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

Applications running on a mobile and wireless devices must be able to adapt gracefully to limited and fluctuating network resources. The variety of applications, platforms upon which they run, and desires of their users, require a variety of adaptation policies to be implemented and maintained. Therefore, it becomes important for adaptation policies to be easy to develop, to debug, and to compose together to form complex policies that satisfy the needs of mobile users.

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 the relative bandwidth allocations across objects of different types. 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 [3, 13, 17]. The advent of pervasive computing [22, 18] 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 [7, 12, 14, 15], energy supply [10], and device heterogeneity [6, 20]. 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 application 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 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, 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 most 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 low level configuration parameters (e.g., fidelity levels, scheduling shares, priorities).

This paper presents Extensible Adaptation via Constraint Solving (EACS), a novel approach that simplifies policy design by distancing users from the intricacies of the adaptation mechanisms. In EACS, users specify adaptation policies by defining subsets of the objects and defining constraints among these subsets.

The initial EACS prototype is limited to transmission policies, which define the order in which a series of objects are 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 elements of set

B) and 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*).

We first developed an EACS simulator, allowing us to run 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 [8], 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 [7] and we then measured the overhead of running EACS transmission policies verus Puppeteer's hand-tuned transmission policy implementations, showing performance comparable, despite EACS policies being dramatically shorter and more abstract.

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 synthesizing high-level EACS descriptions into low-level bandwidth shares that can be feed to an adaptation system. Section 4 presents results from simulations and runs on the Puppeteer adaptation system that evaluate the correctness and performance of several EACS policies. 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 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 chose 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 ordering, assuming that other portions of the adaptation system deal with subsetting and transformations. 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, allow 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, ...), or it's *context*. 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* as a first-class type, providing all the usual first-order logic predicates, such as set 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%).

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

In-order constraints allow for objects in a given set to be transmitted in a sequential order (versus concurrent, interleaved transmission).

Deadlines express how certain objects must arrive before a certain specific time.

In our current implementation, we do not yet have support for deadlines and in-order constraints, although we plan to add them in the future. This paper currently discusses only the operators we have implemented.

2.2 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*.

- 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 might be

application describes the application requesting the object

sizeDone bytes of the object that have been transmitted

sizeOriginal total bytes for the object

2.2.1 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).

2.2.2 Proportional Constraints

We can also express proportional constraints to describe how bandwidth must be shared between objects. We note that these rules are only evaluated against objects that are ready to send (i.e., an object with a priority constraint that prevents its current transmission will not be allocated any bandwidth by a proportional constraint operation).

- **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:

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

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.

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

$$\begin{aligned}\textbf{BandwidthPerElementRatio}(s_1, s_2, r) \quad &\equiv \\ \textbf{BandwidthRatio}\left(s_1, s_2, r\tfrac{|s_1|}{|s_2|}\right) &\end{aligned} \tag{1}$$

One should use the **BandwidthRatio** operator 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.

2.2.3 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:

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

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:

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

2.2.4 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. 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:

$$\begin{aligned} &\text{Policy } P_1\{ \\ &\qquad \textbf{BandwidthRatio}(...); \\ &\qquad ... \\ &\} \\ &\text{Policy } P_2\{ \\ &\qquad \textbf{BandwidthRatio}(...); \\ &\qquad ... \\ &\} \end{aligned} \tag{4}$$

Next, we only need two operators:

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

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. 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 Constraint Resolution

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.

First, we must resolve the priority constraints to determine the set of objects that might be transmitted. If a policy specified **After**(s_1, s_2), then the members of s_1 will be guaranteed to have no bandwidth allocated to them unless $s_2 = \emptyset$. We can view the priority constraints as specifying a *dependency graph* on the the objects to transmit. The set of objects allowed ready to be transmitted is equal to the set of objects with no other objects depending on them. This can be derived by making a linear pass over all the objects 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 members of s_1 is equal to r times the sum of the bandwidth of the members 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 must specify that "equivalent" objects will have the same bandwidth. 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^2 possible constraints, solving for the other variables, one at a time. This algorithm will also detect if the policy is over- or under-constrained by returning inconclusive results, requiring the use of more expensive techniques, as discussed below.

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 collapse nodes in the cycle until the cycle no longer exists. When we collapse 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 loop, searching 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 loop repeats. After the first group of constraints is resolved, the result is a partitioning of Ω into sets, each of which has its own bandwidth share. This set of subsets of Ω is the input to the next round of the loop. The subsequent constraints are then evaluated independently on each subset.

