

Extensible Adaptation via Constraint Solving

Yuri Dotsenko[†], Eyal de Lara[‡], Dan S. Wallach[†], and Willy Zwaenepoel[†]

[†] Department of Computer Science

[‡] Department of Electrical and Computer Engineering
Rice University

Abstract

This paper 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 our language is the ability to express constraints among the objects to be transmitted. A policy can make ordering constraints such as “all text objects are transmitted before any image objects” or a policy might express rules on the relative bandwidth allocations across objects of different types or it can reserve certain amount of bandwidth for an object. Because it is possible to express contradictory constraints, our system finds suitable approximate solutions when no precise solution is available.

1 Introduction

The ability to adapt to limited and varying resources has long been considered a fundamental requirement for mobile computing [1, 2, 3]. The advent of pervasive computing [4, 5] 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 the client happens to be in.

In the last 10 years, researchers have devised several mechanisms for adapting to variation in network connectivity [6, 7, 8, 9], energy supply [10], and device heterogeneity [11, 12]. 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, the presence or absence of other users in the smart space, etc. Moreover, the space of possible adaptations is combinatorial in the number of services provided by the smart space, the number of applications concurrently run-

ning, 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, we believe 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 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, 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 paper presents Extensible Adaptation via Constraint Solving (EACS), a novel approach that makes a big step in simplifying 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 the objects and defining constraints among these subsets; for less advanced users, there is also an option of composing 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 user-defined 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 band-

width as elements of set D) as well as they can allocate (reserve) fixed amount of bandwidth to a set or disable the transmission of the set permanently or temporarily.

We first developed an EACS simulator, allowing us to simulate real EACS policies without actually transmitting data across a network. This allows a policy designer to quickly see how a policy might behave on real data without waiting for the data to traverse a low-speed network. EACS proved effective at expressing complex transmission policies with very few lines of code, a significant improvement over our earlier HATS system [13], where policies were written in Java and were generally constrained to follow the hierarchical structure built into existing documents. We integrated EACS with the Puppeteer adaptation system [6] and we then measured the overhead introduced by EACS policy interpreter, showing it to be negligible.

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

2 Policy Language Design

This section presents the design criteria behind EACS. We present a method for expressing transmission policies in a high-level, compact fashion.

2.1 Transmission policy domain overview

We open the discussion by presenting a descriptive overview of possible transmission policies based on our long-term experience with the Puppeteer system [6].

- **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 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 policies is useful when the subset of data that the user would expect in the nearest 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 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 adequate level of audio perception.
- **Time-constrained policies:** a policy of the kind can be useful to save money the user pays for the transmission time. For example, if the user is browsing the web, the system can be told to spend not more than 30 seconds per 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 subsections explain how to abstract these (and more) policies and formalizes our language to specify them.

2.2 Language Requirements

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. As such, the purpose of our language is to specify how a series of objects, stored on the server and labeled with various attributes, should be transmitted across the network.

Our 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, we introduce two basic concepts: set operations and constraints on these sets. A transmission policy, then, 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. The system to evaluate these policies must be cheap enough to evaluate that it can be reevaluated often, allowing for dynamic changes in the set of objects to be transmitted.

In order to express transmission policies, we need to be able to select groups of the available objects based on their attributes. Constraints might be expressed on an object's *size*, its *type* (text, image, sound, ...), the time the object was added for transmission, its *context*, etc. An object's context will have some elements that are static, such as what kind of application is reading the object, and other elements that are dynamic, such as whether the application requesting the object happens to be the user's foreground application.

By selecting on the attributes, our language must also define *sets* using standard set operations: union, intersection, and difference.

To express transmission policies, constraints can take many forms.

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

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%).

Guaranteed-allocation constraints reserve requested amount of bandwidth to a set of objects (e.g., an audio stream)

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

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's a much simpler language to evaluate. The operators supported include: variables and constants; if-then-else operators; 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).

In addition, a number of other useful primitives are defined:

- Set *result* = **select**(*set*, *expr*);
The function selects all the components of *set* for which *expr* evaluates to *true*.
- void **update**(*set*, *attribute*, *expr*);
The function updates the attribute *attribute* of all the objects of *set* to the value of *expr*.
- Set *result* = **min**(*set*, *expr*);
The function select the component of *set* for which *expr* has minimum value. Likewise, there is a **max** function.
- Set *result* = **get1**(*set*);
The function selects one element from *set*.

