User-specified Adaptive Scheduling in a Streaming Media Network[*]

Michael Hicks, Adithya Nagarajan
Department of Computer Science
Univ. of Maryland, College Park, MD
mwh@cs.umd.edu, sadithya@cs.umd.edu

Robbert van Renesse
Department of Computer Science
Cornell Univ., Ithaca, NY
rvr@cs.cornell.edu

Abstract

In disaster and combat situations, mobile cameras and other sensors transmit real-time data, used by many operators and/or analysis tools. Unfortunately, in the face of limited, unreliable resources, and varying demands, not all users may be able to get the fidelity they require. This paper describes *MediaNet*, a distributed multi-media processing system designed with the above scenarios in mind. Unlike past approaches, MediaNet's users can intuitively specify how the system should adapt based on their individual needs. MediaNet uses both local and online global resource scheduling to improve user performance and network utilization, and adapts without requiring underlying support for resource reservations. Performance experiments show that our scheduling algorithm is reasonably fast, and that user performance and network utilization are both significantly improved.

[*]This work was funded in part by DARPA grant F30620-98-2-0198, DARPA/AFRL-IFGA grant F30602-99-1-0532, a grant under NASA's REE program administered by JPL, NSF-CISE grant 9703470, and by the AFRL/Cornell Information Assurance Institute. The first author was also supported by the AFRL-IFGA Information Assurance Institute under grant AFOSR F49620-01-1-0312.

1 Introduction

Consider a dangerous setting, such as collapsed buildings caused by an earthquake. Novel recording devices, such as cameras carried by Uninhabited Aerial Vehicles (UAVs) or by robots that crawl through rubble, may be deployed to explore the area. The output of these devices can be of interest to many operators. Operators may include rescue workers working in the rubble itself, people overseeing the work in a station somewhere, the press, or software that creates, say, a 3-dimensional model of the scene.

Different operators may require different views of the area, and may have different fidelity requirements or user priorities. Although the operators may work independently of one another, they share many resources, such as the recording devices themselves, compute servers, and networks. These resources have limited capacity, and thus it is necessary to allocate them carefully. Without resource reservation, adaptivity is essential.

The conditions present in this disaster situation are not unique. That is, many applications consist of multiple operators interested in streaming data from multiple sources that must adapt to limited resources, potentially in application-specific ways. Examples include the exchange and aggregation of sensor reports [19], the distribution of media on a home network [30], the performance of reconnaissance and deployment in a military setting [21], and so on.

A number of projects have explored how to provide improved quality of service (QoS) for streaming data in resource-limited conditions. These systems place computations in the network, either within routers themselves (e.g., [5, 11, 34]) or at the application-level using an overlay network (e.g., [1, 32]), and employ system-determined, local adaptations, such as priority-based video frame dropping. While such adaptations impose little overhead, they can be inefficient because they do not take into account global information. Also, existing schemes typically do not consider user preferences in making QoS decisions.

To study whether these problems can be overcome, we are developing a system called *MediaNet* that takes a comprehensive view of streaming data delivery. MediaNet mainly differs from past approaches in two ways. First, rather than making QoS adaptation system-determined, MediaNet allows users to specify how it should adapt under overload conditions. Each user contributes a list of alternative specifications, and associates a utility value with each specification. To some users, color depth may be more important than frame rate, while for other users the preference may be the other way around. The primary goal of MediaNet is to maximize each user's utility.

Second, in addition to using local scheduling, MediaNet employs a *global scheduling service* to divide tasks and flows among network components. This global point of view has benefits to both fairness and performance, because the service can consider specifications from multiple users while accounting for priority and overall network efficiency; the challenge is to do this in a scalable manner. Different from other projects that use global schedulers (e.g., [11, 15, 30]), MediaNet's global scheduling service is continuously looking for improvements based on monitoring feedback. MediaNet employs a completely adaptive overlay network; it does not rely on resource reservations, and adapts to the presence of loads not under its control.

Experimental measurements with our prototype implementation are promising. When using a single global scheduler to implement the global scheduling service, users achieve better performance and the network is more efficiently utilized than without any or with only local adaptations. On the other hand, our system does exact a higher cost for its global adaptations, in terms of scalability and implementation complexity. We consider our work as a step to exploring how to apply adaptations synergistically from various levels in a scheduling hierarchy.

In this paper, we present the MediaNet architecure (Section 2) and our prototype implementation (Sections 3 and 4). We focus on the challenges of implementing a globally-reconfigurable stream-processing system, and show experimental evidence of its costs and benefits (Section 5). We finish up by comparing our approach to related work (Section 6) and propose future research directions (Section 7).

2 MediaNet

MediaNet's architecture defines a *computational network*, consisting of *compute nodes* and *network links*. These elements are responsible for receiving streaming data from various sources, computing on that data, and delivering it to the

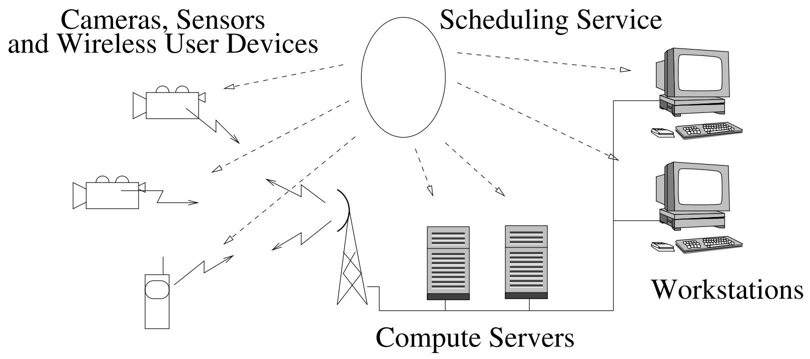


Figure 1: MediaNet architecture.

end-applications. As shown in Figure 1, compute nodes are highly heterogeneous, consisting of cameras, sensors, workstations, and compute servers; as such they have different computational power, available memory, hardware support for video operations, etc. Network links between nodes could be either wired or wireless; as such, the underlying network topology may change at run-time as components physically move around or new parts of the infrastructure are deployed.

The user's interface to this computational network is via a *global scheduling service*; the architecture leaves the implementation of this service abstract. Users communicate their requirements to the service using specifications that consist of what we call *continuous media networks* (CMNs). A CMN is simply a directed acyclic graph (DAG) representing a computational dataflow. The job of the global scheduling service is to combine the CMNs of individual users into a single CMN, and then partition this CMN into subgraphs to be executed on the various compute nodes, based on the current state of the network. The act of combining the CMNs and partitioning them among nodes takes into account issues of fairness, performance, and user-specified adaptation. We elaborate on user specifications next, and follow with a discussion of scheduling.

2.1 Specifications

Each node in a CMN represents an operation mapping zero or more input *frames* (stream-specific packets of data such as video frames, audio clips, etc.) to zero or more output frames. Operations can be simple, *e.g.*, data forwarding, frame prioritizing, and frame dropping; or they can be more complex, *e.g.*, video frame cropping, changing the resolution or color depth, "picture-in-picture" effects, compression, encryption, and audio volume control. We also need operations to receive input from and send output to components external to the DAG, to perform I/O with devices like video cameras and players. The global scheduling service takes into account the bandwidth, latency, and processing requirements of operations.

Operations have a number of associated attributes. One important attribute is the *interval* that indicates the minimum time between operations on subsequent frames (*i.e.*, the inverse of the maximum rate). For better performance, operations can either process input frames immediately, or they can be forced to execute at the specified intervals on queued data. In either case, the interval effectively specifies a soft real-time constraint on the processing of frames; if frames arrive faster than the specified interval, or if the node cannot process them at that interval (perhaps because of downstream congestion), then either backpressure must be applied to the incoming flow or frames must be dropped. How to handle these situations adaptively is considered in the next subsection.

A CMN node can be fixed at a certain location in the actual network (*e.g.*, to indicate the network location of a particular video source), or left unspecified. Moreover, a node can be considered *transitional*, meaning that it is only inserted between mandatory nodes when the CMN is scheduled across multiple compute nodes. Operations can maintain internal soft-state[1] and need not actually operate on packets. Requiring *soft*-state is important for allowing operations to relocate during a reconfiguration.

A user specification is a list of CMNs, where each CMN's relative preference is indicated by a corresponding *utility value*, which is a real number between 0 and 1, where 1 means most desirable. An example is shown in Figure 2(a)[2], where the user specifies three CMNs, having decreasing utility. In each CMN, an MPEG video stream originates at location `pcS`, the frames are prioritized for intelligent dropping by the transitional (as indicated by the `*`) `Prio` operation, and they are finally delivered to the user's player on `pcD`. In the second CMN, the frame rate is reduced by proactively dropping B frames, while in the third CMN the P frames are dropped as well.[3] The MediaNet scheduler can decide which of these specifications to run, and where to run the operations with unspecified locations.

