints among these subsets. Less advanced users can compose new policies from several predefined ones.

The initial EACS prototype is limited to transmission policies, which define the order in which a series of objects is transmitted to a bandwidth-limited device. Users specify transmission policies by grouping data in transit to or from the mobile device into subsets based on the data's type, size, or any other attribute. Users can specify dependencies between the sets (e.g., "all elements of set A should be transferred before any element of set B") or set proportional bandwidth allocations for the sets (e.g., "elements of set C should get 3 times as much bandwidth as elements of set D"), and they can allocate (reserve) a fixed amount of bandwidth to a set or disable the transmission of a set permanently or temporarily.

First, I developed an EACS simulator, allowing simulation of real EACS policies with-

out actually transmitting data across a network. This allows a policy designer to quickly see how a policy behaves on real data without waiting for the data to traverse a low-speed network. Then, I integrated EACS with the Puppeteer adaptation system [dWZ01], and measured the overhead introduced by EACS policy interpreter, showing this overhead to be negligible. EACS proved effective at expressing complex transmission policies with very few lines of code, a significant improvement over HATS system [dWZ02], where policies are written in Java and are generally constrained to follow the hierarchical structure built into existing documents.

The rest of this thesis is organized as follows. Chapter 2 presents the design of the EACS policy language. Chapter 3 describes the semantics of and the process for transforming high-level EACS descriptions into low-level bandwidth shares that can be fed to an adaptation system. Chapter 4 presents results from simulations and runs on the Puppeteer adaption system. Chapter 5 discusses prior work and how EACS differs from these earlier efforts. Finally, Chapter 6 concludes the discussion and identifies the directions for future work.

Chapter 2

Policy Language Design

This chapter presents the design criteria behind EACS and a method for high-level, compact representation of transmission policies.

2.1 Transmission Policy Domain Overview

The goal of a bandwidth adaptation system is to improve latency for network operations by transmitting less data. To accomplish this, an adaptation system has three things it can do: it can choose to send a subset of the desired data (e.g., removing images from a document), it can transform the desired data (e.g., using lossy compression), or it can change the order in which data items are transferred (e.g., sending textual data before sending multimedia data). The EACS policy language is focused on data transmission ordering, assuming that other portions of the adaptation system have selected and transformed these data.

Below I present several transmission policies and scenarios based on my long-term experience with the Puppeteer system [dWZ01].

- Text-first: this is probably the most useful policy in a bandwidth-limited environment. For a document, e.g. an HTML page, the size of text objects is usually much smaller than that of images or multimedia data. Fetching text objects first allows a text-only version of the document to be rendered much faster than the entire document can be rendered. There are more analogous policies based on fetching a certain subset of data first. For example, for a user opening a PowerPoint presentation over

a slow connection, it would be beneficial to fetch (and render) the first slide or two first slides before fetching the rest of the presentation.

- Prefetching: this kind of policy is useful when the subset of data that the user would expect in the near future can be predicted. For example, a user working with slide 5 of a PowerPoint presentation most probably needs slide 4 or 6 next. Thus, the system should try to prefetch those before the user advances to the next page.

- Proportional bandwidth distribution: a user working with two applications that share a slow bandwidth link might give a preference to one of them, allocating $\frac{2}{3}$ of the total bandwidth. Each share can be split among the objects and components within the application. Another useful policy might allocate more, e.g., 80%, of the total bandwidth to the foreground application while splitting the rest among background applications. When another application becomes foreground, the system shifts bandwidth to it.

- Guaranteed allocation: a multimedia stream might require a certain fixed amount of bandwidth to operate properly. As a concrete example, if a 24 kbit/sec audio stream shares a 56 kbit/sec link with other connections, the system should reserve 24 kbit/sec of bandwidth for the audio stream to provide an adequate level of audio perception.

- Time-constrained policies: a policy of this kind can be useful to save money if the user pays for the transmission time. For example, if the user is browsing the web, the system can be told to spend no more than 30 seconds downloading an HTML page. This can result in poorer quality of the fetched page (e.g., some of the images are transcoded) but saves transmission time.

The next sections explain how to abstract these and more transmission policies and

formalizes the language to specify them.

2.2 Language Elements

The goal, in designing the language, is to present a high-level abstraction that hides as much detail as possible, while still making it possible to express interesting transmission policies. To solve this, I introduce two basic concepts: set operations and constraints on these sets. To define a transmission policy, the user first defines various subsets of the objects to be transmitted based on those objects' attributes. Then, constraints are made, saying which sets must go before others and which must be interleaved.

In order to express transmission policies, it is necessary to be able to build groups, *sets*, of the available objects based on their attributes such as *size*, *type* (text, image, sound), the time the object was added for transmission. Objects can have static attributes, such as what kind of application is reading the object, and dynamic attributes, such as whether the application requesting the object happens to be the user's foreground application. It is also possible to build sets by applying standard set operations: union, intersection, and difference to sets defined earlier.

After sets are defined, constraints between them customize the transmission of data. To express transmission policies, constraints can take many forms.

1) **Priority constraints** express how some objects must be transmitted before other objects can begin transmission (e.g., "send text objects first, then image objects").

2) **Proportional constraints** express how bandwidth should be shared among objects being transmitted concurrently (e.g., "Web browsers get 80% of the bandwidth and other types get 20%").

3) **Guaranteed-allocation constraints** reserve a requested amount of bandwidth to a

set of objects (e.g., an audio stream).

4) **Never-transmit constraints** prevent transmission of the objects in the specified sets.

5) **Hierarchical constraints** express precedence ordering for other constraints (e.g., "within Web browsers, text goes before images, but in other documents the bandwidth is shared").

2.3 Language Syntax

The EACS policy language borrows its syntax from the C language, although it is a much simpler language to evaluate. The operators supported include: variables and constants; if-then-else branches; function declaration and invocation; and arithmetic with integer, floating point, and boolean operators. Other supported primitive types are sets, object attributes (as discussed above), and constraints (which can be arguments to other constraints).

2.3.1 Sets

The main function to form new sets is

Set *result* = **select**(*set*, *expr*);

This function selects all the components of *set* for which *expr* evaluates to *true* and stores the result in the variable *result* of type Set.

The function to update the state (attribute) associated with each object is

void **update**(*set*, *attribute*, *expr*);

This function updates the attribute *attribute* of all the objects of *set* to the value of *expr* and returns nothing.

In addition, a number of other useful functions is defined:

- Set *result* = **min**(*set*, *expr*);

  This function selects the component of *set* for which *expr* has minimum value. Likewise, there is a **max** function.

- Set *result* = **get1**(*set*);

  This function randomly selects one element from *set* and stores the result in the variable *result* of type Set.

All the usual set operations are defined, including intersection (**AND**), union (**OR**), and difference (**SUB**). In addition, there are defined two special sets: Ω as the universe of all objects, and $\emptyset$ as the empty set.

In the context of an expression that selects elements from a set, some ephemeral variables are defined while evaluating the predicate *expr* that refer to the attributes of each object:

1) **type** describes what kind of object this is.

2) **application** describes the application requesting the object.

3) **sizeDone** is the number of bytes of the object that have been transmitted.

4) **sizeOriginal** is the size of the object.

5) **fg** is true if the object belongs to the foreground application, and is false otherwise.

6) **timeAddedToTransmission** is the absolute time at which the object was added for transmission.

7) **timeLastTransmitted** is the last absolute time at which a portion of the object was transmitted.

This list is not exhaustive and can be extended if new attributes are added.

2.3.2 Constraints

After sets are defined, a number of constraints can be added. This section describes the operators for the supported constraint types.

Priority Constraints

Given two sets of components s_1 and s_2, a policy writer can express priority constraints:

- **After**(s_1, s_2);
  all components of s_2 must be completely transmitted before any elements in s_1 can begin transmission.

- **Before**(s_1, s_2);
  equivalent to **After**(s_2, s_1).

Proportional Constraints

A policy writer can also express proportional constraints to describe how bandwidth must be shared between objects. There are two types of proportional constraints:

- **BandwidthRatio**(s_1, s_2, r);
  the ratio between the total bandwidth for components in set s_1 and the total bandwidth for components in set s_2 is equal to r. Elements within the same set would have the same bandwidth, assuming there are no other constraints.

- **BandwidthPerElementRatio**(s_1, s_2, r);
  each element of set s_1 has a bandwidth share that is r times more than the bandwidth share of each element of set s_2. Elements within the same set would have the same bandwidth, assuming there are no other constraints.

To explain the difference between **BandwidthRatio** and **BandwidthPerElementRatio**, consider the following example:

$$\begin{aligned} \Omega &= \{A, B, C\} \quad \textit{the universe has three objects} \\ s_1 &= \{A, B\} \\ s_2 &= \{C\} \end{aligned}$$

Also, let a, b, and c denote the respective bandwidth shares of the components in Ω.

The statement **BandwidthRatio**(s_1, s_2, 2) means that all of the following equations hold:

$$\begin{aligned} a + b &= 2c \\ a &= b \\ a + b + c &= 1 \end{aligned}$$

The first equation expresses that the bandwidth allocations a and b together (for the members of s_1) must be twice the bandwidth allocation c (for the sole member of s_2). The second equation expresses that allocations a and b must be the same, as they are in an equivalence class (s_1) with each other. The final equation, $a + b + c = 1$, says that all the available bandwidth must go somewhere, avoiding a degenerate solution, such as $a = b = c = 0$. The solution $a = b = c = \frac{1}{3}$ shows the desired bandwidth distribution.