All the usual set operations are defined, including intersection (**AND**), union (**OR**), and difference (**SUB**). In addition, we define Ω as the universe of all objects and we write \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:

type describes what kind of object this is

application describes the application requesting the object

sizeDone bytes of the object that have been transmitted

sizeOriginal total bytes for the object

fg true if the object belongs to the foreground application, false otherwise

timeAddedToTransmission the absolute time when the object was added for transmission

timeLastTransmitted the last absolute time when a portion of the object was transmitted

Clearly, the list is not exhaustive and can be extended.

2.3.1 Priority Constraints

Next, we have the operators that express various constraints. Given two sets of components s_1 and s_2 , we might 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).

2.3.2 Proportional Constraints

We 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);
means that the ratio between the total bandwidth the components in set s_1 and the total bandwidth of the 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);
means that 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:

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

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

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

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

The first equation expresses that the bandwidth allocated to a and b together (the members of s_1) must be twice what is given to c (the sole member of s_2). The second equation expresses that a and b must have the same bandwidth, as they're 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 $a = b = c = 1/3$ 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 pairs 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, $a = b = 2/5$ and $c = 1/5$, shows the desired bandwidth distribution.

It can be noticed that **BandwidthRatio** and **BandwidthPerElementRatio** are related. For the same non-empty sets s_1 and s_2 :

$$\begin{aligned}\mathbf{BandwidthPerElementRatio}(s_1, s_2, r) &\equiv \\ \mathbf{BandwidthRatio}(s_1, s_2, r \frac{|s_1|}{|s_2|}) &\end{aligned} \quad (1)$$

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 fairly 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 component getting 0.8% of the available bandwidth while the image gets 20%. Whereas, **BandwidthPerElementRatio**($s_{text}, s_{img}, 4$) would result in each text component getting roughly 1% and the image getting roughly .25%. Since either operator may be preferable in any given situation, we provide both.

2.3.3 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 constraints is useful for transmission of real-time streams and possibly for transmission of a set of objects within a certain period of time.

2.3.4 Never-Transmit Constraints

The operator **Never**(s) specifies that set s is not transmitted. This operator has precedence over the other operators, effectively removing s from Ω until the policy is reevaluated. See section 3.3 for more information on reevaluation.

2.3.5 Hierarchical Constraints

If the user is working with several applications using the network concurrently, perhaps a Web browser and a word processor, we 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. We introduce a new operator: \rightarrow . Following the above example, imagine we wish to give four times the bandwidth to the Web browser over the word processor, and then within each application give three times the bandwidth to text over images. The EACS policy would be written:

$$\begin{aligned}
rule_1 &= \mathbf{BandwidthRatio}(\mathbf{select}(\Omega, \text{type} == \text{"html"}), \\
&\quad \mathbf{select}(\Omega, \text{type} == \text{"doc"}), 4); \\
rule_2 &= \mathbf{BandwidthPerElementRatio}(\mathbf{select}(\Omega, \text{type} == \text{"text"}), \\
&\quad \mathbf{select}(\Omega, \text{type} == \text{"image"}), 3); \\
rule_1 &\rightarrow rule_2;
\end{aligned} \tag{2}$$

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 to say:

$$rule_2 \rightarrow rule_1 \tag{3}$$

2.3.6 Composition

The language described thus far allows for a wide range of policies to be expressed. One of our 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 pre-defined policies provided by a policy designer, perhaps through a friendly user interface. Our system supports two mechanism for policy composition: *concatenation* and *hierarchy*. Both are quite simple. First, we add new language syntax to name policies and give them separate name spaces:

$$\begin{aligned}
&\text{Policy } P_1 \{ \\
&\quad \mathbf{BandwidthRatio}(\dots); \\
&\quad \dots \\
&\} \\
&\text{Policy } P_2 \{ \\
&\quad \mathbf{BandwidthRatio}(\dots); \\
&\quad \dots \\
&\}
\end{aligned} \tag{4}$$

Next, we only need two operators:

$$\begin{aligned}
\text{Policy } P_{concatenation} &= P_1 + P_2; \\
\text{Policy } P_{hierarchy} &= P_1 \rightarrow P_2;
\end{aligned} \tag{5}$$

Composition via concatenation simply computes the union of the constraints from each policy. This might yield an over-constrained system, but we already have mechanisms to resolve such policies (see 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 we already have. As a result of these two simple operators, EACS supports an easy and comprehensible mechanism to compose arbitrary bandwidth policies.