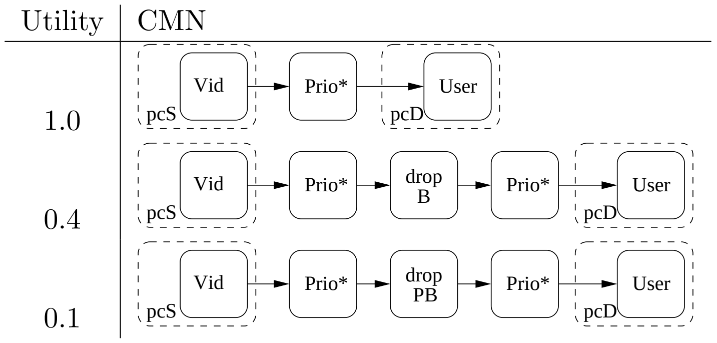
We do not expect users will author CMNs directly, but rather provide higher-level preferences, such as the general adaptation methodology and the streams of interest. For example, a user might specify (in some declarative format) "I want MPEG stream X from location Y, and I want to adapt using frame dropping." A *weaving* tool, which is part of the global scheduling service, would index these preferences into a database that contains template CMNs and stream specifications. The template CMNs are basically like the CMNs we have shown, but without any stream-specific data, like the stream location, resource usage characteristics, etc., while the stream specifications would include this missing information. The weaver then merges the template and the stream specification of the requested movie together to create an almost-complete CMN; only the utility values have not been filled in. This idea is shown in Figure 2(b).

The weaver should set utility values to share resources fairly among users of potentially differing priority. Utility values have both *relative* and *absolute* effect. That is, a user's alternative specifications are prioritized relatively by the ordering of their utilities, while the partic-

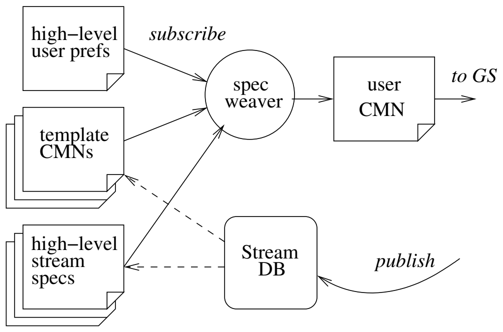
[1]Soft state is state not strictly required for correctness, e.g. caches.

[2]Though not shown here, we encode user specifications, and consequently CMNs, as XML documents.

[3]In MPEG streams, I frames are essentially JPEG images, while P frames and B frames exploit temporal locality, including "deltas" from adjacent frames. P frames rely on the most temporally-recent P or I frame, and B frames rely on the prior I or P frame, and the next appearing I or P frame. Therefore, I frames are more important than P frames, while B frames are the least important.



(a) Example User CMN



(b) MediaNet User API

Figure 2: User specification APIs

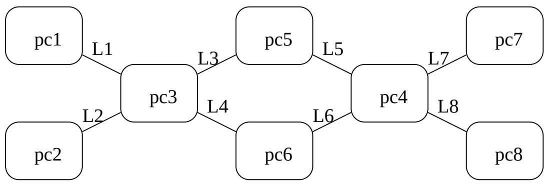
ular magnitude of a utility value relates globally to the utility values of other users. For example, a higher priority user might have the same specification as in Figure 2(a), but have utility values 1.0, 0.2, and 0.1, respectively. When resources became limited, this user would be forced to adapt only after a user having the utilities assigned in Figure 2(a). We expect to report on the implementation of this aspect of the MediaNet architecture in future work; in the meantime, our implementation assumes utilities are set fairly by hand.

2.2 Scheduling

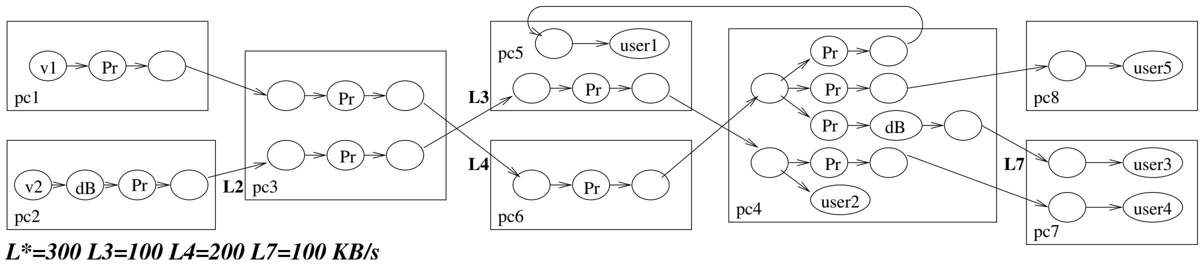
Once a user provides a specification, the global scheduling service schedules it on the network in conjunction with all existing user specifications. An example schedule generated by our prototype implementation is depicted in Figure 3. Here we have combined five user specifications of equal priority, each varying from that in Figure 2(a) only in the user and video locations, and scheduled them on a sample network. In Figure 3 the `v1` and `v2` nodes correspond to the video sources, the `Pr` nodes correspond to the frame priority-setting `Prio` operations, the `dB` node corresponds to the `drop B` operation. The empty circles are *send* and *receive* operations inserted by the global scheduling service to transport data between nodes. The available link bandwidths are 300 KB/s in general, with 200 KB/s on link `L4`, and 100 KB/s on links `L3` and `L7`. For both videos, the utility 1.0 configuration requires roughly 145 KB/s, and the utility 0.4 configuration requires roughly 90 KB/s.

The quality of a schedule can be evaluated by the provided per-user utility, and the network-wide utilization, in terms of CPU, memory, and bandwidth usage. Schedule evaluation in an absolute sense is difficult because the scheduling problem is almost certainly NP-complete, so generating an optimal schedule for comparison is not feasible in general. Therefore, we must assess schedules manually (if possible), or compare them with schedules from different algorithms.

For the example schedule, users 1, 3, and 5 are receiving the best possible performance: users 1 and 3 have utility 1.0 since they require no intervening `dB` nodes, while user 5 must have its B frames dropped, resulting in 0.4 utility, due



(a) A sample network



(b) A sample schedule on the above network

Figure 3: A global configuration

to the 100 KB/s limitation on its last hop link L7. Users 2 and 4 also receive utility 0.4, having their B frames dropped at pc2; this is the best that can be expected given the requirements of users 1, 3, and 5 and assuming that striping is not supported.[4] Given the user utilities it is supporting, the network utilization is good, as it is not wasting bandwidth. For example, video 2's dB node is scheduled at pc2 rather pc3, which is connected to the congested link; this avoids wasting bandwidth across link L2.

While other architectures with similar global scheduling services either set up only the initial computational flow [30], or reschedule very rarely (such as when compute nodes or network links fail), MediaNet's global scheduling service operates *on-line*, performing continuous scheduling. As such, the service needs regular reports of current conditions, including changes to link and CPU/memory loads, and changes to topology. Because of delays in detecting and reporting changing information, changes to the schedule necessarily occur on the order of seconds. To mitigate these delays, user specifications can employ local adaptations, like intelligent packet dropping or upstream backpressuring.

[4]It would be possible for users 2 and 4 to receive utility 1.0 *rather* than users 1 and 3, but this is an arbitrary decision given that all users are of equal priority.

3 Global Scheduling

The MediaNet architecture leaves the implementation of the global scheduling service abstract, admitting the possibility of a variety of implementations. The most straightforward implementation would be as a single *global scheduler* (GS) which computes a CMN subgraph for each node and sends it to the *local scheduler* (LS) running the node. The LS implements the CMN and periodically reports local resource usage to the GS, which can periodically recompute and redistribute its schedules, as necessary. This approach has the benefit that since the scheduler can consider the entire network and all of its users, it can likely achieve better fairness and performance, but at the cost of scalability. Conversely, a completely distributed approach would improve scalability but likely degrade performance.

We believe that the best approach will be to use a hierarchy of GSs, each responsible for subcomponents of the network and combined user CMNs. The users will provide their specifications to a top-level GS, which will aggregate all of the specifications and disseminate partitions of them to its child schedulers. These will do likewise, until ultimately a single CMN is provided to the LS for implementation on a compute node. Conversely, each LS will report its available resources to its parent GS, which will report aggregated resource amounts to its parent, and so on. Moreover, the hierarchy will be best created on-the-fly, depending on the size of the network, or its current state. For small networks (e.g. 5-15 nodes, with 5-10 users, as might be expected in the motivated disaster situation), a single GS will likely be ideal, while for larger networks, more hierarchy will reduce the system-wide effects of reconfiguration, reduce monitoring overhead, etc.