Proportional contradictions These are the most difficult over-constrained problems to solve. Our solution to this problem also works well for under-constrained problems. Assume there are N objects in the system. That means that there are N^2 possible proportional relationships among objects. 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. 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:

$$Error = (a-2b)^2 + (b-3c)^2 + (c-4a)^2 \qquad (6)$$

subject to the constraints:

$$\begin{array}{rcll} a+b+c & = & 1 & \textit{(allocate all available bandwidth)} \\ a,b,c & > & 0 & \textit{(avoid degenerate solutions)} \end{array} \qquad (7)$$

In the case that, for some of the N^2 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:

$$Error = (a-2b)^2 + (c-3d)^2 + K(a-c)^2 + K(b-d)^2 \tag{8}$$

where K is a tunable parameter between zero and one.

The solution of these quadratic minimization methods is straightforward. Gaussian elimination can derive the global minimum error for these function in $O(N^3)$ time where $N = |\Omega|$. If this proved to be too expensive, we could use more sophisticated methods, such as the $O(N^2)$ Jacobi iteration method. Performance issues are discussed more in section 3.4.

3.3 Policy Reevaluation

As the system is transmitting objects, various *events* can occur which require the policy to be reevaluated. 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:

$$\begin{aligned} \text{Set } s = \textbf{select}(\quad & \Omega, \\ & \text{type} == \text{"image"}\&\& \\ & \quad (\textbf{sizeDone} < 1/7 * \textbf{sizeOriginal}), \\ & 3); \end{aligned} \tag{9}$$

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. An alternate approach is to periodically reevaluate the transmission policy, perhaps once every few seconds, to discover, at run-time, when set membership changes. Such periodic reevaluations can potentially waste CPU time to only discover that nothing has changed.

3.4 CPU Efficiency

To make our system sufficiently general purpose, we must also be concerned that the CPU costs of policy reevaluation, even when the costs can be perfectly overlapped with data transmission, can have a potentially serious impact on overall system performance, particularly as transmission policies become increasingly complex. This leads us to favor designs where policy reevaluation still occurs when requested, but is delayed for a pre-specified amount of time. During this delay, other requests to reevaluate the policy are dropped. This way, if a large number of events occur in a small amount of time, there will be only one policy reevaluation. Policy writers should be aware of this, expecting that, for short periods of time, the system will not necessarily respect their policies precisely. One result of this is that policies which tend to select only small numbers of objects for transmission at any given time will experience more volatile behavior than policies which select large numbers of objects for transmission; the large number of objects helps insulate the system from the effect of any one object completing its transmission.

Furthermore, the growth in the number of objects in Ω could lead to a significant growth in the runtime for evaluating the bandwidth shares, particularly with over-constrained policies that might require $O(|\Omega|^3)$ runtime. We observe that, for many common policies, there exist a large class of *equivalent* objects. If two objects in Ω have exactly the same attributes, they will have the same bandwidth shares allocated to them. This leads to an important optimization: we can treat all the objects in a given equivalence-class as a single object when solving the constraints. Of course, we must scale the coefficients appropriately. As a result of this optimization, common policies which tend to allocate bandwidth among only a small number of sets (perhaps text vs. non-text data) can now run in time proportional to the number of sets rather than the number of individual objects. This makes policy resolution runtime proportional really to complexity of the policy itself, rather than to the number of objects being scheduled by the policy.

We can further take advantage of these equivalence classes by putting all the objects in a given equivalence class into the same queue for the low-level packet scheduler, using the bandwidth for the shared queue in a round-robbin fashion. This simplifies the number of queues the packet scheduler must manage, which can potentially result in efficiency gains there, as well.

4 Evaluation

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

Name	Type	Size	bg/fg
doc1.text1	text	5Kb	fg
doc1.img1	image	60Kb	fg
doc1.img2	image	110Kb	fg
doc2.text1	text	26Kb	bg
doc2.img1	image	30Kb	bg
doc2.img2	image	240Kb	bg

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.

4.1 Correctness

We evaluate the correctness 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 128 Kb/sec.