3 Policy Resolution

We show how EACS resolves policies, including cases where a given policy might be over- or under-constraining on the solution space. We also discuss 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. These shares would then be used as input to a low-level packet scheduling 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 to other arbitrary subsets of objects, we need a robust methodology for resolving the constraints and deriving these bandwidth shares.

It can be observed that usually a policy does not need to semantically differentiate individual objects, but rather treats 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 constraints 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 (see section 3.3). Objects within a partition are transmitted in round-robin fashion allowing further complexity reduction on the level of the network scheduler.

We 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 **Allocate** operators are resolved in the order determined by the first pass. Each **Allocate**(s, x)

operator assigns $\frac{x}{BW}$ bandwidth share to partition s , where BW is the current estimation of the available bandwidth, decrements the amount of the available bandwidth and removes objects of s from further consideration. If there is not enough available bandwidth, the rest of the available bandwidth is allocated. Similarly, **Never**(s) operator disables transmission of objects in set s and removes them from further consideration.

At the next stage, we must resolve the priority constraints 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 where a target is not compiled until all of its prerequisites are satisfied. We can view the priority constraints 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, we must resolve the proportional constraints and hierarchy constraints. 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 1. Lastly, we must add some equations that serve as sanity checks. We wish to solve the above constraints subject to the sum of the bandwidth shares being equal to 1.0. If there exists a unique solution to this problem, we can find it in $O(N^2)$ time, in the number of variables, using a simple substitution technique; we set the bandwidth of the first object to 1.0, then start looping over all $N * (N - 1)$ 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. We note that N is the number of partitions, 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., $a = b/2$ and $a = 2b$). It's 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, we must use more expensive techniques to solve for the bandwidth shares because there is no longer a single, correct answer.

Priority contradictions If the priorities cannot be resolved, that implies there must be a cycle in the priority graph. Our solution is to merge nodes in the cycle until the cycle no longer exists. When we merge two nodes 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, we must define how to evaluate hierarchical constraints over arbitrary graphs. First, we must remove cycles, as we did with priority contradictions. After this, we 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. Our solution to this problem also works well for under-constrained problems. In an over-constrained situation, we can have 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 $b = a/2$ generated by the solver. Merging nodes together would not necessarily give desirable results.

Instead, we consider the constraints to be goals which must be achieved. For the above example, we now wish to minimize the following equation:

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

subject to the constraints:

$$\begin{aligned} a + b + c &= 1 \quad (\text{allocate all available bandwidth}) \\ a, b, c &> 0 \quad (\text{avoid degenerate solutions}) \end{aligned} \tag{6}$$

In the case that, for some of the possible relationships, we have no proportional constraints, then 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, we synthesize new constraints that, barring anything else, should make them equal. We don't want the synthetic constraints to interact poorly with the original constraints, so we must scale down their effect, minimizing the following constrained equation:

$$\begin{aligned} \text{Error} = & \\ & (a - 2b)^2 + (c - 3d)^2 + (b - a/2)^2 + \\ & (d - c/3)^2 + K(a - c)^2 + K(b - d)^2 \end{aligned}$$

where K is a tunable parameter between zero and one.

The solution of this 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 the Ω that 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 which require the policy to be reevaluated to obtain the set of bandwidth shares consistent with the policy specification at the 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 that the policy cares about (e.g., the user moved a different application to the foreground). All of these events would require the transmission policy to be reevaluated.

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

```
Set s = select(  $\Omega$ ,
               type == "image"&&
               (sizeDone < 1/7 * sizeOriginal));
(7)
```

As objects in this set were 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 when 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.

Name	Type	Size	bg/fg
doc1.txt1	text	5 Kbyte	foreground
doc1.img1	image	60 Kbyte	foreground
doc1.img2	image	110 Kbyte	foreground
doc2.txt1	text	26 Kbyte	background
doc2.img1	image	30 Kbyte	background
doc2.img2	image	240 Kbyte	background

Figure 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.

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. We found that the value of 2 seconds worked very well for all our experiments on a 56 Kbit/sec network. Though 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 our 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. It is important to emphasize that the complexity of the solver is quadratic and can be even cubic in terms of the number of partitions of the Ω , but 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 10-20, 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 in our experiments) *not* increasing the overhead significantly.

4 Evaluation

In this section we first use simulation to demonstrate how the EACS policy resolver works. We then present performance measurements for a sample EACS policy running on the Puppeteer adaptation system.