In this work, we describe a plausible algorithm for the GS in this hierarchical arrangement. Our current implementation uses only a single GS, and so we omit the details of how the hierarchy might be created on-the-fly, how the schedulers might partition user specifications, etc. These important details will be left to future work. In this section, we describe our global scheduling algorithm and characterize its running time. The next section describes our prototype implementation of this algorithm and the associated infrastructure.

3.1 Overview

The goal of any MediaNet global scheduling algorithm is to maximize the minimum utility of each of its users while utilizing network resources efficiently. In the case of our particular algorithm, given U user specifications, each with one or more CMNs having utility values between 0 and 1, the goal is to find a minimum utility sufficient for all users, and assign higher utility values for as many users as possible, possibly preferencing higher priority users. Secondarily, given the maximal aggregate utility it is able to achieve, it will choose the schedule that least taxes the network's resources.

The algorithm works as follows. Each user is assigned a utility, and the operations at that utility are combined with the other users' operations (at their respective utility) to create a single, global CMN of user operations. It then considers possible assignments of user operations to network hosts, along with the necessary intervening send and receive operations. For each assignment, it calculates a *score*, based on how effectively the user specifications are met and how efficiently the network is utilized. Each score is either negative, in which case the network does not have the resources to schedule the given operations, or the score is between 0 and 1, where 1 indicates the network is untaxed, and 0 indicates that at least some part of its resources (whether CPU, network bandwdith, etc.) is completely utilized. The assignment chosen is the one with (1) the highest minimum utility for all users, (2) the highest aggregate utility (i.e. the sum of all user utilities considered) above this minimum, and (3) the maximal score for that aggregate utility.

Our algorithm is not guaranteed to generate an optimal solution, but it is reasonably fast and works well on the examples we have considered (as we demonstrate in Section 5). In particular, we show in Section 3.3.1 that the algorithm is polynomial in the size of the network and the number of users, which is faster than a brute-force enumeration of possible schedules, which would be exponential in the size of the network. This speed is important since the algorithm will be run on-line, as network conditions and user specifications change over time. We first discuss our scoring algorithm, and then present how assignments are chosen.

3.2 Calculating the Score

To calculate the score, the scheduler maintains a model of the network and its available resources, as well as costs for user operations. The network is described as a graph (V, E) where V is the set of hosts h in the network, and E is the set of links l connecting the hosts; a single link can connect more than two hosts (i.e. they can be broadcast). We abuse notation and sometimes use V and E to refer to the cardinality of V and E (rather than $|V|$ and $|E|$).

The model assigns a *capacity* to each host and link, where **capacity**(h) is in terms of instructions per second, and **capacity**(l) is in terms of bytes per second. In addition, network links are assigned a *latency* **latency**(l) (in seconds). Each operation o has associated cost functions **cost**(o) and **fsize**(o), which are the approximate number of instructions the operation takes, and the average size of any output frames, respectively. In our implementation, cost functions are parameterized by frame inputs, architecture type, etc.

In addition, the scheduler uses information supplied in the user CMN during its calculation. For example, we use **interval**(o) and **maxdelay**(o) to denote the minimum interval and maximum acceptable delay, as indicated by the user CMN, where **interval**(o) is the minimum time between subsequent invocations of operation o, and **maxdelay**(o) is maximum acceptable delay between the time a frame enters the CMN and operation o processes it. These and other predicates, as well as some notational conventions, are summarized in Table 1.

Using this information, the scheduler can approximate how long it takes for any operation to compute on any host, and how long it takes to propagate the output over any network. In the future we intend to use more detailed monitoring so that the GS can update its model over time.

The total score is the minimum of three separate scores: the *host score*, the *network score*, and the *operation score*. These scores measure, respectively, the leftover ratio of computational capacity, the leftover network capacity, and the leftover ratio of acceptable delay; we make these notions precise below. The larger the scores, the less loaded the system is, and thus the more preferable the assignment.

Each score is calculated in the same way, using the following methodology. We first calculate a *local score* $ls(x)$ for each of n *entities* x. For example, for the host score, the entities are the hosts $h \in V$, and the local scores are the computational loads $L(h)$ on each host. We then determine the *scaled leftover capacity* $slc(x)$ by subtracting the local score from, and dividing it by, the *local capacity* $c(x)$:

$$slc(x) = \frac{c(x) - ls(x)}{c(x)}$$

Metavariables

o, p	$\in$ *Operations*	(i.e. some operation)
O, U, S, R	$\in \mathcal{P}(\mathit{Operations})$	(i.e. a set of operations)
h	$\in V$	(i.e. a host)
l	$\in E$	(i.e. a link)

Predicates on the network (V, E) *and its operation model*

hosts(l)	The set of hosts connected via link l
latency(l)	The time required to send one bit across link l
capacity(l)	The bandwidth available on link l
capacity(h)	The computational cycles per second available on host h
cost(o)	The computational cost of operation o
fsize(o)	The average size of frames output from operation o

Predicates derived from the global CMN

inputs(o)	The set of operations inputting into operation o
interval(o)	The minimum interval of operation o
maxdelay(o)	The maximum acceptable delay for operation o

Predicates on the schedule being considered

link(o, p)	Assuming o and p are connected send and receive operations, this is the link between them
host(o)	The host on which operation o is scheduled
ops(h)	The set of operations scheduled on host h
rops(h)	The set of receive operations scheduled on host h ($\subseteq$ **ops**(h))

Table 1: Definitions for Global Scheduling Algorithm

When the load exceeds the capacity, $slc(x)$ will be negative; otherwise it will be between 0 and 1 (higher is better). Finally, we aggregate the scaled leftover capacities into a single value. To favor assignments that avoid overloading a single entity, we use the *harmonic mean*, which strongly weights lower values, when all $slc(x)$ are non-negative. If any $slc(x)$ is ≤ 0, we use the smallest individual value:

$$m = \begin{cases} \dfrac{1}{\sum_{\text{all } x} \dfrac{1}{n \times slc(x)}} & slc(x) > 0 \text{ for all } x \\ \mathbf{min}_{\text{all } x}(slc(x)) & \text{otherwise} \end{cases}$$

Using this technique, we calculate the three scores as follows:

- *Host Score.* For the host score, the entities x are the hosts h, the local score is the computational load on the host $L(h)$, and the capacity is the host's total computational capacity $\mathbf{capacity}(h)$. The computational load $L(h)$ is, for every operation o scheduled on host h, the cost of the operation o divided by its specified minimum interval:

$$L(h) = \sum_{o \in \mathbf{ops}(h)} \frac{\mathbf{cost}(o)}{\mathbf{interval}(o)}$$

- *Network Score.* For the network score, the entities are the network links l, the local score is the required bandwidth on the link $B(l)$, and the capacity is the link's total capacity $\mathbf{capacity}(l)$. The required bandwidth $B(l)$ is calculated by determining, for all hosts h connected by link l, the receive operations whose data arrives over link l, and summing their relevant output frame sizes (which match their input frame sizes) divided by their intervals:

$$B(l) = \sum_{h \in \mathbf{hosts}(l)} \sum_{o \in \mathbf{rops}(h)} \frac{\mathbf{fsize}(o)}{\mathbf{interval}(o)}$$

- *Operation Score.* Finally, for the operation score, the entities are the operations o, the local score is the operational delay $D(o)$, and the capacity is the user's maximum acceptable delay $\mathbf{maxdelay}(o)$. The operational delay $D(o)$ is intuitively the maximum time the operation must wait from the time the CMN first receives a frame to the time the operation in question can operate on it. To calculate this, we first determine the maximum delay on each host $md(h)$ as the sum of the costs of all operations that run on that host:

$$md(h) = \sum_{o \in \mathbf{ops}(h)} \mathbf{cost}(o)$$

The idea is that $md(h)$ is the maximum time an operation could be delayed due to other operations running on the same host; we assume no operation o will run twice once an operation p becomes runnable. We then determine the maximum delay $nd(o,p)$ on each pair of connected operations in a similar manner:

$$nd(o,p) = \begin{cases} \mathbf{latency}(\mathbf{link}(o,p)) & \mathbf{link}(o,p) \text{ defined} \\ 0 & \text{otherwise} \end{cases}$$

That is, $nd(o,p)$ is the delay of the network connecting operations p and o, which is $\mathbf{latency}(\mathbf{link}(o,p))$ when $\mathbf{link}(o,p)$ is defined (i.e. when o and p are paired send and receive operations), and 0 otherwise. Finally, we calculate the delay $D(o)$ as the sum of maximum delay on the local host and all of the delays of o's upstream neighbors and the intervening network links (if any):

$$D(o) = md(\mathbf{host}(o)) + \sum_{p \in \mathbf{inputs}(o)} D(p) + nd(o, p)$$

When operations take inputs from multiple operations we also add the minimum interval, as the operation may have to wait that long to be scheduled. When aggregating $D(o)$, we only consider operations o for which the user has specified a maximum acceptable delay $\mathbf{maxdelay}(o)$.