For the same Ω and sets as in the example above, consider the statement:

BandwidthPerElementRatio(s_1, s_2, 2);

This yields a different set of equations:

$$\begin{aligned} a &= 2c \\ b &= 2c \\ a &= b \\ a+b+c &= 1 \end{aligned}$$

The first two equations express the rule that, for every pair of components from s_1 and s_2, the bandwidth ratio should be 2. The third and fourth rules are the same as above. The solution, $a = b = \frac{2}{5}$ and $c = \frac{1}{5}$, shows the desired bandwidth distribution.

BandwidthRatio and **BandwidthPerElementRatio** are related. For the same non-empty sets s_1 and s_2:

$$\begin{aligned} \mathbf{BandwidthPerElementRatio}(s_1, s_2, r) \equiv \\ \mathbf{BandwidthRatio}\left(s_1, s_2, r\frac{|s_1|}{|s_2|}\right) \end{aligned} \tag{2.1}$$

The **BandwidthRatio** operator should be used to specify, for example, the bandwidth shares of two different applications using the network at the same time. Regardless of how many or how few objects are being requested by each application, the total bandwidth will be shared as specified across the two applications. The operator **BandwidthPerElementRatio** is useful when, for example, text and image components are downloaded concurrently. Consider a case when there are 100 small text objects and one large image object. **BandwidthRatio**(s_{text}, s_{img}, 4) will result in each text object getting 0.8% of the available bandwidth while the image gets 20%. On the other hand, **BandwidthPerElementRatio**(s_{text}, s_{img}, 4) would result in each text component getting

roughly 1% and the image getting roughly 0.25%. Since either operator may be preferable in any given situation, both are provided.

Guaranteed-Allocation Constraints

The operator **Allocate**(s, x) reserves x kbit/sec of network bandwidth to transmit set s. If there is less than x kbit/sec bandwidth available, the system reserves all available bandwidth to set s and generates a notification. This type of constraint is useful for transmission of real-time streams and possibly for transmission of a set of objects within a certain period of time.

Never-Transmit Constraints

The operator **Never**(s) specifies that set s is not to be transmitted. This operator has precedence over the other operators, effectively removing s from Ω.

Hierarchical Constraints

If the user is working with several applications using the network concurrently, perhaps a Web browser and a word processor, he might wish to separately allocate bandwidth *across* applications, and then allocate the bandwidth *within* each application across different types of objects. Such a structure mimics the hierarchy already present in documents. For this purpose, I introduce a new operator: $\rightarrow$. Following the above example, imagine the user wishes to give four times the bandwidth to the Web browser over the word processor, and then within each application to give three times the bandwidth to text over images. The

EACS policy would be written:

$$\begin{aligned} rule_1 &= \mathbf{BandwidthRatio}(\\ &\qquad \mathbf{select}(\Omega, \text{type} == \text{“html”}), \\ &\qquad \mathbf{select}(\Omega, \text{type} == \text{“doc”}), 4); \\ rule_2 &= \mathbf{BandwidthPerElementRatio}(\\ &\qquad \mathbf{select}(\Omega, \text{type} == \text{“text”}), \\ &\qquad \mathbf{select}(\Omega, \text{type} == \text{“image”}), 3); \\ rule_1 &\rightarrow rule_2; \end{aligned} \tag{2.2}$$

Thus, rule precedence and partitioning allow constructing hierarchical policies, and these policies need not strictly follow the hierarchy of the application's own component hierarchy. It is equally easy to build, for example, a hierarchy that first splits texts and images, and then splits based on application type, simply by changing the last line of the policy in equation 2.2 to say:

$$rule_2 \rightarrow rule_1 \tag{2.3}$$

2.3.3 Policy Composition

The language described thus far allows for a wide range of policies to be expressed. One of the design goals is to support policy *composition*, when a user might wish to, somehow, mix policies together, resulting in some aggregate policy that combines the effects of the original policies. Such a composition is especially beneficial for ordinary users who do not possess enough expertise to create complicated policies but can easily compose several predefined policies provided by a policy designer, perhaps through a friendly user interface. The system supports two mechanism for policy composition: *concatenation* and *hierarchy*. Both are quite simple. First, I add new language syntax to name policies and give them

separate name spaces:

$$\begin{array}{l} \text{Policy } P_1\{ \\ \quad \textbf{BandwidthRatio}(...); \\ \quad ... \\ \} \\ \text{Policy } P_2\{ \\ \quad \textbf{BandwidthRatio}(...); \\ \quad ... \\ \} \end{array} \tag{2.4}$$

Next, I only need two operators to produce two new policies $P_{concatenation}$ and $P_{hierarchy}$:

$$\begin{array}{lcl} \text{Policy } P_{concatenation} & = & P_1 + P_2; \\ \text{Policy } P_{hierarchy} & = & P_1 \rightarrow P_2; \end{array} \tag{2.5}$$

Composition via concatenation simply computes the union of the constraints from each policy. This might yield an over-constrained system. I will explain how to resolve such policies in Section 3.2.

Composition via hierarchy applies hierarchical constraints, piecewise, across the constraints from P_1 to P_2. The aggregate policy can likewise be evaluated using mechanisms that are described in Section 3.2. As a result of these two simple operators, EACS supports an easy and comprehensible mechanism to compose arbitrary bandwidth policies.

Chapter 3

Policy Resolution

This chapter shows how EACS resolves policies, including cases where a given policy might be over- or under-constraining on the solution space and also discusses CPU efficiency issues with policy resolution.

3.1 Language Semantics

The ultimate goal of the EACS constraint resolver is to assign every object in the system a number, between zero and one, that represents the share of network bandwidth to be allocated to that object. The set of all available objects Ω is taken from the underlying adaptation system. The objects' shares are then used as input to a low-level packet scheduling system (of the adaptation system) that can multiplex the objects together with the requested proportional shares of the total network bandwidth. In the EACS language, where arbitrary subsets of the objects to be transmitted can be chosen and then have their bandwidth constrained by other arbitrary subsets of objects, a robust methodology is needed for resolving the constraints and deriving these bandwidth shares.

Usually a policy does not need to semantically differentiate individual objects rather than simply treating many objects, e.g., with the same attributes (all texts), as a *partition*, a set with the important property that all the objects of the set are assigned an equal bandwidth share. The number of partitions cannot exceed the number of different objects in the Ω and usually is much less than the number of objects in the Ω. Thus, logically, con-

straints should be generated between partitions of the Ω, which can be easily derived from the arguments of constraint operators, significantly reducing the complexity of the solver (Section 3.3). Objects within a partition are transmitted in round-robin fashion allowing further complexity reduction on the level of the network scheduler.

I describe now the precedence of constraint operators and how constraints introduced by each operator are resolved. The solver accumulates all the constraints first, by making a pass over the script program. Then, all constraints introduced by $\mathbf{Allocate}$ operators are resolved in the order determined by the first pass. Each $\mathbf{Allocate}(s, x)$ operator assigns $\frac{x}{BW}$ bandwidth share to partition s, where BW is the current estimation of the available bandwidth, then decrements the amount of the available bandwidth, and removes the objects of s from further consideration. If there is not enough available bandwidth, the rest of the available bandwidth is allocated. Similarly, the $\mathbf{Never}(s)$ operator disables transmission of objects in set s and removes them from further consideration.

At the next stage, the priority constraints must be resolved to determine the set of partition that might be transmitted. If a policy specified **After**(s_1, s_2), then the partitions of s_1 will be guaranteed to have no bandwidth allocated to them unless $s_2 = \emptyset$. This is similar to well-known *make* utility rules in which a target is not compiled until all of its prerequisites are satisfied. Priority constraints can be viewed as specifying a *dependency graph* on the partitions of objects to transmit. The set of partitions allowed ready to be transmitted is equal to the set of partitions with no other partitions depending on them. This can be derived by making a linear pass over all the partitions in the system and checking for adjacent nodes in the dependency graph.

Subsequent to this, the proportional and hierarchical constraints must be resolved. Both of these specify *linear equations* on the bandwidth allocations. **BandwidthRatio**(s_1, s_2, r) adds one rule: the sum of the bandwidth to the partitions of s_1 is equal to r times

the sum of the bandwidth of the partitions of s_2. **BandwidthPerElementRatio**(s_1, s_2, r) generates a similar rule, based on the relationship in equation 2.1. Lastly, some equations must be added that serve as sanity checks. The above constraints should be solved subject to the sum of the bandwidth shares being equal to 1.0. If there exists a unique solution to this problem, it can be found in $O(N^2)$ time, in the number of variables, using a simple substitution technique: set the bandwidth of the first object to 1.0, then start looping over all $N \times (N-1)$ possible pairwise constraints, solving for the other variables, one at a time. This algorithm will also detect if the policy is over- or under-constrained, which requires the use of more expensive techniques, as discussed below. An important observation is that N is the number of partitions, which is typically much smaller than the number of objects.

3.2 Over- and Under-Constrained Policies

Constraints may specify contradictions in priority (e.g., "a before b and b before a") or in proportions (e.g., "$a = \frac{b}{2}$ and $a = 2b$"). It is also possible for a system to be under-constrained, occurring when objects of some type are simply not mentioned in a transmission policy or do not have relationships to all other objects that can be solved (e.g., in the policy "$a = 2b, c = 2d$", there is no relationship between a and c).

In either case, more expensive techniques must be used to solve for the bandwidth shares because there is no longer a single, correct answer.

Priority contradictions If the priorities cannot be resolved, it implies that there must be a cycle in the priority graph. My solution is to merge nodes in the cycle until the cycle no longer exists. When two nodes are merged together, this implies that the two original sets will now be merged together, in terms of their priorities. Any proportional constraints on the original sets would still hold.