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 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 0.31 seconds — the time that it take to transfer the 5 KB of text over the simulated 128 Kb/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 $s_1 = \textbf{min}($  $\textbf{select}(\Omega, \text{type} == \text{"text"})$,
                          $\textbf{sizeOriginal})$;
// define other components
Set $s_2 = \Omega$ SUB $s_1$;
// transmit the minimum-size text first
$rule_1 = \textbf{Before}(s_1, s_2)$;
```

Figure 2: *Text First* EACS code.

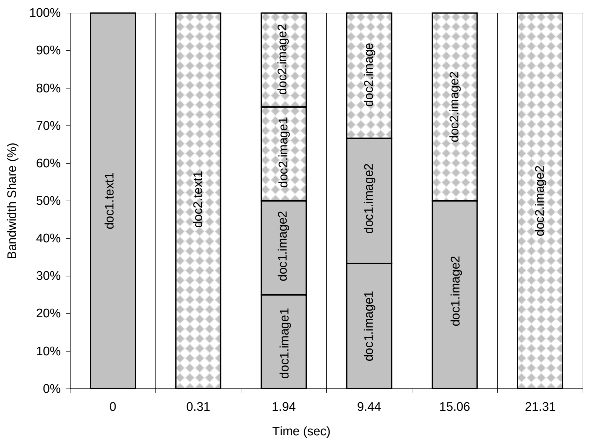

Figure 3: Simulation results for *Text First* policy.

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 the 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 the 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 5 shows the simulator's output for the *Focus* policy. An 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