Note that there is a tension between latency and bandwidth: a path with minimal latency may not be the most bandwidth-plentiful path. Similarly, there are other tensions, say between minimizing CPU over and minimizing network usage. Our approach to specification allows these issues to be resolved, as the scheduler tries to optimize the worst utility achieved by all its users.

3.2.1 Cost of Scoring

The cost of scoring an assignment is a function of the network and the user specification assignment. The user assignment consists of the mapping of O operations to network nodes, where O is broken down into S send operations, R receive operations, and U (other) user operations. Furthermore, we define

$$\mathbf{deg_{in}}(O) = \sum_{o \in O} |\mathbf{inputs}(o)|$$

This is the total number of inputs for all operations in O, which corresponds to the number of edges in the CMN graph defined by the O operations. If each operation has only a single input, then $\mathbf{deg_{in}}(O) = |O|$.

Given these definitions, the time to compute the node score, network score, and operation score are as follows:

host score	$\mathcal{O}(V + O)$
network score	$\mathcal{O}(E + R)$
operation score	$\mathcal{O}(O + \mathbf{deg_{in}}(O))$

For the host score, the V component comes from taking the mean over all hosts h in the network, the O component is due to calculating the score $L(h)$. For the network score, the E component comes from taking the mean over all network links l, and the R component is due to calculating the required bandwidth $B(l)$ on each link by looking at all receive operations. Finally, for the operation score, the O component comes from taking the mean over all operations, as well as calculating the maximum delay $md(h)$ for each host h. The $\mathbf{deg_{in}}(O)$ component comes from summing the upstream delays for each operation.

This leads to a total cost of $\mathcal{O}(V + E + O + \mathbf{deg_{in}}(O))$. When $\mathbf{deg_{in}}(O)$ is roughly equal to O (e.g. for multicast-style applications), we have $\mathcal{O}(V + E + O)$.

3.3 Choosing an Assignment

To pick an assignment that maximizes user utility, we must consider user specifications at various levels of utility, pick possible assignments, score them, and choose the best one. Even when ignoring multiple utility levels, and the need for placing intervening send and receive operations, it is easy to see that for a particular network and CMN, there are U^V possible assignments. This means a brute force enumeration of assignments is infeasible. Therefore, a reasonable algorithm requires a way to prune the assignment space while still arriving at a good schedule.

We present an algorithm here that is roughly $\mathcal{O}(V^2U^2(U^2V+E))$.[5] The algorithm works well in the scenarios we have considered, but could certainly be improved; e.g. ideas from related work could be applied [12, 30]. The basic idea that recurs throughout the algorithm is that rather than consider an entire search space (such as assignments of user operations to nodes, or the combinations of users' CMNs at different utility levels), we break a space into more coarse-grained pieces, and make locally beneficial decisions.

We create an assignment of operations to hosts in two nested phases. In the outermost phase, the scheduler does a binary search on the utility space, trying to find the best utility assignment. When evaluating utility u, the scheduler picks for each user the CMN that has utility u or the closest one below u. It then merges the CMNs and executes the inner phase, described next, to find best-scoring assignment. If the score of the assignment is nonnegative, the scheduler tries a higher utility value; otherwise a lower one. This process continues until the remaining utility space becomes smaller than some pre-chosen ϵ. The utility chosen is the lower bound of this space. If 0, the algorithm could not find an assignment that works.

As an optimization, after we have arrived at a lower bound, we try to improve the utility of some (but not all) users, by increasing the utility of each individual user one at a time. When at some point this fails because not enough resources are available, the algorithm finishes. The order in which users are considered in this phase depends on their priority.

The inner phase tries to find a reasonable assignment for a given global CMN (that is, the CMN resulting from merging the user CMNs of various utilities). The first thing it does is to discover the connected components of the global CMN. These are essentially the multicast trees of user operations that originate from a particular video source. Then, for each of the c connected components, the inner phase does the following:

1. It calculates the all-pairs "shortest" paths of the network, using maximum *bottleneck bandwidth* (also referred to as the *path bandwidth*)[6] as the metric of optimality; this takes time $\mathcal{O}(V^3)$. We could also use the maximum-weight spanning tree, rather than all-pairs shortest paths, to speed up the computation to $\mathcal{O}(E+V\log V)$ when using Prim's algorithm with Fibonacci heaps.

2. It assigns all operations to default locations (either their assigned location, or on one particular node). Next, it inserts send and receive operations in the CMN to connect operations adjacent in the CMN but assigned to different hosts. Using the most bandwidth-plentiful links, determined by the shortest path computation just completed, the GS creates a tree of paths originating from each operation to its immediate downstream neighbors in the CMN. At each intermediate (physical) host in this tree, the scheduler inserts receive and send operations to forward the data. It can then calculate the score.

[5] Recall that $R \leq O$, so it gets folded into the O portion of the equation.

[6] The maximum bottleneck bandwidth of a path is the bandwidth of the link along the path with the smallest available bandwidth.

	Shortest paths for all CCs	$\mathcal{O}(cV^3)$	
		$= \mathcal{O}(UV^3)$	since $c \leq \lvert U \rvert$
$+$	Each possible assignment	UV	
$\times$	Inserting send/receive ops	$\mathcal{O}(\mathbf{deg_{in}}(U) \cdot V)$	
$\times$	Calculating the score	$\mathcal{O}(V + E + U + S + R + \mathbf{deg_{in}}(O))$	since $\mathbf{deg_{in}}(S \cup R) = S \cup R$
		$= \mathcal{O}(V + E + U + S + R + \mathbf{deg_{in}}(U))$	since $(\lvert S\rvert + \lvert R\rvert) \leq \lvert U\rvert \cdot \lvert V\rvert$
		$= \mathcal{O}(V + E + U + UV + \mathbf{deg_{in}}(U))$	
		$= \mathcal{O}(UV + E + \mathbf{deg_{in}}(U))$	
$+$	Updating graph for all CCs	$\mathcal{O}(c(V + E))$	
		$= \mathcal{O}(U(V + E))$	since $c \leq \lvert U \rvert$
$=$	$\mathcal{O}(U(V^3 + V + E)) + \mathcal{O}(UV \cdot \mathbf{deg_{in}}(U) \cdot V \cdot (UV + E + \mathbf{deg_{in}}(U)))$		
$=$	$\mathcal{O}(UV^3 + UV + UE + V^2U \cdot \mathbf{deg_{in}}(U) \cdot (UV + E + \mathbf{deg_{in}}(U)))$		
$=$	$\mathcal{O}(V^2U \cdot \mathbf{deg_{in}}(U) \cdot (UV + E + \mathbf{deg_{in}}(U)))$		
$=$	$\mathcal{O}(V^2U^2 \cdot (UV + E + U))$		assuming $\mathbf{deg_{in}}(U) = U$
$=$	$\mathcal{O}(V^2U^2 \cdot (UV + E))$		

Table 2: Breakdown of Global Scheduling Running Time, Inner phase

The cost of inserting the send and receive operations is roughly $\mathcal{O}(\mathbf{deg_{in}}(U) \cdot V)$. This cost arises from the fact that for each user operation input o, if its predecessor is located on a different node, we determine the shortest path between the two (using information already calculated), and merge this path with the tree already rooted at o. The cost of merging the path is essentially the length of the path itself when using a predecessor matrix implementation. No path will ever be greater than the number of nodes in the network, so this length is bounded by V.

3. At this point, it tries to greedily improve the score by relocating each (movable) user operation to each possible host, remembering for each operation the location that improves the score most. Send and receive operations must be inserted for every change. This process continues until no more operations can be moved. If no operations are fixed, $U \cdot V$ assignments of operations to nodes will be considered.

4. Finally, the connected component under consideration is fixed at its best scheduling, and the loads on the links in the GS network model are updated; this takes time $\mathcal{O}(V + E)$. The scheduler then moves on to the next connected component.

Note that the order that each of the c connected components will affect the resulting schedule, favoring the components considered first. This order should be consistent, once chosen, to avoid frequent reconfigurations. Ordering could be determined randomly, by per-user priority, and other metrics.

3.3.1 Cost of choosing an assignment

The running time of the inner phase of the algorithm is $\mathcal{O}(V^2U \cdot \mathbf{deg_{in}}(U) \cdot (UV + E +$

$\mathbf{deg_{in}}(U)))$, as broken down in Table 2. In the case of a multicast-style application, $\mathbf{deg_{in}}(U) = U$, yielding a running time of $\mathcal{O}(V^2U^2 \cdot (UV + E))$. This phase will be run for each combination of user utilities considered by the outer phase.