4.1 Simulation

We give the simulation results of our implementation with two EACS policies that set different strategies for the transmission of a small sets of components from two different documents. Table 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. The results we present in the following sections assume a bottleneck bandwidth of 56 Kbit/sec.

```

// define the minimum-size text component
Set  $s_1 = \mathbf{min}(\mathbf{select}(\Omega, \text{type} == \text{"text"}), \mathbf{sizeOriginal})$ ;
// define other components
Set  $s_2 = \Omega \text{ SUB } s_1$ ;
// transmit the minimum-size text first
rule $_1 = \mathbf{Before}(s_1, s_2)$ ;

```

Figure 2: *Text First* EACS code.

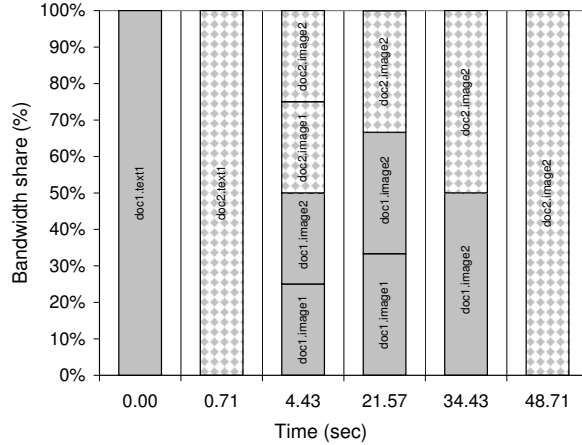


Figure 3: Simulation results for *Text First* policy.

4.1.1 Text First

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

Figure 2 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 component. Finally, the last statement ensures that all elements of s_1 are transferred before any elements of s_2 .

Figure 3 shows the simulator’s output for the *Text First* policy. The figures 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 we discuss in this section, EACS reconfigures the transmission policy, starting a new time step, only after some component finishes transmission.

Figure 3 shows that in the first time step, all 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 take to transfer the

```

Set  $s_1 = \mathbf{select}(\Omega, \text{foreground})$ ;
Set  $s_2 = \mathbf{select}(\Omega, \text{! foreground})$ ;
Set  $s_3 = \mathbf{select}(\Omega, \text{type} == \text{"text"})$ ;
Set  $s_4 = \mathbf{select}(\Omega, \text{type} == \text{"image"})$ ;
rule $_1 = \mathbf{BandwidthRatio}(s_1, s_2, 2)$ ;
rule $_2 = \mathbf{BandwidthPerElementRatio}(s_3, s_4, 5)$ ;
rule $_1 \rightarrow \text{rule}_2$ ;

```

Figure 4: *Focus* EACS code.

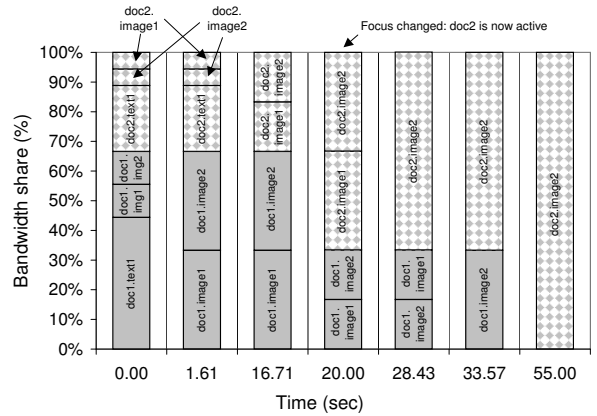


Figure 5: Simulation results for *Focus* policy.

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.

4.1.2 Focus

This 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 belong to the background document. Within the previous bounds, the policy then gives four times more bandwidth to each text component than to each image components.

Figure 4 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,

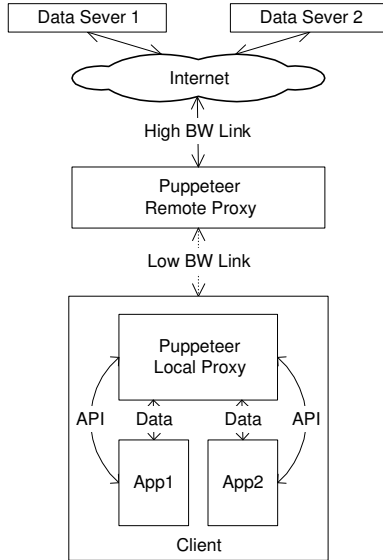


Figure 6: Puppeteer architecture.

and second on the component’s type.