Hierarchical contradictions As with priority constraints, there should be a way to define how to evaluate hierarchical constraints over arbitrary graphs. First, I must remove cycles, as I did with priority contradictions. After this, I search for all nodes in the hierarchical constraint graph that have no incoming arrows. These constraints are resolved together and are removed from the graph. Then, the process repeats. After the first group of constraints is resolved, the result is a subdivision of Ω into partitions, each of which has its own bandwidth share. This set of partitions of Ω is the input to the next iteration. The subsequent constraints are then evaluated independently on each partition.

Proportional contradictions These are the most difficult over-constrained problems to solve. My solution to this problem also works well for under-constrained problems. In an over-constrained situation, there can be cycles, much as is the case with priority contradictions, e.g., "a=2b, b=3c, c=4a". However, unlike the priority constraints, which can be represented as a directed, unweighted graph, the proportional constraints would be a directed, weighted graph with certain symmetry: for any pair a and b, if there is a constraint "$a = 2b$", there is the corresponding constraint "$b = \frac{a}{2}$" generated by the solver. Merging nodes together would not necessarily give desirable results.

Instead, the constraints can be considered as goals that must be achieved. For the above example, I now wish to minimize the following function, which intuitively represents the cumulative standard deviation for all the desired goals (fractions):

$$
\begin{aligned}
Error = & \\
& (a-2b)^2 + (b-3c)^2 + (c-4a)^2 + \\
& (b-\frac{a}{2})^2 + (c-\frac{b}{3})^2 + (a-\frac{c}{4})^2
\end{aligned}
$$

Set $HTML = \mathbf{select}(\Omega, \text{application} ==\text{“html”})$;
Set $PPT = \mathbf{select}(\Omega, \text{application} ==\text{“ppt”})$;
// WEB traffic gets 3 times more bandwidth than PowerPoint (PPT) traffic
$rule_1 = \mathbf{BandwidthRatio}(HTML, PPT, 3)$;

Set $TEXT = \mathbf{select}(\Omega, \text{type} ==\text{“text”})$;
Set $IMAGE = \mathbf{select}(\Omega, \text{type} ==\text{“image”})$;
// TEXT components get 4 time more bandwidth than IMAGE components
$rule_2 = \mathbf{BandwidthRatio}(TEXT, IMAGE, 4)$;

Figure 3.1 : HTML vs. PPT, and TEXT vs. IMAGE.

subject to the constraints:

$$\begin{aligned} a+b+c &= 1 \quad \textit{(allocate all available bandwidth)} \\ a,b,c &> 0 \quad \textit{(avoid degenerate solutions)} \end{aligned} \tag{3.1}$$

It is not trivial to deduce the $Error$ function in the case when sets, arguments of the constraint operators, overlap in arbitrary fashion. I scetch the process of generating the $Error$ function for a simple example policy shown in Figure 3.1.

The policy says that WEB traffic must get 3 times more bandwidth than PowerPoint traffic, and, at the same time, TEXT components traffic must get 4 times more bandwidth than IMAGE components traffic. There is no hierarchical constraint, and therefore, both rules must be evaluated simultaneously on overlapping subsets. In the general case, the four subsets, HTML, PPT, TEXT, and IMAGE, form four partitions of Ω:

$$\begin{aligned} A &= HTML \cap TEXT \\ B &= PPT \cap TEXT \\ C &= HTML \cap IMAGE \\ D &= PPT \cap IMAGE \end{aligned} \tag{3.2}$$

These partitions are easily constructed from the arguments of the constraint operators. For

simplicity, in this example I assume that there is only one component in each partition. Each rule is considered independently on the set of all partitions. Thus, $rule_1$ generates the following constraints among the partition shares:

$$\begin{aligned} A+C &= 3\times(B+D) \\ A &= \tfrac{1}{2}\times(A+C) \\ C &= \tfrac{1}{2}\times(A+C) \\ B &= \tfrac{1}{2}\times(B+D) \\ D &= \tfrac{1}{2}\times(B+D) \end{aligned} \tag{3.3}$$

The first equation comes from the constraint operator (HTML=A+B, PPT=C+D). The following four equations define bandwidth fraction of each partition, expressed via the bandwidth fraction of the set to which the partition belongs. For example, the HTML set contains two partitions, A and C, each having one element; these produce equations 2 and 3. Partition membership is known at run-time and such equations can be constructed. Then, the system of equations is transformed to an equivalent one that contains only pairwise relations between partitions:

$$\begin{aligned} A &= 3\times B \\ A &= C \\ A &= 3\times D \\ B &= \ldots \\ &\ldots \end{aligned} \tag{3.4}$$

The other rule, $rule_2$, generates a similar set of constraints. Finally, all pairwise constraints are accumulated in the *dependence matrix* given on Figure 3.2.

Each cell has two numbers, the first one corresponding to the constraint generated by the first rule, and the second one, by the second rule. If a cell has 0, there is no constraints between the partitions. The $Error$ function is then trivially constructed based on

	A	B	C	D
A	0	$3, 1$	$1, 4$	$3, 4$
B	$\frac{1}{3}, 1$	0	$\frac{1}{3}, 4$	$1, 4$
C	$1, \frac{1}{4}$	$3, \frac{1}{4}$	0	$3, 1$
D	$\frac{1}{3}, \frac{1}{4}$	$1, \frac{1}{4}$	$\frac{1}{3}, 1$	0

Figure 3.2 : Dependence matrix represents all pairwise fractions between partition shares.

the coefficients of the matrix.

In the case that, a subset of the possible relationships is completely independent from another such a subset, the system is under-constrained. For example, consider the system "a=2b, c=3d". What should the relationship be between a and c or between b and d? Since there is no correct answer, I synthesize new constraints that, barring anything else, should make them equal. It is not desirable that the synthetic constraints interact poorly with the original constraints, so their effect is scaled down by the factor K, minimizing the following constrained equation:

$$Error = (a - 2b)^2 + (c - 3d)^2 + (b - \frac{a}{2})^2 + (d - \frac{c}{3})^2 + K(a - c)^2 + K(b - d)^2$$

where K is a tunable parameter between zero and one, though in all my experiments I always used the value of 1. Additional experimentation is needed to determine the best value of the factor K.

The solution of the $Error$ quadratic minimization problem is straightforward. Gaussian elimination can derive the global minimum for the $Error$ function (which falls within the constrained region) in $O(N^3)$ time, where N is the number of *partitions* of Ω, which tends

to be small for practical policies. Performance issues are discussed more in Section 3.3.

3.3 Policy Reevaluation and CPU Efficiency

As the system is transmitting objects, various *events* can occur that require the policy to be reevaluated to obtain the set of bandwidth shares consistent with the policy specification at that moment. An object may have finished being transmitted, a new object may have been dynamically added to Ω on the server, or some external event may have occurred (e.g., the user moved a different application to the foreground).

Some transmission policies can be written such that they change continuously as packets are transmitted. For example, consider a policy that selects a set of images that have had at most $\frac{1}{7}$ of their data transmitted:

$$\begin{aligned} \text{Set } s = \textbf{select}(\quad \Omega, \ \ & \text{type} == \text{“image“} \ \&\& \\ & (\text{sizeDone} < \tfrac{1}{7} \times \text{sizeOriginal})); \end{aligned} \tag{3.5}$$

As objects in this set are transmitted across the network, the data transmitted ($sizeDone$) would eventually get large enough that the object should be removed from the set.

Solving this problem would require detecting that a set has a dynamic expression as above and statically solving for the point at which any given object ceases to be a member of the set. However, with arbitrary mathematical expressions, it will not generally be possible to derive such solutions.

A better solution is to periodically reevaluate the transmission policy, perhaps once every few seconds, to discover, at run-time, when set membership or other system parameters change. The reevaluation period depends on network speed to provide good agility and accuracy of the system. I found that if the reevaluation period is equal or less than 2 seconds, the system was responding fast to changes imposed by a policy and the bandwidth shares

were in statistical agreement with the policy specification for all the experiments on a 56 kbit/sec network. Although such periodic reevaluations can potentially waste CPU time to only discover that nothing has changed, the overhead of the policy interpreter is negligible. In all my experiments the overhead of the policy interpreter was less than 1%, even when the policy script was reevaluated 100 times per second to stress the system. Although the complexity of the solver is quadratic and even can be cubic in terms of the number of partitions of Ω, it is *linear* in terms of the script size and the number of objects in Ω. It seems that for all practical policies, the number of partitions would not exceed 10 or 20, keeping CPU overhead at a very low level. On the contrary, the number of objects in Ω can reach several hundreds (it was around 200 for my experimental data set), *not* increasing the overhead significantly.

Chapter 4

Evaluation

In this chapter I first use simulation to demonstrate how the EACS policy resolver works. Then I present performance measurements for sample EACS policies running on the Puppeteer adaptation system.

4.1 Simulation

This section presents the simulation results for two EACS policies that set different strategies for the transmission of a small sets of components from two different documents. Table 4.1 shows, for each component, its name, type, and size, and whether the document to which the component belongs is currently in the background or foreground. These components are fed to the simulator that virtually transmits the objects with the current bandwidth shares until a recomputation of the bandwidth shares is required. This section gives the results for simple policies that do not need periodic reevaluation. All results presented in the following sections assume a bottleneck bandwidth of 56 kbit/sec.

4.1.1 Text First

The *Text First* policy exploits the fact that in many documents, text accounts for a small proportion of a document's content. This policy enables the application to return control to the user faster. The images are then downloaded in the background, while the user browses the text.