While performing binary search, the outer phase will consider $\log \frac{1}{\epsilon}$ possible schedules. During the optimization phase it will consider $n \cdot U$ additional schedules, where U is the total number of users, and n is the maximum number of distinct utilities in each user specification. For each schedule, the U user operations at the utilities being considered will have to be combined into the global CMN. This yields a total running time of $\mathcal{O}(U + V^2U \cdot \mathbf{deg_{in}}(U) \cdot (UV + E + \mathbf{deg_{in}}(U))(\log \frac{1}{\epsilon} + Un))$.

We are most interested in how the algorithm scales as we increase the size of the network (V, E) and the number of users. With this in mind, we can simplify the characterization of the running time by relating other parameters to these variables. In particular, we can assume that U, which is the sum total of all user operations for a particular combination of user utilities, is equal to $u \cdot x$ for some constant x, where u is the number of individual users. This is basically assuming that all user CMNs have fewer than x nodes. Moreover, we can hold n and $\log \frac{1}{\epsilon}$ constant. This results in a simplification of the running time to $\mathcal{O}(V^2U^2(UV\mathbf{deg_{in}}(U) + E + \mathbf{deg_{in}}(U)^2))$, broken down in Table 3. When holding $\mathbf{deg_{in}}(U) = U$ for multicast-style applications, we arrive at $\mathcal{O}(V^2U^2(U^2V + E))$. This basically illustrates that the dominating cost of the algorithm is the number of users being considered, and secondarily the number of nodes in the network. Being polynomial in V, E, and U makes the algorithm scale better than the exponential, brute-force approach.

3.4 Discussion

Here we discuss some aspects of the global scheduling algorithm.

Utility and Resource Usage A key tenet of the algorithm is that for a given user, a lower utility CMN will require fewer resources to schedule. Furthermore, as described in Section 2.1, the absolute magnitude of user utility values needs to be set based on a user's priority and the total resources consumed. At the moment, we assume that users only have access to adaptation templates that can, in aggregate, meet these requirements.

In future work, we plan to develop a generic notion of resource usage that combines the CPU and bandwidth requirements[7] of a CMN as if it were scheduled on a *virtual topology*, which should in some way approximate the actual one.

Relating utilities to resource usage depends on a well-defined generic notion of resource usage, which depends on the available resources. For example, in a CPU-plentiful environment we would expect more weight on the bandwidth. If, for the next lower user utility, the bandwidth requirement drops by 100 KB/s while required CPU time goes up by 1 ms, then the overall result would be a significant decrease in resource use. On the other hand, for CPU-poor environments, this situation may actually signal a resource increase. Our algorithm can weight single resource scores (*i.e.*, host score, network score, and/or operation score) by taking each to a par-

[7]Other resource requirements are likely to be useful, particularly memory requirements. To keep things simple, we expect to combine CPU and memory use as a single metric. On the other hand, our current algorithm could be extended to include memory counters without much trouble.

$$
\begin{array}{rll}
 & \mathcal{O}(V^2U\mathbf{deg_{in}}(U)(UV+E+\mathbf{deg_{in}}(U))(\log\frac{1}{\epsilon}+Un)) & \\
= & \mathcal{O}((V^3U^2\mathbf{deg_{in}}(U)+V^2UE+V^2U\mathbf{deg_{in}}(U)^2)(\log\frac{1}{\epsilon}+Un)) & \\
= & \mathcal{O}((V^3U^2\mathbf{deg_{in}}(U)+V^2UE+V^2U\mathbf{deg_{in}}(U)^2)(Un)) & \log\frac{1}{\epsilon}\text{ constant} \\
= & \mathcal{O}(V^3U^3\mathbf{deg_{in}}(U)+V^2U^2E+V^2U^2\mathbf{deg_{in}}(U)^2) & n\text{ constant} \\
= & \mathcal{O}(V^2U^2(UV\mathbf{deg_{in}}(U)+E+\mathbf{deg_{in}}(U)^2)) & \\
\hline
= & \mathcal{O}(V^2U^2(U^2V+E+U^2)) & \text{assuming }\mathbf{deg_{in}}(U)=U \\
= & \mathcal{O}(V^2U^2(U^2V+E)) &
\end{array}
$$

Table 3: Simplifying the Running Time Characterization of the Algorithm

ticular power. (For a negative score s and power x, use $-(-s)^x$.) After doing so, each score will still be negative for exceeding a capacity, 0 for reaching it exactly, and 1 for being completely unloaded.

Implementation As part of our prototype, we have implemented the GS in C, consisting of about 10,000 lines of code. We measured the performance of the scheduler on-line for the experiments presented in Section 5.3, in which we had an eight node network with five users. We measured running times of between 1 ms and 90 ms, with the longer running times for the cases when the network was more loaded, and thus more possibilities were considered. Much of the running time is due to 'constant factors' in our implementation that we have yet to tune. For example, we use an excessive amount of allocation when inserting send and receive operations for each configuration

4 Local Scheduling

The LS is responsible for implementing the CMN provided by its parent GS, reporting back resource consumption information (monitoring), and safely reconfiguring to use a new CMN when directed to do so; we describe all of these tasks in this section.

```
struct Operation {
  string_t id;
  double interval;
  fn_t<streambuff_t,int> ? inports;
  fn_t<streambuff_t,int> ? outports;
  fn_t<fn_t<streambuff_t,int> ?,int> schedule_f;
};
```

Figure 4: A user operation in Cyclone

In our prototype, the LSs are written in the type-safe systems language Cyclone [22], which is based on C, comprising roughly 13,000 lines of code. Cyclone is simply C at its core, but with both restrictions to ensure type-safety and enhancements for greater flexibility (*e.g.*, exceptions, tagged unions, garbage collection, a variety of safe pointer types, etc.).

4.1 Implementing a CMN

When a compute node receives a reconfiguration message, it translates the CMN into a graph of data structures implementing its operations. The Cyclone implementation of an operation is shown in Figure 4. Each operation consists of a name and a compute interval, which map directly from the user specification. It also contains 0 or more input ports and 0 or more output ports, where the output ports refer to the inport ports of the downstream operations. We use an upcall model where each input port is a closure that expects a frame, along with some operation-specific state, encoded as the environment of the closure.[8] An `inports` closure is invoked by the upstream operation when it invokes its corresponding `outports` closure. In the figure, the syntax `fn_t<streambuff_t,int>` indicates the type of a closure specialized to inputting `streambuff_t` structures (our definition of frames), and returning integers. The `?` syntax indicates a dynamically-sized (*i.e.*, `malloc`'ed) array.

In addition to being invoked by receiving input data, an operation can be scheduled to run at the requested interval, by calling its `schedule_f` function. This function is also a closure, having some hidden state relating to the operation, and additionally the list of the operation's output ports, so that it can send any generated data downstream. For example, we implement `monitor` operations that wake up at specific intervals and generate monitoring messages; these messages are sent out by invoking the provided outports.

In essence, the local scheduling algorithm is as follows: after creating components that implement the operations, it sorts them topologically based on their data-flow. Next, it uses deadlines to ensure that operations are run as soon as possible after the prescribed interval elapses (if one is given), following the topological ordering. When operations are data-driven the LS simply runs the operations when frames arrive.

[8]Closures are not supported directly in Cyclone, but via abstractions for *existential types* which can be used to encode them [27].

4.2 Transport

To allow legacy applications to use MediaNet seamlessly, MediaNet's transport protocol, implemented by its inserted send and receive operations, needs to meet the API expected by the application. For example, if the application uses TCP to receive its data, then MediaNet must not only connect to that application via TCP on its last hop, but also needs to ensure that data is delivered to the receiver reliably, in order, and without duplication by that point. A UDP-based application would impose fewer requirements. MediaNet should support a variety of transport protocols between send and receive operations to maximize the performance of the system while still meeting the minimal requirements of the application. For expedience, our prototype implementation uses TCP exclusively; we plan to support other transport protocols, such as UDP, RTP [38] over UDP, and possibly others.

One benefit of using TCP within MediaNet is that it readily communicates bandwidth limitations, mitigating the need for external available bandwidth detection facilities. In particular, when the TCP send buffer fills up, the application receives an `EWOULDBLOCK` error and therefore queues its frames until more bandwidth is available. Once the application queue is filled, the consequent action depends on the application semantics. For streams that can tolerate dropped frames, like video streams, MediaNet will start dropping frames based on priority. User-supplied operations are used to set the priority (see Figure 2), supporting local adaptation. If a stream cannot tolerate lost data, then

MediaNet will exert backpressure to the sending application, effectively to throttle its rate (until a reconfiguration can take place). Choosing a reasonable queue size is important for reconfigurations, and we mention it further below.

4.3 Monitoring

For adaptive reconfiguration to be profitable, the GS must be reasonably well-informed of changes to the network, particularly those to its topology and to the loads on nodes and links. We have been focusing on bandwidth limitations in our experiments, and therefore on available bandwidth reporting; we have yet to implement a CPU monitor.