```
Set $s_1$ = **select**($\Omega$, fg);
Set $s_2$ = **select**($\Omega$, ! fg);
Set $s_3$ = **select**($\Omega$, type==”text”);
Set $s_4$ = **select**($\Omega$, type==”image”);
$rule_1$ = **BandwidthRatio**($s_1$, $s_2$, 2);
$rule_2$ = **BandwidthPerElementRatio**($s_3$, $s_4$, 5);
$rule_1 \rightarrow rule_2$;
```

Figure 4: *Focus* EACS code.

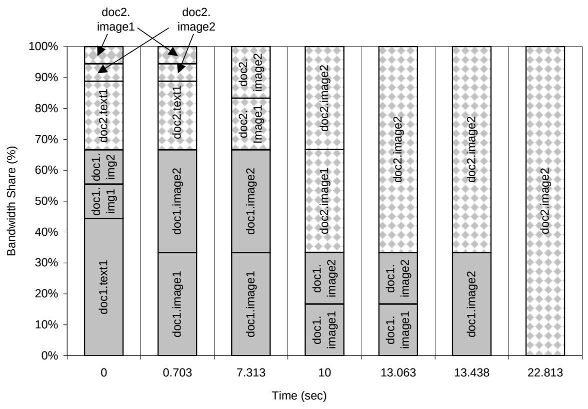

Figure 5: Simulation results for *Focus* policy.

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 10 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 [7]. 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-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 application, such as those in the Microsoft Office or Sun’s 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.

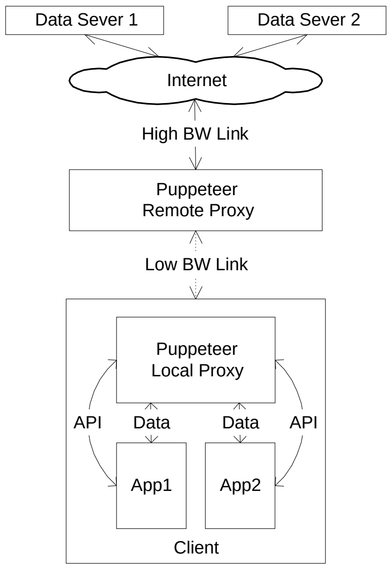

Figure 6: Puppeteer architecture.

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 run-time 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 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 [4], to distribute bandwidth among the component subsets according to their rate allocations.

4.2.2 Text First

We quantify the performance of the *Text First* EACS transmission policy presented in section 4.1.1 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).

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

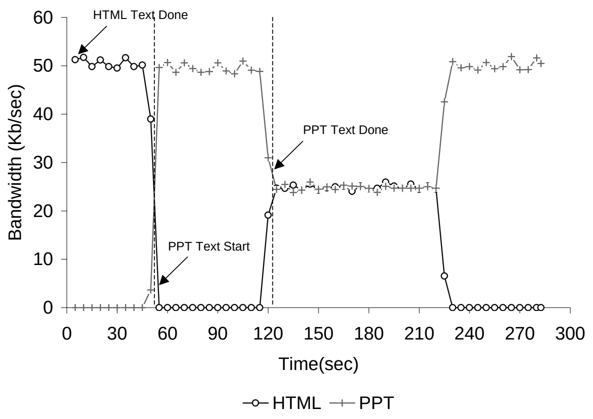

Figure 7: Bandwidth allocations for loading a Web page and a PowerPoint presentation with the EACS *Text First* policy.

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; 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 running 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 [16]. 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.

Figures 7 show the bandwidth allocations for loading the two documents over a 56 Kb/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 generated transmission policy achieves an average of 90% network utilization, and comes within 3% of the network utilization achieved by a similar hand-tuned transmission policy.

5 Related Work

In HATS [8], 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 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, we can write a 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 [9, 23], provides differentiated service in network hierarchies [4, 11], or adds differentiated services to general purpose operating systems [2, 1, 5, 19]. EACS is fundamentally built on the concept of solving constraints, which is also an area that has been extensively studied [21].

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 difference in throughput between EACS and hand-tuned implementations of the same policies.

In the future, there are a number of additional features that would be beneficial to study with EACS. We would like to study whether GUIs or other techniques could aid unsophisticated users in selecting appropriate bandwidth 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.

References

[1] ALMESBERGER, W. Linux network traffic control - implementation overview, Apr. 1999. `http://lrcwww.epfl.ch/linux-diffserv/`.

[2] ANDERSON, D. P. Metascheduling for continuous media. *ACM Transactions on Computer Systems 11*, 3 (Aug. 1993), 226–252.

[3] BAGRODIA, R., CHU, W. W., KLEINROCK, L., AND POPEK, G. Vision, issues, and architecture for nomadic computing. *IEEE Personal Communications 2*, 6 (Dec. 1995), 14–27.

[4] BENNETT, J., AND ZHANG, H. Hierarchical packet fair queuing algorithms. In *Proceedings of SIGCOMM'96* (Stanford, California, Aug. 1996).

[5] CHEN, S., AND NAHRSTEDT, K. 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).

[6] CHU, H., SONG, H., WONG, C., AND KURAKAKE, S. Seamless applications over roam system. In *Proceedings of the Workshop on Application Models and Programming Tools for Ubiquitos Computing (UbiTools'01)* (Atlanta, GA, Sept. 2001).

[7] DE LARA, E., WALLACH, D. S., AND ZWAENEPOEL, W. Puppeteer: Component-based adaptation for mobile computing. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems* (San Francisco, California, Mar. 2001).

[8] DE LARA, E., WALLACH, D. S., AND ZWAENEPOEL, W. Hats: Hierarchical adaptive transmission scheduling. In *Multimedia Computing and Networking* (San Jose, California, Jan. 2002).

[9] FERRARI, D., RAMAEKERS, J., AND VENTE, G. Client-network interactions in quality of service communication environments. In *Proceedings of the 4th IFIP Conference on High Performance Networking* (Liege, Belgium, Dec. 1992).

[10] FLINN, J., AND SATYANARAYANAN, M. Energy-aware adaptation for mobile applications. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)* (Kiawah Island Resort, SC, Dec. 1999).

[11] FLOYD, S., AND JACOBSON, V. Link-sharing and resource management models for packet networks. *IEEE/ACM Transactions on Networking 3*, 4 (Aug. 1995), 365–386.

[12] FOX, A., GRIBBLE, S. D., BREWER, E. A., AND AMIR, E. Adapting to network and client variability via on-demand dynamic distillation. *SIGPLAN Notices 31*, 9 (Sept. 1996), 160–170.

[13] KATZ, R. H. Adaptation and mobility in wireless information systems. *IEEE Personal Communications 1*, 1 (1994), 6–17.

[14] KOSNER, R., AND KRAMP, T. Structuring QoS-supporting services with smart proxies. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)* (New York, New York, Apr. 2000).

[15] NOBLE, B. D., SATYANARAYANAN, M., NARAYANAN, D., TILTON, J. E., FLINN, J., AND WALKER, K. R. Agile application-aware adaptation for mobility. *Operating Systems Review (ACM) 51*, 5 (Dec. 1997), 276–287.

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

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

[18] SATYANARAYANAN, M. Pervasive computing: Vision and challenges. *IEEE Personal Communications* (2001).

[19] SUNDARAM, V., CHANDRA, A., GOYAL, P., SHENOY, P., SAHNI, J., AND VIN, H. Application performance in the QLinux multimedia operating system. In *Proceedings of the Eighth ACM Conference on Multimedia* (Los Angeles, California, Nov. 2000).

[20] TAKASHIO, K., MORI, M., AND TOKUDA, H. 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, Sept. 2001).

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

[22] WEISER, M. Some computer science problems in ubiquitous computing. *Communications of the ACM* (July 1993).

[23] ZHANG, L., DEERING, S., AND ESTRIN, D. RSVP: A new resource ReSerVation Protocol. *IEEE Network 7*, 5 (Sept. 1993), 8–18.