Name	Type	Size	application status (bg/fg)
doc1.text1	text	5 kbyte	foreground
doc1.img1	image	60 kbyte	foreground
doc1.img2	image	110 kbyte	foreground
doc2.text1	text	26 kbyte	background
doc2.img1	image	30 kbyte	background
doc2.img2	image	240 kbyte	background

Table 4.1 : Set of components to transfer. The table shows, for each component, the component's name, type, and size, and whether the document is in the background or foreground.

Figure 4.1 shows the EACS code that implements the *Text First* policy. The first statement defines a set s_1, consisting of the text component with the minimum size. The second statement creates a second set s_2, containing all other components. Finally, the last statement ensures that all elements of s_1 are transferred before any elements of s_2.

Figure 4.2 shows the simulator's output for the *Text First* policy. The Y-axis shows the bandwidth allocations for the various components over several time steps. The number below each vertical bar shows the time at which the transmission policy is reconfigured. For the policies discussed in this section, EACS reconfigures the transmission policy, starting a new time step, only after some component finishes transmission.

In the first time step, all of the bandwidth is allocated to the doc1.text1 component, as it is the smallest of the two text components. This allocation lasts for 0.71 seconds — the time that it takes to transfer the 5 kbyte of text over the simulated 56 kbit/sec link. When EACS detects that doc1.text1 has finish transmission, it reconfigures the transmission policy and starts transmitting doc2.text1 — the only remaining text component. After doc2.text1 is transmitted, EACS reconfigures the transmission policy and starts transmitting all remaining images in parallel. In subsequent time steps, as smaller images finish transmission, EACS reconfigures the transmission policy to evenly distribute available bandwidth among the remaining images.

// define the minimum-size text component
Set $s_1 = \mathbf{min}(\mathbf{select}(\Omega, \text{type} == \text{"text"}), \text{sizeOriginal})$;
// define other components
Set $s_2 = \Omega$ SUB s_1;
// transmit the minimum-size text first
$rule_1 = \mathbf{Before}(s_1, s_2)$;

Figure 4.1 : Example *Text First* EACS policy.

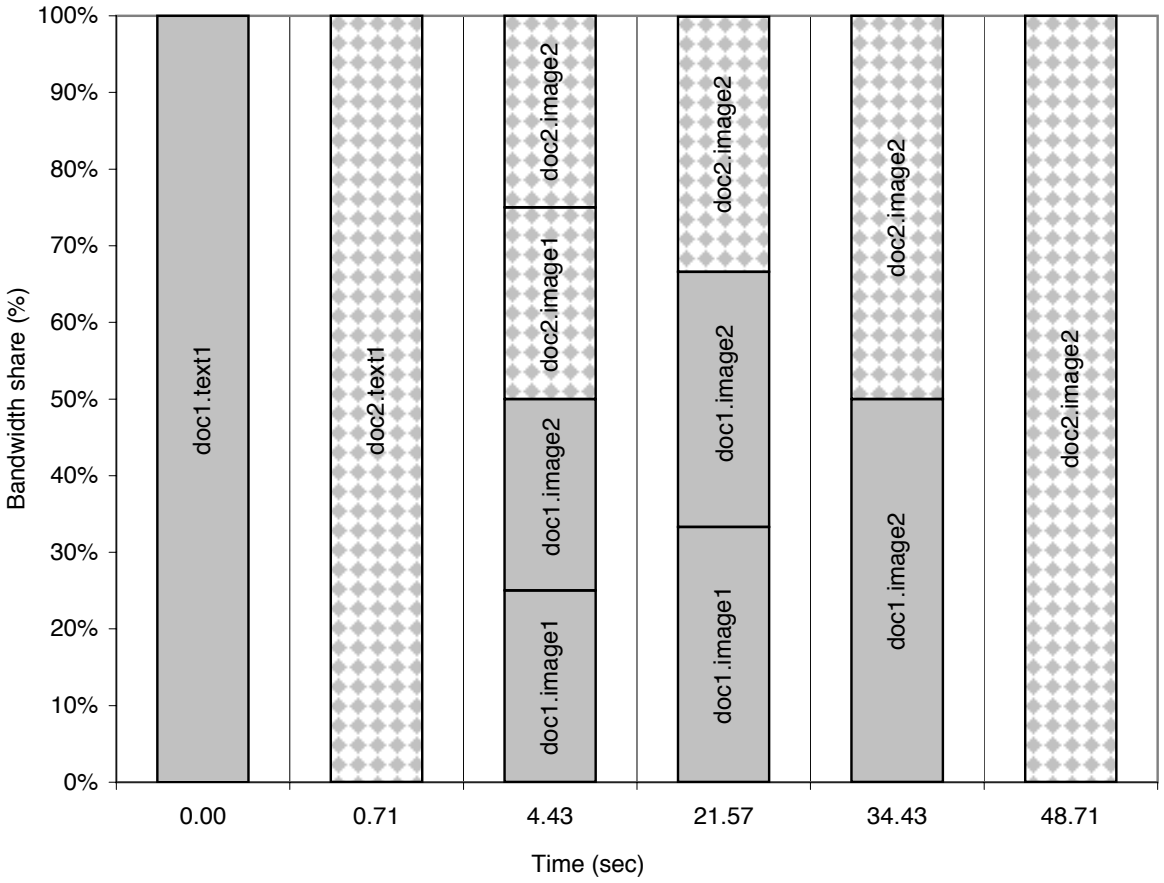

Figure 4.2 : Simulation results for *Text First* policy.

4.1.2 Focus

The *Focus* policy gives twice as much bandwidth to components that belong to the document that happens to be on the foreground than the bandwidth given to components that the belong to background documents. Within the previous bounds, the policy then gives four times more bandwidth to each text component than to each image components.

Figure 4.3 shows the EACS code that implements the *Focus* policy. The first four statements create four sets, dividing the components according to whether they belong to the foreground or background document, and whether they are text or image. The next two statements set the relative bandwidth proportions for the sets. Finally, the last statement specifies that bandwidth should be split based first on the document to which the component belongs, and second on the component's type.

Figure 4.4 shows the simulator's output for the *Focus* policy. At the simulation's beginning, Document1 is in the foreground and its components get $\frac{2}{3}$ of the available bandwidth. This bandwidth is further split between Document1's text and image components, giving Document1's text component $\frac{4}{9}$ of the systemwide available bandwidth ($\frac{2}{3} \times \frac{4}{6} = \frac{4}{9}$) and $\frac{1}{9}$ of the systemwide available bandwidth to each of the two images. After 20 seconds, Document2 moves to the foreground and EACS reconfigures the transmission policy, shifting $\frac{2}{3}$ of the bandwidth to Document2.

4.2 Performance

I measured the performance of EACS with a proof-of-concept implementation on top of the Puppeteer component-based adaptation system [dWZ01]. In the rest of this section, I first describe how the EACS prototype fits in the Puppeteer architecture. Then I present experimental results for several transmission scheduling policies.

Set $s_1 = \mathbf{select}(\Omega, \text{fgr} == \text{true})$;
Set $s_2 = \mathbf{select}(\Omega, \text{fgr} == \text{false})$;
Set $s_3 = \mathbf{select}(\Omega, \text{type} ==$"text"$)$;
Set $s_4 = \mathbf{select}(\Omega, \text{type} ==$"image"$)$;
$rule_1 = \mathbf{BandwidthRatio}(s_1, s_2, 2)$;
$rule_2 = \mathbf{BandwidthPerElementRatio}(s_3, s_4, 5)$;
$rule_1 \rightarrow rule_2$;

Figure 4.3 : Example *Focus* EACS policy.

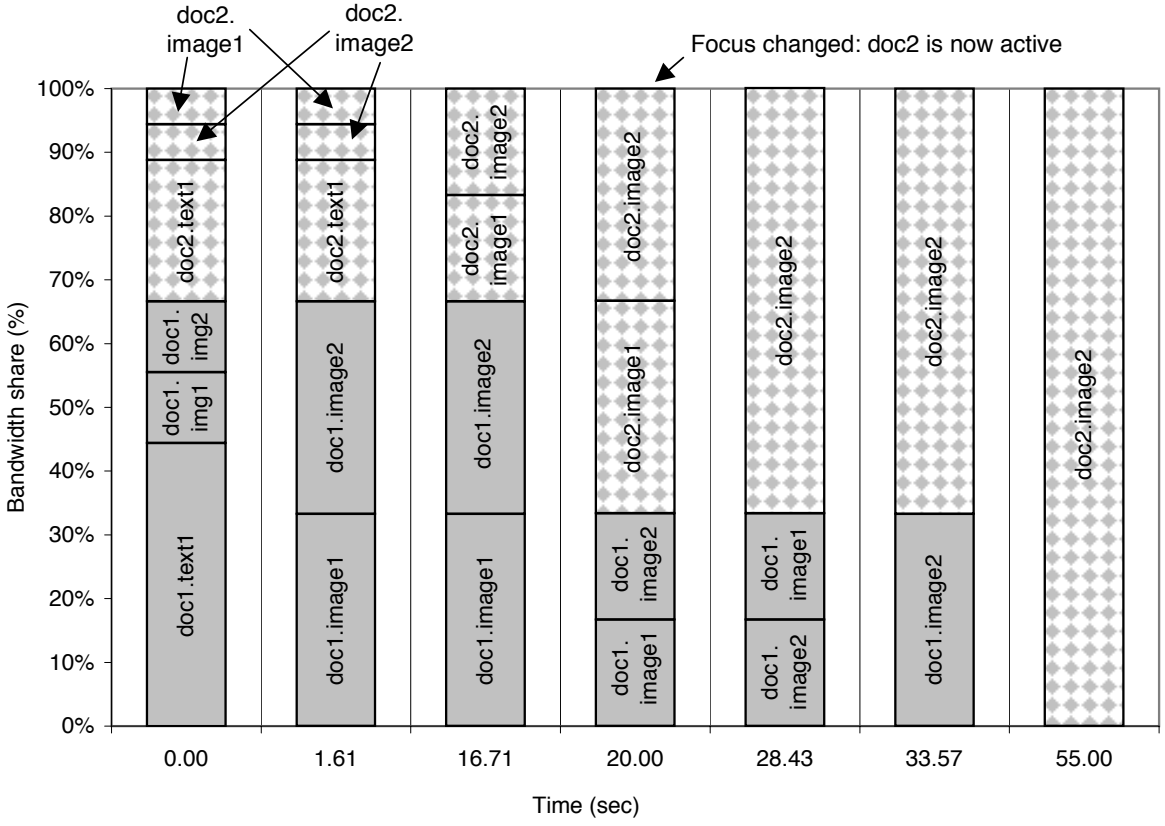

Figure 4.4 : Simulation results for *Focus* policy.