Available bandwidth detection is an ongoing area of research with no clear, general solutions as yet [13]. In particular, various techniques trade off accuracy, overhead, and measurement time. For example, packet-pair-based estimates [13, 4, 17] can quickly predict available bandwidth with extremely low overhead (just a few packets), but only reliably so for single hop links [20, 4], including wireless links [4]. On the other hand, Jain and Dovrolis' approach using one-way delays [20] works for multi-hop paths with reasonably low overhead (on the order of a few hundred packets), but the estimation time is typically between 10–30s and is often within a couple Mbps of the "actual" value. These limits to accuracy and speed constrain the timescales and magnitude of the changes made by the global scheduling service.

In our implementation, each LS notes how much data is sent and dropped (at the application level) for a given link, and sends this information periodically to the GS. These reports provide a low-overhead, highly relevant way of assessing the available bandwidth. It is low-overhead because the information is piggybacked on the actual stream being sent, and it is most relevant because it directly reports the value of interest to the global scheduler: how much data can a node send across a particular link using the appropriate transport protocol?

We observe that each link report will indicate that either the link can support the bandwidth imposed on it, or it cannot. That is, either all the data intended to be sent was sent, or else frames were dropped or backpressure was applied. For the latter case, the GS knows that the link is at peak capacity, so it sets its estimate to the reported sent bandwidth (for broadcast links, reports must be aggregated). In the former case, it knows the capacity is *at least* the reported amount.

Unfortunately, this approach only provides an upper bound on the available bandwidth when a link is overloaded; this is the main drawback of the technique. To compensate, if the GS does not receive a link peak capacity report for some time, it assumes that additional bandwidth might be available and so begins "creeping" its bandwidth estimate for that link at regular intervals in the spirit of TCP's additive increase. The net effect is that eventually a reconfiguration will take place that uses the assumed-available bandwidth; if the estimation is incorrect then the new configuration will fail and another will take place to compensate. We currently increase the estimate by a constant $w = 3\%$ each second beginning $t = 5$ seconds after a peak capacity report. The choice of values for w and t essentially determines how rapidly the GS tries to find unused bandwidth.

There is a tension between monitoring and accuracy: the more frequently that monitoring information is sent, the more accurate the GS network model will be, but the more overhead there will be on network links. To reduce traffic but maintain accuracy, each LS sends non-peak reports only when the reported bandwidth increases by $\Delta = 10\%$. Peak reports are sent every $r = 1$ seconds. We have found that this approach (rather than sending *all* reports every $r = 1$ seconds) reduces monitoring traffic by roughly 75% in our experiments.

As future work, we plan to incorporate other forms of feedback and estimation into our link estimates to improve the accuracy of the GS's network view. For example, Jain and Dovrolis' technique of finding an *increasing trend* in the latencies of sent packets could be incorporated into our measurements to determine an upper bound *before* a link becomes overloaded. In general, we wish to associate "confidence" measures with link bandwidth estimates, so that mostly estimated links are not weighted as highly as those with recent measurements during scheduling. We also imagine that "link profiles" could be used to estimate unmeasured links based on past usage. Finally, we could consider testing unmeasured links after a new schedule is determined but before it is used to configure the network.

4.4 Reconfiguration

When a global reconfiguration is initiated, it should take effect quickly and safely, without negatively impacting perceived user quality. We do this by defining a protocol that allows old and new configurations to run in parallel until the old configuration can be removed. We use a number of mechanisms to ensure the old configuration is removed as quickly as possible, while preserving the application's expected stream semantics.

The protocol works as follows. Whenever the GS calculates a new schedule, it sends a new CMN to each LS. The LS schedules this CMN immediately upon receipt, in parallel with its existing configuration. So that the two configurations do not interfere, the GS assigns different TCP port numbers to its inserted send and receive operations. As such, these operations will establish connections, but connections to the video source and receiver outside of MediaNet (which are still using the same ports) will be delayed until they are closed by the old configuration. Next, all video source applications are notified that a reconfiguration has taken place (we do this using out-of-band TCP data from the downstream MediaNet node). They each close their connections to MediaNet and reconnect, this time connecting to the new configuration. In the meantime, the old configuration will continue to forward any data it has toward the destination; when a LS's old queues are flushed the old configuration is removed. When the last bit of old data is sent to the video receiver, the new configuration will be able to connect to it and forward its data.

Using this protocol, we minimize the time during which the video source and receiver are disconnected from MediaNet; for our experiments this time averages 1 ms, which is far less than a typical video inter-frame interval of 33 ms. Even so, to reduce the total switching time we must reduce the time that the old configuration stays connected to the receiving application; during this time, frames from the new configuration queue while waiting to connect to the receiver. In the case that frames can be dropped, we reduce the old configuration's lifetime by quickly clearing its application queues via priority-based frame dropping, as described next. Otherwise, the queues must clear naturally; this suggests that when frames cannot be dropped, application queue lengths should be relatively short, so as to permit quicker reconfigurations.

We initiate frame dropping in two ways. First, we tie together the queue lengths of connections using the same link, so that higher priority new frames can force the dropping of lower-priority old ones sharing the same link. Second, for those cases in which the old and new paths are not shared, we set a drop timer (currently going off every 0.5 s) that proactively drops increasingly higher priority frames from the old queues. Using these methods, our average reconfiguration time in the larger experiment in Section 5.3 is 0.3 s, with a maximum time of about 1.1 s; these times are easily within the buffering window of most video players.

For stability, global reconfigurations are initiated at most once per *reconfiguration window* w, currently with $w = 5$ seconds. The larger this window, the less adaptive, but the more stable the system. We are currently experimenting with different kinds of windows for limiting reconfigurations, based on the quality of the network model, rather than on a fixed timeout.

5 Experiments

In this section, we present experiments that measure MediaNet while delivering an MPEG video stream under various topologies and load conditions. We show that MediaNet consistently delivers good performance and efficient network utilization by effectively utilizing redundant paths, by exploiting commonality in user specifications, and by carefully locating CMN operations.

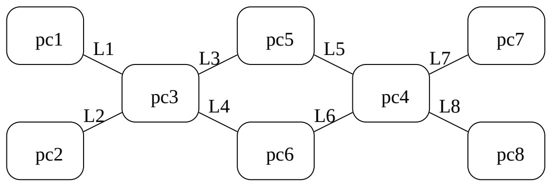


Figure 5: Experimental Topology on Emulab.

5.1 Configuration

The experiments were performed on *Emulab* [14, 41], configured to use the topology shown in Figure 5. Each node is a 850MHz Intel Pentium III Processor running RedHat Linux 7.1, having 512MB RAM and 5 Intel EtherExpress Pro 10/100Mbps Ethernet cards. Emulab supports "dynamic events scheduling" to inject traffic shaping events on-the-fly, implemented by Dummynet nodes [36]; we use this to increase and decrease the available bandwidth on various links during our experiments. In all experiments we ran a LS on every node and the GS on `pc3`.

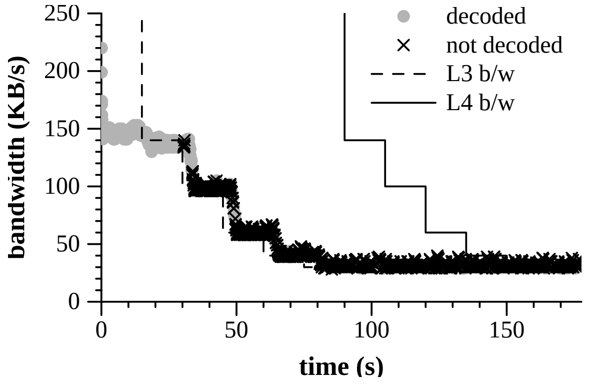
For the source video, we loop a MPEG video stream with the following frame distribution:

Frame Type	Size (B)	Frequency (Hz)
I	13500	2
P	7625	8
B	2850	20

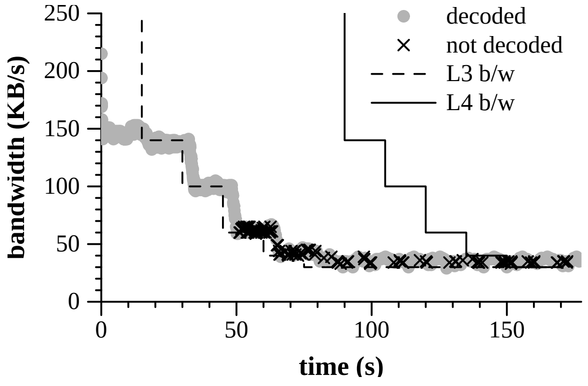
This video requires about 145 KB/s to send at its full rate, about 88 KB/s to send only I and P frames, and about 27 KB/s to send only I frames.