Figure 5 shows the simulator’s output for the *Focus* policy. At the simulation’s beginning, Document1 is on the foreground and its components get $2/3$ of the available bandwidth. This bandwidth is further split between Document1’s text and image components, giving Document1’s text component $4/9$ of the system-wide available bandwidth ($2/3 * 4/6 = 4/9$) and $1/9$ of the system-wide available bandwidth to each of the two images. After 20 seconds, Document2 moves to the foreground and EACS reconfigures the transmission policy shifting $2/3$ of the bandwidth to Document2.

4.2 Performance

We measured the performance of EACS with a proof-of-concept implementation on top of the Puppeteer component-based adaptation system [6]. In the rest of this section, we first describe how our EACS prototype fits in the Puppeteer architecture. We then present experimental results for a sample transmission scheduling policy.

4.2.1 Puppeteer

Puppeteer adapts component-based 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 application, such as those in the Microsoft Office or Sun OpenOffice suits, by providing to the applications transformed versions of documents which 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 6 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 its runtime API. The remote proxy runs at the other end of the bandwidth-limited device 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^2Q+ Packet Fair Queuing (PFQ) algorithm [14], to distribute bandwidth among the partitions according to their rate allocations; round-robin schedulers further split bandwidth among components within each partition.

4.2.2 Text First

We quantify the performance of the *Text First* + EACS transmission policy presented on figure 7 by simultaneously loading an image-rich 668 KB Web page and a 1.05 MB PowerPoint presentation using Internet Explorer 5.5 (IE5) and Microsoft PowerPoint (PPT). The only difference of this policy from the one presented in section 4.2.2 is *rule₂* that enforces equal bandwidth distribution between IE5 and PPT components of the same type.

For this experiment we assume that the Web page is requested first and that a few seconds later the user starts downloading the PowerPoint presentation; while some of the images of the Web page are still being transferred. 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.

Our objective is to minimize the time that it takes to get all the 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.

We ran our experiments on a platform consisting of two Pentium III 500 MHz and one Athlon 1.2 GHz run-

```

Set  $s_1 = \text{select}(\Omega, \text{type}=="\text{text}");$ 
Set  $s_2 = \Omega \text{ SUB } s_1;$ 
 $\text{rule}_1 = \text{Before}(s_1, s_2);$ 
Set  $s_3 = \text{select}(\Omega, \text{application}=="\text{html}");$ 
Set  $s_4 = \text{select}(\Omega, \text{application}=="\text{ppt}");$ 
 $\text{rule}_2 = \text{BandwidthRatio}(s_3, s_4, 1);$ 

```

Figure 7: *Text First* + EACS code.

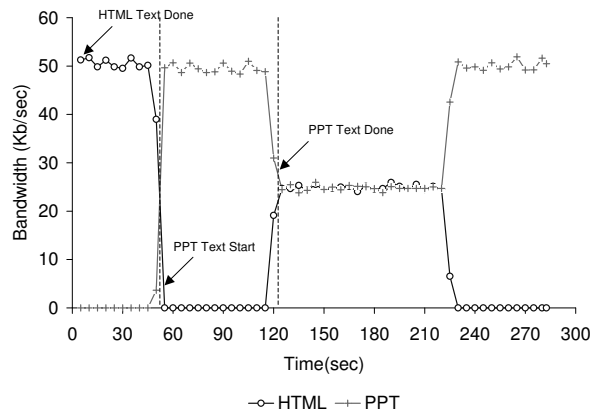


Figure 8: Bandwidth allocations for loading a Web page and a PowerPoint presentation with the EACS *Text First* policy.

ning Windows 2000. We configured the two Pentium III 500 MHz machines as: a data server running Apache 1.3, which stores the two documents we use in our experiments; and a client that runs the user’s applications and the Puppeteer local proxy. We ran the Puppeteer remote proxy and our EACS resolver on the Athlon 1.2 GHz. The local and remote Puppeteer proxies communicate via another PC running the DummyNet network simulator [15]. 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.

Figure 8 shows the bandwidth allocations for loading the two documents over a 56 Kbit/sec network link. The figure shows that *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 the Puppeteer; we measured 90% bandwidth utilization, the same as the Puppeteer achieves without EACS.

Policy	Lines of code
text-first (or focus) (see figures 2 and 4)	5
text-first+ (see figure 7)	9
allocate 32 Kbit/sec to audio stream	5
low fidelity images first	8
max 30 sec per web page	18

Figure 9: Script sizes for several EACS policies.