4.2.1 Puppeteer

Puppeteer adapts component-base applications running on bandwidth-limited devices by calling on the run-time interfaces these application expose. Puppeteer reduces the time it takes to load documents in component-based applications, such as those in the Microsoft Office or Sun OpenOffice suits, by providing to the applications transformed versions of documents that consists of a subset of the components of the original documents (e.g., just a few pages or slides). After the document is rendered and the application returns control to the user, Puppeteer uses the application's exposed API to extend the document with additional components or to upgrade the fidelity of components transmitted with low fidelity.

Figure 4.5 shows Puppeteer's system architecture. All data flowing in and out of the bandwidth-limited device goes through the Puppeteer local and remote proxies. The local proxy runs on the bandwidth-limited device and adapts the application by calling on the application's run-time API. The remote proxy runs at the other end of the bandwidth-limited link and has fast access (relative to the bandwidth-limited client) to the servers storing the documents being adapted.

The initial EACS implementation runs on the Puppeteer remote proxy (running it on the Puppeteer local proxy would cause extra bandwidth overhead due to transmission of scheduling information to the remote proxy) and is limited to scheduling data flowing into the bandwidth-limited client. The EACS prototype relies on a scheduler that implements the WF^{2}Q+ Packet Fair Queuing (PFQ) algorithm [BZ96], to distribute bandwidth among the partitions according to their rate allocations; round-robin schedulers further split bandwidth among components within each partition.

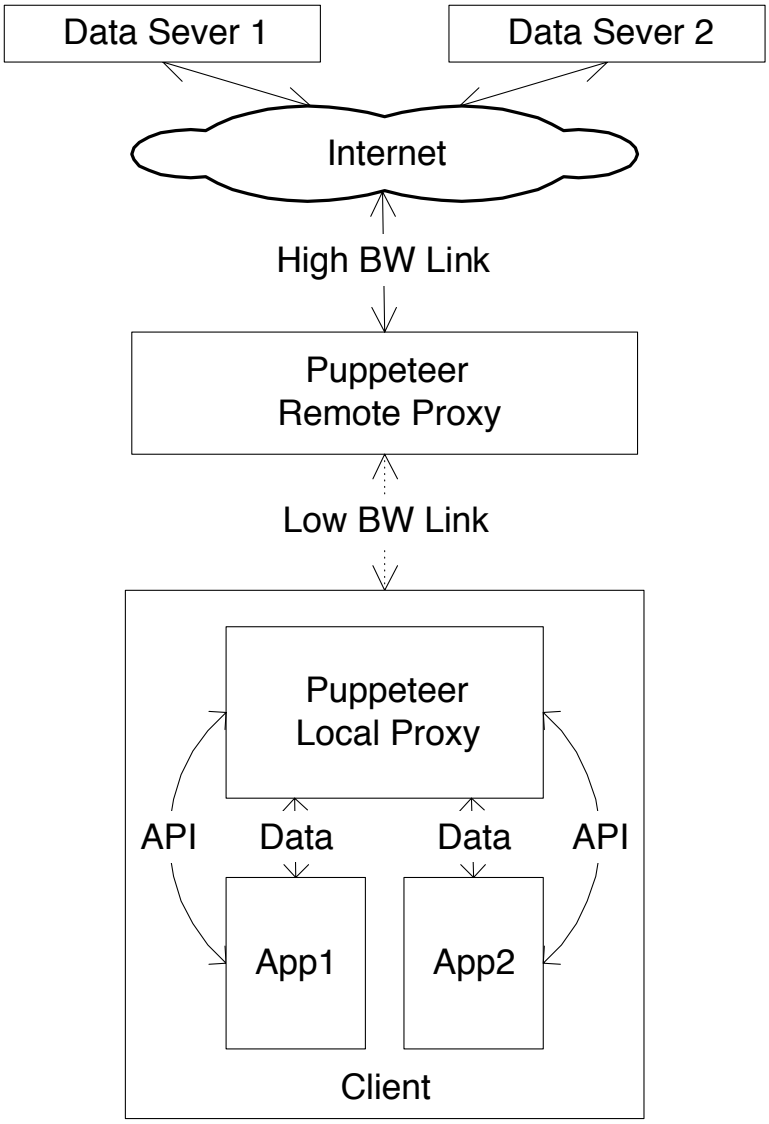

Figure 4.5 : Puppeteer architecture.

4.2.2 Text First + Policy

The only difference of *Text First +* policy from the *Text First* policy presented in Section 4.1.1 is $rule_2$ that enforces equal bandwidth distribution between IE5 and PPT components of the same type. I quantify the performance of the *Text First +* EACS transmission policy presented on Figure 4.6 by simultaneously loading an image-rich 668 kbyte Web page and a 1.05 Mbyte PowerPoint presentation using Internet Explorer 5.5 (IE5) and Microsoft PowerPoint (PPT).

For this experiment, the Web page is requested first, and a few seconds later the user starts downloading the PowerPoint presentation, while some of the images of the Web page are still being transferred. The adaptation policy loads a document into the application as soon as all of document's 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.

The objective is to minimize the time that it takes to get all PowerPoint text components to the client. A best effort scheduler would just split the available bandwidth equally between the Web page and PowerPoint components. Instead, *Text First +* should prioritize the transmission of PowerPoint slide text, potentially cutting in half the time to open a text-only version of the presentation.

I ran the experiments on a platform consisting of two Pentium III 500 MHz machines and one Athlon 1.2 GHz machine, all running Windows 2000. One of Pentium III 500 MHz machines was configures as a data server running Apache 1.3, which stores the two documents used in the experiments, and the other was configured as a client that runs the user's applications and the Puppeteer local proxy. The Puppeteer remote proxy and EACS resolver were run on the Athlon 1.2 GHz machine. The local and remote Puppeteer proxies communicate via another PC running the DummyNet network simulator [Riz97]. This

Set s_1 = **select**(Ω, type==“text”);
Set s_2 = Ω SUB s_1;
$rule_1$ = **Before**(s_1, s_2);
Set s_3 = **select**(Ω, application==“html”);
Set s_4 = **select**(Ω, application==“ppt”);
$rule_2$ = **BandwidthRatio**(s_3, s_4, 1);

Figure 4.6 : Example *Text First* + EACS policy.

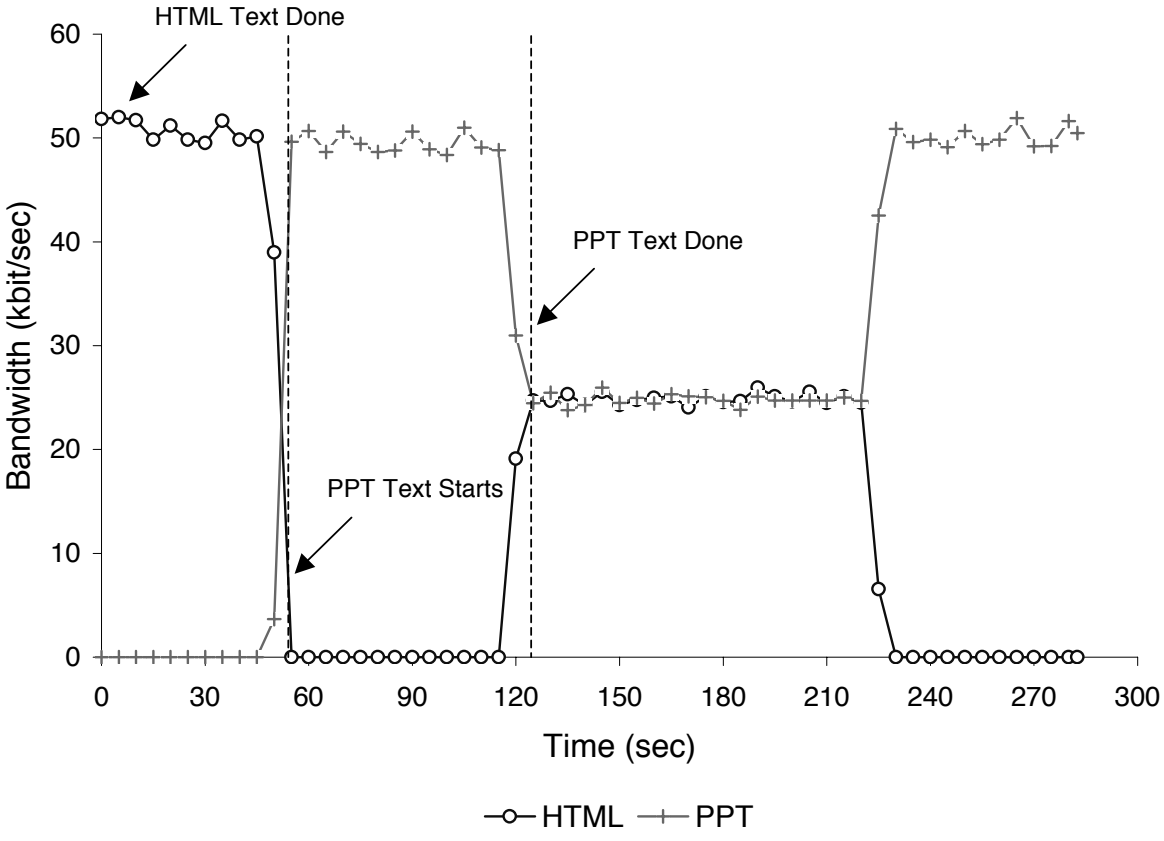

Figure 4.7 : Bandwidth allocations for loading a Web page and a PowerPoint presentation with the EACS *Text First* + policy.

setup allows emulating 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.