5.2 Exploiting Global Adaptation

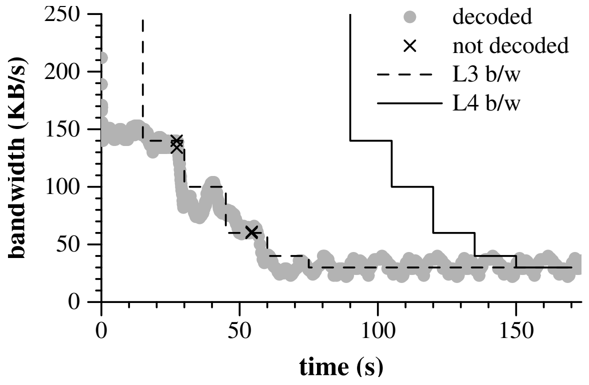
To demonstrate the benefit of local adaptation under network load, and then the added benefit of global adaptation, we compare four different configurations:



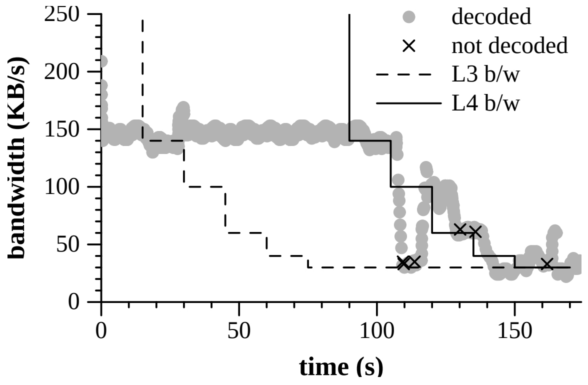
(a) no adaptation



(b) local priority-based frame dropping



(c) local proactive frame dropping



(d) MediaNet

Figure 6: User-perceived performance under diminishing bandwidth for various adaptivity schemes.

- The "no adaptation" configuration consists of streaming the data at the desired play rate, oblivious to network conditions. We implement this with the MediaNet LSs only.

- The "priority-based frame dropping" configuration consists of tagging P, B, and I frames with successively higher priority, and during overload the lowest priority frames are dropped. This approximates some past approaches on intelligent frame dropping [5, 23].

- The "proactive frame dropping" configuration also consists of intelligently dropping video frames during overload. In this case, the LS observes when frames start getting dropped for a particular link, and then adapts by proactively dropping *all* B frames, and then later *all* P frames. When dropping frames, the LS will occasionally attempt to improve the configuration; *i.e.*, if it is dropping P and B frames, it will try just dropping B frames. This configuration approximates past approaches to intelligent, in-network frame dropping, as well as end-to-end layered approaches [26] (where each frame type essentially defines a layer). In particular, the path of the data never changes, just what data is sent along that path. For this experiment, we implement this approach by using the GS but preventing it from choosing alternate paths.

- Finally, the "global adaptation" uses MediaNet's GS with the user specification depicted in Figure 2.

For each configuration, we ran an experiment that uses the diamond portion (`pc3` to `pc6`) of our topology (Figure 5), with a single video sender on `pc3` and a receiver on `pc4` The experiment measures the video player's performance, in terms of the received bandwidth and the decodable frames, as we lower both link `L3`'s and `L4`'s available bandwidth over time.

Each of the graphs shown in this section has the same format. Each light gray circle in the figure is a correctly-decoded frame, while each black $\times$ is an incorrectly decoded one. The figure plots time versus bandwidth, so the x-location is the time the frame is received, and the y-location is the bandwidth seen by the player at that time (aggregated over the previous second). The available bandwidth, as set by Dummynet, is shown as dashed and/or solid lines. Dropped frames are not shown.

Figure 6(a) shows the no adaptation case. At the start, the route to the receiver is fixed along L3, and as the available bandwidth on the link drops the video quality degrades. The application cannot decode the majority of the received frames because temporally important frames (I and P frames) are being dropped. During playback, each undecodable frame manifests as a "glitch" noticeable by the user. In this case, the large and constant clumping of glitches is quite disruptive. In the players we have used, these result in a checkerboard pattern momentarily appearing and corrupting the playback; corrupted playback persists until a frame can be correctly decoded.

Figure 6(b) shows the priority-based dropping case. In this case, playback improves when dropping B frames, but remains poor under highly loaded conditions. Until roughly time 50, the player can decode all of its received frames, but after that a large fraction of frames cannot be decoded properly. This is because by this time we are only sending I or P frames, so any dropped P frame could prevent downstream P frames from being decoded.

In contrast, when using local adaptation along the same path, the performance improves significantly, as shown in Figure 6(c). The few glitches that occur are as a result of a sudden drop in available bandwidth, and due to attempts to obtain a better configuration when no resources are available. By dropping all B and/or P frames, we avoid dropping frames that could lead to temporal glitches.

While the proactive frame-dropping adaptation significantly improves playback along a congested path, it fails to use alternative paths that could further improve playback. In contrast, MediaNet's global scheduler reconfigures the network to utilize redundant paths.[9] Figure 6(d) shows how MediaNet's GS reroutes traffic through `pc6` when `L3` becomes congested at roughly time 30, utilizing the idle `L4`. Later, `L4`'s bandwidth is reduced as well, which causes MediaNet to start dropping frames until it reaches the same level as the local case.

A number of times in this experiment, the GS optimistically assumes that more bandwidth is available on unmaximized links and attempts to improve the total utility. At time 105 when link L4's bandwidth drops, it tries to reroute the flow through link L3. However, L3 has even lower available bandwidth, and so after the reconfiguration window expires (here set to 5 seconds), the GS returns the configuration to link L4, at utility 0.4 (dropping B frames). Similar failed attempts occur at times 120 and 155. Our user configuration mitigates the negative effects of such reconfigurations by intelligently dropping frames until the network is reconfigured. Ideally we could prevent these spurious configurations without becoming so conservative so as to degenerate to local adaptation only; possible approaches are discussed in Section 4.3.

We should emphasize that MediaNet's contribution is not simply multi-path routing or local adaptation, since each has been explored in prior contexts. Rather, MediaNet's global scheduling service encapsulates a more general way of performing adaptation on a network-wide basis, based on individually-specified adaptation preferences and metrics. In so doing, it in effect employs both local adaptations (*i.e.*, proactive frame dropping) and global adaptations (*i.e.*, path rerouting), among others, to meet the needs of users and the network.

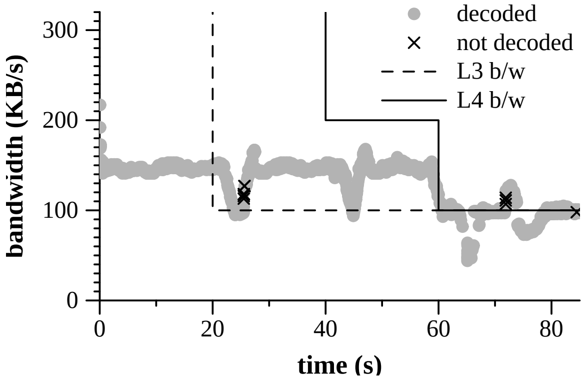
5.3 Multi-user Sharing

To examine how resources are shared among users, we configured MediaNet with two video sources and five clients. Video `v1` on `pc1` has three clients: users `user1` on `pc5`, `user3` on `pc7`, and `user5` on `pc8`; video `v2` on `pc2` has two clients: users `user2` on `pc4` and `user4` on `pc7`. Each user specification varies from the one shown in Figure 2 in the specification of the video source and user locations.

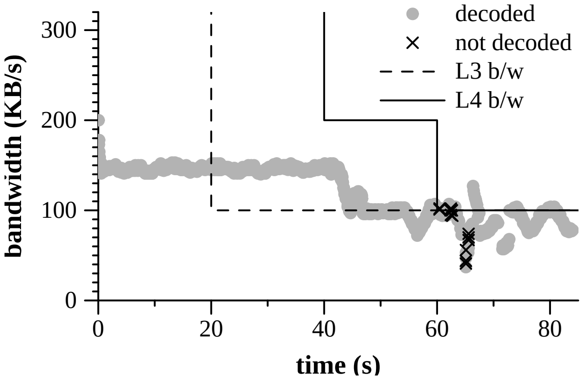
If all links are fully available, the GS assigns the operations as shown in Figure 8(a). The unlabeled operations are TCP sends and receives, and the `Pr` operations assign frame priorities for intelligent dropping. In combining the five user specifications, the GS has essentially created two multicast dissemination trees, and uses `L3` for the `v1` stream and `L4` for `v2`.

The performance as seen at the two sets of receivers is shown in Figure 7. At time 20, the bandwidth on link L3 is reduced to 100 KB/s,

[9] Redundant paths occur frequently in the wide area [37], and mobile hosts often have multiple networks available, *e.g.*, many laptops have cellular, 802.11b, and Ethernet.