4.3 Policy expression efficiency

Figure 9 shows the sizes of sample policy scripts that we implemented and tested on top of the Puppeteer. Implementation of equivalent policies in Java requires several hundred lines of code. In contrast, the EACS policy scripts are much shorter and do not deal with the details of the adaptation system.

5 Related Work

In HATS [13], we experimented with combining dynamic control over bandwidth scheduling and adaptation. While this combination enabled us to adapt multiple applications in concert (our intended purpose), it required coding transmission scheduling policies in Java. As a result, a significant programming effort was needed to implement 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, we can 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 [16, 17], provides differentiated service in network hierarchies [14, 18], or adds differentiated services to general purpose operating systems [19, 20, 21, 22]. EACS is fundamentally built on the concept of solving constraints, which is also an area that has been extensively studied [23].

6 Conclusions

In this paper, we 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, our 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 are free to 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, when executing bandwidth policies with real data, we observe very little system overhead.

In the future, there are 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. We also would like to investigate how to add notions of object subsets and transformations (removing or transcoding objects) into the EACS policy language as well as how and when to update user's application data. Using the composition mechanisms in EACS, we 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.

References

- [1] R. Bagrodia, W. W. Chu, L. Kleinrock, and G. Popek, "Vision, issues, and architecture for nomadic computing," *IEEE Personal Communications*, vol. 2, pp. 14–27, Dec. 1995.
- [2] R. H. Katz, "Adaptation and mobility in wireless information systems," *IEEE Personal Communications*, vol. 1, no. 1, pp. 6–17, 1994.
- [3] M. Satyanarayanan, "Fundamental challenges in mobile computing," in *Fifteenth ACM Symposium on Principles of Distributed Computing*, (Philadelphia, Pennsylvania), May 1996.
- [4] M. Weiser, "Some computer science problems in ubiquitous computing," *Communications of the ACM*, July 1993.
- [5] M. Satyanarayanan, "Pervasive computing: Vision and challenges," *IEEE Personal Communications*, 2001.
- [6] 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.
- [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*, vol. 31, pp. 160–170, Sept. 1996.
- [8] 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.
- [9] 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)*, vol. 51, pp. 276–287, Dec. 1997.
- [10] J. 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, South Carolina), Dec. 1999.
- [11] H. Chu, H. Song, C. Wong, and S. Kurakake, "Seamless applications over roam system," in *Proceedings of the Workshop on Application Models and Programming Tools for Ubiquitous Computing (UbiTools'01)*, (Atlanta, Georgia), Sept. 2001.
- [12] K. Takashio, M. Mori, and H. 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 Ubiquitous Computing (UbiTools'01)*, (Atlanta, Georgia), Sept. 2001.
- [13] E. de Lara, D. S. Wallach, and W. Zwaenepoel, "Hats: Hierarchical adaptive transmission scheduling," in *Multimedia Computing and Networking*, (San Jose, California), Jan. 2002.
- [14] J. Bennett and H. Zhang, "Hierarchical packet fair queuing algorithms," in *Proceedings of SIGCOMM'96*, (Stanford, California), Aug. 1996.
- [15] L. Rizzo, "DummyNet: a simple approach to the evaluation of network protocols," *ACM Computer Communication Review*, vol. 27, pp. 13–41, Jan. 1997.
- [16] 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.
- [17] L. Zhang, S. Deering, and D. Estrin, "RSVP: A new resource ReSerVation Protocol," *IEEE Network*, vol. 7, pp. 8–18, Sept. 1993.
- [18] S. Floyd and V. Jacobson, "Link-sharing and resource management models for packet networks," *IEEE/ACM Transactions on Networking*, vol. 3, pp. 365–386, Aug. 1995.
- [19] D. P. Anderson, "Metascheduling for continuous media," *ACM Transactions on Computer Systems*, vol. 11, pp. 226–252, Aug. 1993.
- [20] W. Almesberger, "Linux network traffic control - implementation overview," Apr. 1999. <http://lrcwww.epfl.ch/linux-diffserv/>.
- [21] S. Chen and K. Nahrstedt, "Hierarchical scheduling for multiple classes of applications in connection-oriented integrated-service networks," in *Proceedings of the IEEE International Conference on Multimedia Computing and Systems (ICMCS)*, (Florence, Italy), June 1999.
- [22] 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.
- [23] E. P. K. Tsang, *Foundations of Constraint Satisfaction*. London and San Diego: Academic Press, 1993. ISBN 0-12-701610-4.