Figure 4.7 shows the bandwidth allocations for loading the two documents over a 56 kbit/sec network link. *Text First* + reallocates bandwidth from sending images embedded in the Web page to sending PPT slides. This reallocation lowers the time to load a text-only version of the PPT presentation by 49% compared to a best-effort scheduler. Moreover, the EACS policy interpreter does not introduce extra bandwidth overhead to the current implementation of Puppeteer; I measured 90% bandwidth utilization, the same as Puppeteer achieves without EACS.

4.2.3 Focus Policy

Figure 4.8 shows the EACS code that implements the *Focus* policy as well as network bandwidth distribution in time. The *Focus* EACS transmission policy is similar to the *Focus* presented in Section 4.1.2. The difference is $rule_2$ that enforces equal bandwidth distribution between IE5 and PPT components of the same type. The same documents as mentioned above were used for the experiment.

In this experiment, the user first requests the Web page and in 40 seconds opens the PPT presentation. Before 40 seconds, the Web page is being downloaded alone and allocated all available bandwidth. When the user opens the PPT document, it becomes the *foreground* application and $rule_1$ forces Puppeteer to shift the bandwidth to PPT presentation traffic, which is now allocated 80%. The time of the bandwidth reallocation on the figure is shifted 8 seconds because the request to open the PPT presentation must get to the server first and, then, the document must be parsed before the server can start actual transmission of the data. $rule_2$ allows taking advantage of using the *Text First* policy for each of the

Set $s_1 = \mathbf{select}(\Omega, \text{fgr} == \text{true})$;
Set $s_2 = \Omega$ SUB s_1;
$rule_1 = \mathbf{BandwidthRatio}(s_1, s_2, 4)$;
Set $s_3 = \mathbf{select}(\Omega, \text{application} ==\text{“text”})$;
Set $s_4 = \mathbf{select}(\Omega, \text{application} ==\text{“image”})$;
$rule_2 = \mathbf{Before}(s_3, s_4)$;
$rule_1 \rightarrow rule_2$;

Figure 4.8 : Example *Focus* EACS policy.

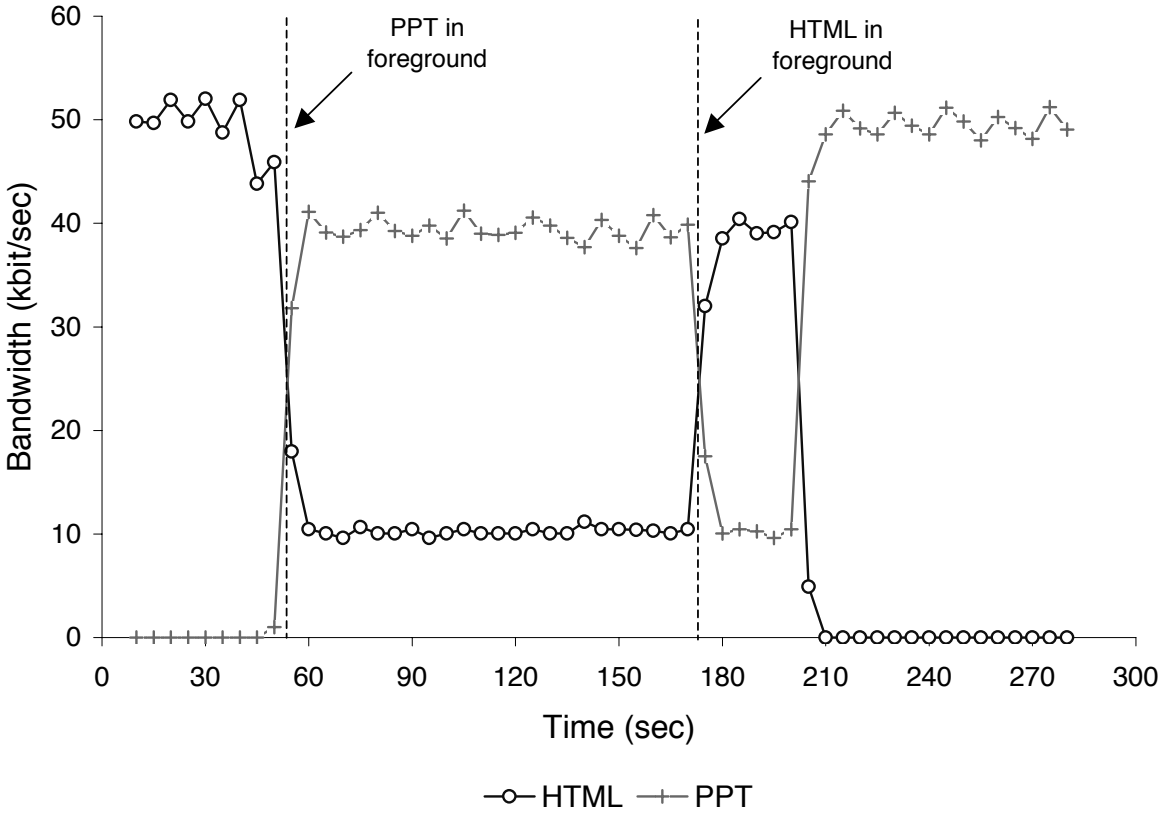

Figure 4.9 : Bandwidth allocations for loading a Web page and a PowerPoint presentation with the EACS *Focus* policy.

applications. Another bandwidth shift occurs at 170 seconds, when the user selects the Web browser to be in the foreground.

4.2.4 Guaranteed-Bandwidth Allocation Policy for Audio Stream

The EACS *Allocate(s, x)* operator reserves x kbit/sec of available bandwidth to transmit set s. It can be used for transmission of real-time streams. Figure 4.10 shows the EACS policy code that implements the *Audio* policy. In this setup, the user is listening to a 32 kbit/sec audio stream while browsing the Web. The total available bandwidth is 56 kbit/sec. In this scenario, a best-effort scheduler would divide the bandwidth between the two streams roughly equally (28 kbit/sec), which causes dropped packets in the audio stream, resulting in poor audio perception. The *Allocate(AUDIO, 32)* operator reserves 32 kbit/sec for the audio stream, which avoids this problem. Knowing the available bandwidth BW, EACS reserves $\frac{x}{BW}$ of the bandwidth for the audio by adjusting the coefficients of Puppeteer's proportional scheduler accordingly. The rest of the bandwidth is used by the HTML traffic. The resulting throughput of the audio stream on Figure 4.11 is slightly higher than 32 kbit/sec. Following the approach of the original HATS [dWZ02] paper, all throughput results are presented counting control header, which contributes 12 "extra" bytes, the size of the control header, per 256-byte packet. This fact explains the difference for the audio stream throughput.

The *Allocate* operator can be used for reserving the bandwidth not only for real-time streams. For example, it might be desirable to transmit a subset of data within a certain amount of time. Then the *Allocate* operator can be used to reserve a fraction of the bandwidth to avoid interference of other traffic. Moreover, the order of transmission of the set, e.g., text-first, can be defined by other types of constraints for the set.

Set $AUDIO = \mathbf{select}(\Omega, \text{type} == \text{"audio"})$;
Set $s_2 = \Omega$ SUB $AUDIO$;
$rule_1 = \mathbf{Allocate}(AUDIO, 32)$;
// Here might follow other constraints for set s_2.

Figure 4.10 : Example *Audio* EACS policy.

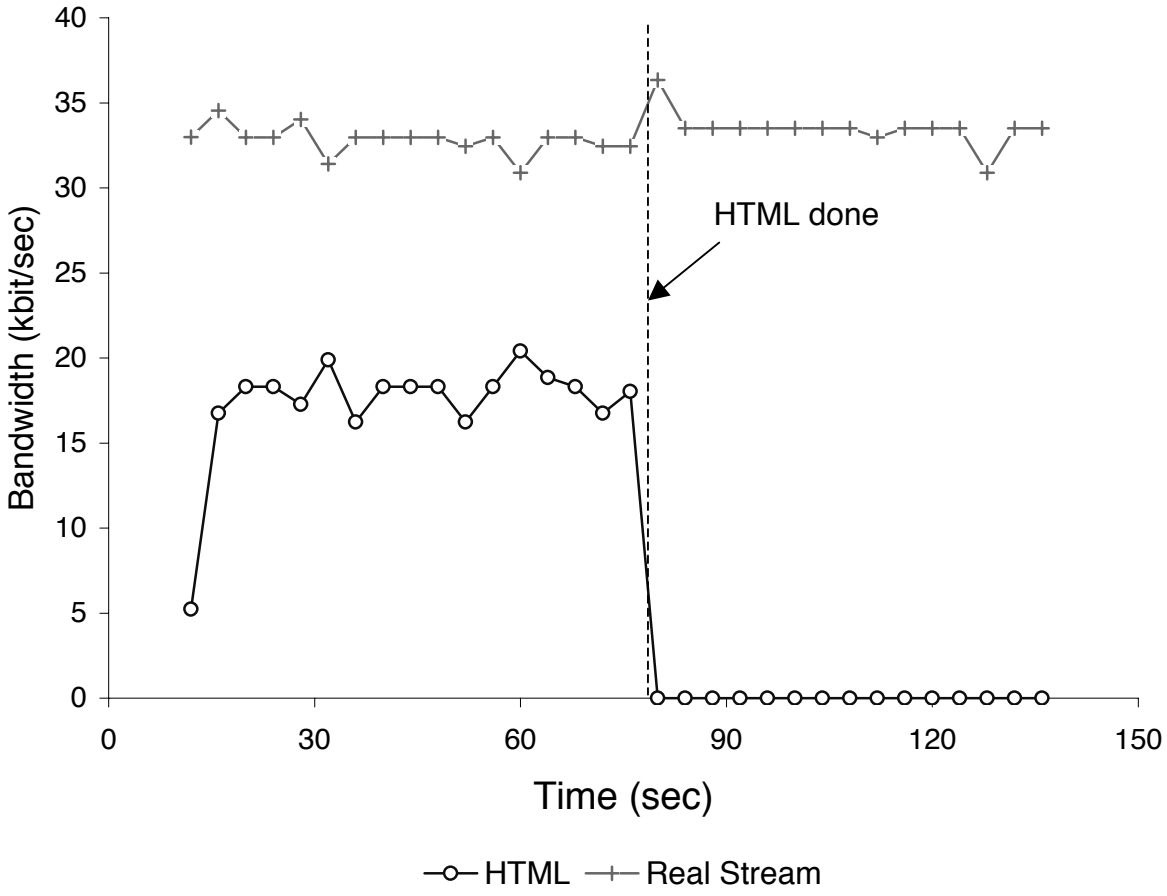

Figure 4.11 : Bandwidth allocations for 32 kbit/sec audio stream and loading a Web page with the EACS *Audio* policy.

4.2.5 Web Pages Preview Policy

This section shows how time-based constraints can be added to the system. Figure 4.12 presents *Web Page Preview* policy that limits transmission of HTML objects to a certain maximum duration each. To motivate the policy, imagine a mobile user who pays for time spent for data transmission. If the user wants to limit the amount of money spent for transmission of a Web page he is currently browsing, it is necessary to limit the duration of transmission of the page. In other words, the high-level description of the policy is: "spend at most T seconds per HTML objects I download". Each object in the system has an attribute *timeTransmissionStarted* that records the time (ms) when the object was added for transmission. It is also assumed that "new" HTML traffic can always be transmitted. The first two statements of the policy script determine what HTML objects in the system are "new", meaning they have been transmitted for less than *timeout* milliseconds. The system variable *currentSystemTime* contains the current time (ms) when this round of script evaluation started. Then all the "old" HTML objects are removed from the transmission. Transmission of the NEW_HTML set and the NOT_NEW_HTML set can be customized by arbitrary additional constraints. In this example policy, 50% of the available bandwidth is allocated to the NEW_HTML set and text is transmitted first and then the smallest image.

Figure 4.13 shows bandwidth distribution in time for a possible scenario in which the user is downloading a Web page and starts downloading a PowerPoint presentation after 20 seconds. The figure shows how bandwidth is equally redistributed between PPT and the Web browser when both applications download data concurrently, and how the system stops HTML traffic in 70 seconds, redistributing all the bandwidth to the PPT presentation.

```
// global variable-helpers
double timeout = 70 * 1000;

void script () {
    // select all HTML objects
    Set HTML = select(Ω, application == "html");
    // select only "new" HTML objects
    Set NEW_HTML = select(HTML,
        (currentSystemTime − timeTransmissionStarted) ≤ timeout);

    // calculate OLD_HTML components
    Set OLD_HTML = HTML SUB NEW_HTML;
    // disable them
    rule100 := Never(OLD_HTML);

    // other constraints
    Set TEXT = select(Ω, type == "text");
    Set NOT_TEXT = Ω SUB TEXT;
    Set NOT_NEW_HTML = Ω SUB NEW_HTML;

    // give 50% of bandwidth to NEW_HTML traffic
    rule1 := BandwidthRatio(NEW_HTML, NOT_NEW_HTML, 1);
    // transmit TEXT first
    rule2 := Before(TEXT, NOT_TEXT);
    rule1 → rule2;

    // Transmit min-size images for NEW_HTML
    Set NEW_HTML_IMAGE = select(NEW_HTML, ! type == "image");
    Set NEW_HTML_IMAGE_MIN =
        min(NEW_HTML_IMAGE, sizeOriginal);
    rule3 := Before(NEW_HTML_IMAGE_MIN,
        NEW_HTML_IMAGE SUB NEW_HTML_IMAGE_MIN);
    rule1 → rule3;
}
```

Figure 4.12 : Example *Web Page Preview* policy.

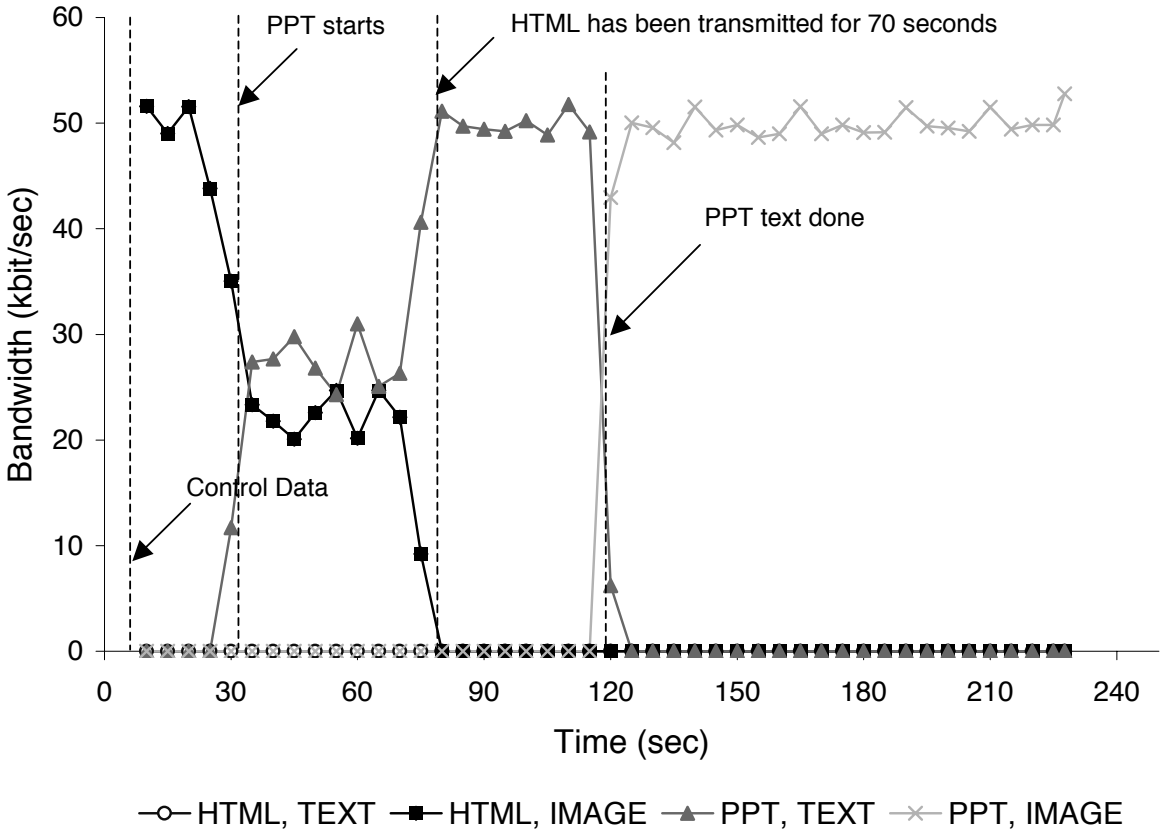

Figure 4.13 : Bandwidth allocations for 70 seconds per HTML page with the EACS *Web Page Preview* policy.

4.2.6 Web Pages Preview Policy Variations

As it was mentioned in the previous subsection, the transmission of a set of data constrained by time (NEW_HTML) can be constrained further. Two more examples of such constraining are "transmit all the images in round-robin fashion" and "fetch all the images at equal fidelity level" provided all text components have been transmitted.

The first one of these constraints is straightforward to express. In fact, the default downloading strategy for an equivalence class (NEW_HTML) is exactly to fetch the objects of the class in round-robin fashion. Thus, no additional constraining on NEW_HTML images is necessary. In this policy, the user gets small images at high-fidelity level and potentially gets larger images with poorer fidelity if the bandwidth is restricted and the entire set cannot be download within the specified period of time.

The second policy code is given on Figure 4.14. I assume here that all the images are in a progressive format, e.g., Progressive JPEG of JPEG2000. Thus, the fidelity level is defined as the ratio of downloaded bytes to the total size of the image. The script determines HTML low-fidelity images to send and if this set turns out to be empty, increases the fidelity level by 10%. The initial fidelity level is chosen to be 30%. In this policy, the user can see all the images at roughly equal fidelity level within 10% inaccuracy (possibly except some images that reached 100% fidelity).

I measured the CPU utilization of the entire Puppeteer system for this policy for three different network bandwidth values of 56, 128, and 384 kbit/sec. The measurements were done for policy reevaluation intervals of 1000 ms (reevaluate the script once per second), 100 ms, and 10 ms (100 times per second). The CPU of the server machine, where the policy script was evaluated, was always utilized less than 1%, excluding the periods while Puppeteer was parsing documents. This suggests that the concept of periodic policy reevaluation would work well for slow networks in most of the cases. For all reevaluation in-

```
double fidLevel = 0.3;

void script () {
        // define HTML_NEW_IMAGE set (similar as it was done above)
        ...
        if (cardinality(HTML_NEW_IMAGE) > 0) {
           Set lowFidImgs = select(HTML_NEW_IMAGE,
                 (sizeDone / sizeOriginal) < fidLevel);

           // all the images reached this fidelity level,
           // increment the fidelity level
           if ( pow(lowFidImgs) == 0 )  fidLevel = fidLevel + 0.1;
           rule1 := Before(lowFidImgs,
                 HTML_NEW_IMAGE SUB lowFidImgs);
        }
}
```

Figure 4.14 : The EACS *Web Page Preview* policy code.

tervals, bandwidth utilization reaches approximately the same level corresponding to the network speed and does not change for the tested reevaluation periods.

4.2.7 Web Page Image Preview Policy

Another useful policy is to make Puppeteer bring a new image from an HTML page at least every T seconds. However, this formulation is too broad. It should be restricted to specify how much resources (bandwidth) the user agrees to use. For example, he can allocate at most 50% of the available bandwidth. If there is not enough bandwidth to fetch the entire image, the image is displayed at the corresponding fidelity level.

The policy source code is presented on Figure 4.15. Though this policy is similar to the *Web Pages Preview* policy, it demonstrates the usage of the *update* function to remember the state associated with an object, updating the field *timeActualTransmissionStarted* when the system starts transmitting the image.

```
// field declaration to remember when the system starts transmitting the object
system field double timeActualTransmissionStarted;      // initialized to 0

// global variable-helpers
double timeout = 30 * 1000;

void script() {
    // select all HTML objects
    Set HTML = select(Ω, application == "html");
    // select all HTML images
    Set HTML_IMAGES = select(HTML, type == "image");

    // Transmit HTML text first (just an agreement)
    rule1 := Before(HTML SUB HTML_IMAGES, HTML_IMAGES);

    // select HTML image that is being transmitted
    Set ACTIVE_HTML_IMAGES = select(HTML_IMAGES,
        (currentSystemTime−timeActualTransmissionStarted) < timeout);

    if (pow(ACTIVE_HTML_IMAGES) == 0) {
        // No active image: make one active
        ACTIVE_HTML_IMAGES = get1(select(HTML_IMAGES,
            timeActualTransmissionStarted == 0));
        update(ACTIVE_HTML_IMAGES, "timeActualTransmissionStarted",
            currentSystemTime);
    }

    // the user agrees to allocate 50% of bandwidth for this preview
    rule2 := BandwidthRatio(ACTIVE_HTML_IMAGES,
        Ω SUB ACTIVE_HTML_IMAGES, 1);
    // or it can be: try to reach certain fidelity
    // or it can be: allocate fixed amount of the bandwidth, ...

    // other constraints can be imposed
    ...
}
```

Figure 4.15 : The EACS *Web Page Image Preview* policy code.

Policy	Lines of code
text-first (or focus) (Figures 4.1 and 4.3)	5
text-first+ (Figure 4.6)	9
allocate 32 kbit/sec to audio stream (Figure 4.10)	5
low fidelity images first (Figure 4.14)	8
max 30 seconds per web page (Figure 4.12)	18

Table 4.2 : Script sizes for several EACS policies.

4.3 Policy Expression Efficiency

Figure 4.2 shows the sizes of sample policy scripts that I implemented and tested on top of Puppeteer. Implementation of equivalent policies in Java requires several hundred lines of code each. In contrast, the EACS policy scripts are much shorter and do not deal with the details of the adaptation system.

Chapter 5

Related Work

HATS [dWZ02] experimented with combining dynamic control over bandwidth scheduling and adaptation. While this combination enables to adapt multiple applications in concert (HATS intended purpose), it required coding transmission scheduling policies in Java. As a result, a significant programming effort was needed to implemented every new transmission strategy. Moreover, the HATS system was limited to hierarchical transmission strategies that are closely linked with the hierarchical structure of the applications, documents, and components running on the bandwidth-limited device. In contrast, EACS supports the implementation of hierarchical transmission strategies based on other criteria. For example, it is straightforward to write an EACS policy that splits bandwidth first based on component type and then based on the application or documents that owns the component.

EACS provides a language to specify a domain-specific scheduling policy. Network scheduling is, by itself, a robust field of research including work that enables clients to specify their quality of service (QoS) network requirements [FRV92, ZDE93], provides differentiated service in network hierarchies [BZ96, FJ95], or adds differentiated services to general purpose operating systems [And93, Alm99, CN99, SCG+00]. EACS is fundamentally built on the concept of solving constraints, which is also an area that has been extensively studied [Tsa93]. EACS differs from other constraint-based network scheduling strategies in the ability to provide high-level specification (via data sets and constraints) of how to transmit any data, and then configures lower-layer packet scheduler(s) to do the real transmission. Most other works target the low-layer transmission or real-time streams.

Chapter 6

Conclusion and Future Work

In this work, I have demonstrated a general-purpose system for specifying bandwidth usage policies, where the user can specify constraints that apply to different sets of objects based on their attributes. Constraints can specify that some objects go before other objects, or they can specify that some objects must get a specific proportion of the available bandwidth. Even if these policies are under- or over-constraining, the system can still efficiently solve for optimal proportions of the total bandwidth to be applied to each individual object. As a result of this freedom, users can write and compose bandwidth policies without being forced to worry about any of the low-level details of bandwidth policy implementations.

The Extensible Adaptation via Constraint Solving (EACS) system provides the user with a simulator, to simplify and accelerate the design and testing of bandwidth policies. Furthermore, executing bandwidth policies with real data shows very little system overhead.

In the future, there is a number of additional features that would be beneficial to study with EACS. It is interesting to study how much information (attributes) should be exposed by an adaptation system to cover a large domain of transmission policies. I also would like to investigate how to add the notion of object subsets and transformations (removing or transcoding objects) into the EACS policy language as well as how and when to update the user's application data. Using the composition mechanisms in EACS, I would like to simplify policy design even further to aid unsophisticated users in selecting appropriate adaptation policies from predefined policy blocks. The user interface would generate code

for the underlying EACS system to evaluate, bringing the power and flexibility of EACS to the widest possible spectrum of users.

Bibliography

[Alm99] W. Almesberger. Linux network traffic control - implementation overview, April 1999. `http://lrcwww.epfl.ch/linux-diffserv/`.

[And93] David P. Anderson. Metascheduling for continuous media. *ACM Transactions on Computer Systems*, 11(3):226–252, August 1993.

[BCKP95] 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.

[BZ96] J. Bennett and H. Zhang. Hierarchical packet fair queuing algorithms. In *Proceedings of SIGCOMM'96*, Stanford, California, August 1996.

[CN99] 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.

[CSWK01] Hao-hua Chu, Henry Song, Candy Wong, and Shoji Kurakake. Seamless applications over roam system. In *Proceedings of the Workshop on Application Models and Programming Tools for Ubiquitos Computing (UbiTools'01)*, Atlanta, GA, September 2001.

[dWZ01] 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*, San Francisco, California, March 2001.

[dWZ02] Eyal de Lara, Dan S. Wallach, and Willy Zwaenepoel. Hats: Hierarchical adaptive transmission scheduling. In *Multimedia Computing and Networking*, San Jose, California, January 2002.

[FGBA96] A. Fox, S. D. Gribble, E. A. Brewer, and E. Amir. Adapting to network and client variability via on-demand dynamic distillation. *SIGPLAN Notices*, 31(9):160–170, September 1996.

[FJ95] 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.

[FRV92] 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, December 1992.

[FS99] Jason Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, Kiawah Island Resort, SC, December 1999.

[Kat94] Randy H. Katz. Adaptation and mobility in wireless information systems. *IEEE Personal Communications*, 1(1):6–17, 1994.

[KK00] 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, April 2000.

[NSN+97] 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(5):276–287, December 1997.

[Riz97] L. Rizzo. DummyNet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1):13–41, January 1997.

[Sat96] M. Satyanarayanan. Fundamental challenges in mobile computing. In *Fifteenth ACM Symposium on Principles of Distributed Computing*, Philadelphia, Pennsylvania, May 1996.

[Sat01] M. Satyanarayanan. Pervasive computing: Vision and challenges. *IEEE Personal Communications*, 2001.

[SCG+00] 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, November 2000.

[TMT01] Kazunori Takashio, Masakazu Mori, and Hideyuki Tokuda. m-p@gent: A framework for describing environment-aware mobile agents on ubiquitous computing environment. In *Proceedings of the Workshop on Application Models and Programming Tools for Ubiquitos Computing (UbiTools'01)*, Atlanta, GA, September 2001.

[Tsa93] E. P. K. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, London and San Diego, 1993. ISBN 0-12-701610-4.

[Wei93] Mark Weiser. Some computer science problems in ubiquitous computing. *Communications of the ACM*, July 1993.

[ZDE93] L. Zhang, S. Deering, and D. Estrin. RSVP: A new resource ReSerVation Protocol. *IEEE Network*, 7(5):8–18, September 1993.