(a) users 1, 3, 5



(b) users 2, 4

Figure 7: User-perceived performance for multiple user scenario.

and so the GS reconfigures the network to be as in Figure 8(b) where all flows go along link L4 so as to maintain utility 1.0 for all users. At time 40, the bandwidth on link L4 is dropped to 200 KB/s, making it impossible to carry both streams along that link. As such, the GS reconfigures to be as in Figure 8(c), in which `v2` is sent along link L3 with its B frames dropped (as indicated by the `dB` node on `pc2`), using the utility 0.4 CMN, while `v1` goes along link L4 at utility 1.0 for all users. Notice that the GS has scheduled the `dB` (dropping B frames) node *at the source* `pc2` rather than at the node connected to the congested link, for better network utilization. At time 60, L4's bandwidth also drops to 100 KB/s, which results in all flows now operating at utility 0.4, as shown in Figure 8(d). Here we can see that this is essentially the same as the unloaded configuration in Figure 8(a) but with `dB` nodes on both of the video source hosts.

During the run, the GS guesses that additional bandwidth might be available on various links, and so attempts to improve the configuration. This occurs at time 50 (to improve to Figure 8(a)), but fails and reverts back at time 55. A similar attempt is made at time 80 (to go up to Figure 8(c)).

6 Related Work

Although distributed multi-media research has been popular for decades, the idea of multimedia processing in the network was first inspired by the problems of digital video broadcasting in heterogeneous networks [40, 33]. The goal of providing adaptive QoS for streaming data is shared by a number of systems, including *Active Networks* applications [39, 34, 5], QoS middleware substrates [23, 24, 25], and application-layer in-network processors [43, 1, 2]. Other projects have targeted the dissemination to mobile, wireless workstations, such as Quasar [18] and Odyssey [31]. None of these systems focuses on sharing resources among many users with differing adaptation preferences, though adaptivity mechanisms and resource models are quite relevant.

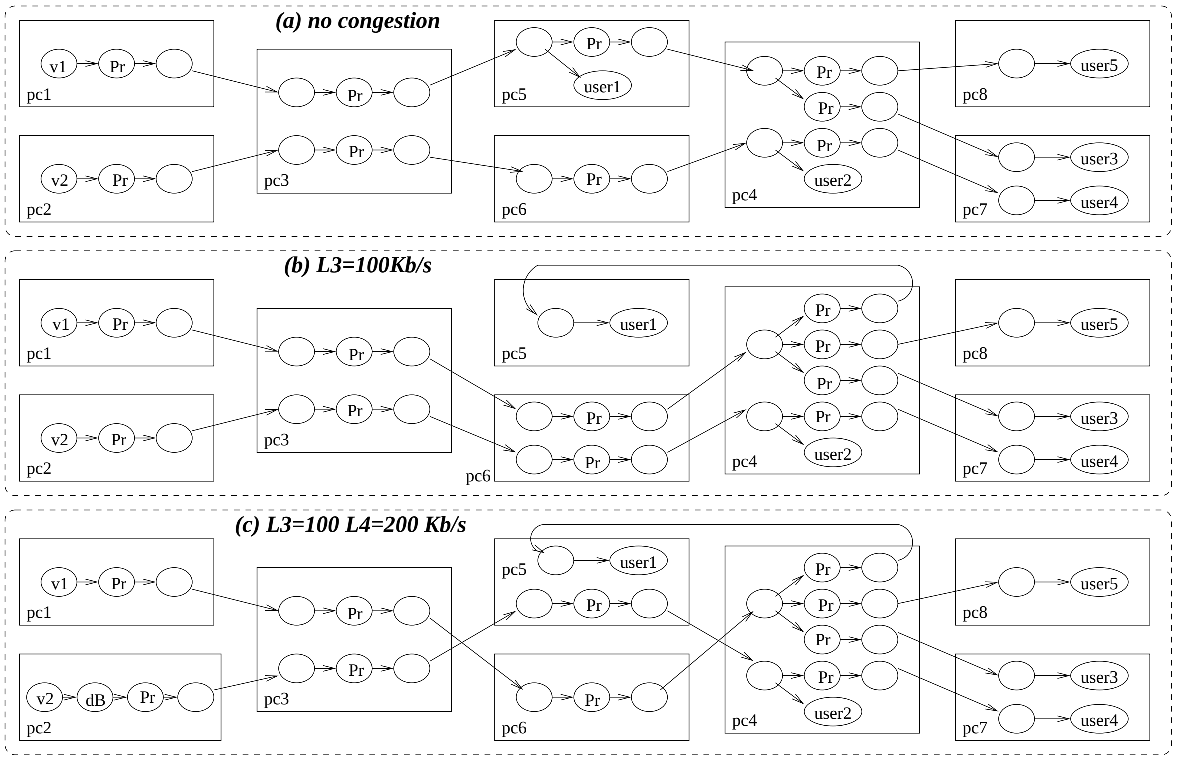


Figure 8: Scheduling under varying conditions for multiple users.

A few systems have considered efficient stream adaptations shared among many users. Layered multicast [26, 34, 35] shares resources efficiently among many users, and Degas [32] contains decentralized protocols for task distribution and load balancing of streaming data operations. Layered multicast layers are coarse-grained abstractions, however, and do not support more "computational adaptations" like transcoding. Degas similarly fails to account for user preferences in scheduling adaptations.

MediaNet shares some mechanisms with certain overlay networks (*e.g.*, RON [3]) that, in addition to constructing a flexible virtual network on top of physical networks, can provide improved network performance via alternative paths in conjunction with bandwidth probing and failure detection [3, 37, 10]. To date, these systems have not been concerned with QoS (*i.e.*, real-time constraints) of streaming data, or sharing of resources among many users.

An alternative approach to adaptive QoS is *reservation-based* QoS, in which resources like CPU and bandwidth are allocated for applications in advance [9, 7, 8]. The drawbacks of reservations are that underlying support is not widely available, and allocated resources can be underutilized, resulting in inefficiency. A number of systems looked at application-specific scheduling in reservation-capable environments, for example, the OMEGA end-system architecture [28, 29].

A number of systems share our goal of supporting user-specified, adaptive streaming data applications, including CANS [15], Conductor [42], Darwin [11], End-to-end Media Paths [30], Ninja [16], PATHS [6], and [12]. Central to all of these systems is the notion of paths of stream transformers that must be scheduled on the network, and the presence of a centralized *plan manager* to schedule paths across the network, similar to MediaNet's GS. However, these systems only use the plan manager at initialization or rarely, while MediaNet's GS runs continuously. Less attention has been paid to exploring fast, on-line scheduling algorithms that are nonetheless effective, which would be needed in a scalable on-line system. As such, these systems do not take advantage of path-based, user-specified adaptation. In addition, plan managers appear to consider scheduling only for a particular application or flow, as opposed to the combination of many or all existing applications or flows, and therefore miss opportunities to improve both per-user and network performance, for example, by aggregation and/or re-routing.

7 Conclusions

MediaNet is an architecture for user-specified, globally-adaptive QoS in distributed streaming data applications. It has two clear benefits. First, adaptations are user-specified, rather than a system-determined. Second, MediaNet's global and local scheduling approach in effect employs both global and local adaptations; our experiments demonstrate better user and system performance in three ways:

1. The GS aggregates users' continuous media networks, removing redundancy in a multicast-like fashion.

2. It utilizes redundant resources, such as alternative, unloaded routing paths.

3. It adapts proactively to prevent wasted resources, for example by dropping frames close to the source when there is downstream congestion.

While our work is a promising first step, many questions remain. Three important areas are *scalability*, *accuracy*, and *applications*. For the first, we are interested in examining hindrances to growth—such as monitoring message overhead, GS running times, and network instability—to understand possible tradeoffs. For example, by increasing the reconfiguration window we limit the effects of configuration-flapping, but could be stuck with an ill-advised configuration. Or, by reducing the frequency of monitoring messages we reduce monitoring overhead, but increase the possibility of a bad schedule. We are particularly interested in devising a hierarchical system, such as used in Darwin [11].

A central requirement to an on-line adaptive system is to have an accurate view of its resources. As mentioned earlier (Section 4.3), we are interested in employing additional monitoring techniques and better heuristics for weighing information.

Another possible enhancement would be to consider splitting (or 'striping') data across multiple paths between a single source and destination. Doing so would require that the split data be properly resequenced upon reaching the destination, unless the receiving operation could tolerate out-of-order arrival. While striping would allow greater efficiency, it could significantly increase the scheduling overhead.

Finally, we have just scratched the surface of MediaNet's possibilities by experimenting with only network-limited (*i.e.*, video plus frame dropping) applications. We believe that MediaNet's generality will be quite useful when considering CPU-limited cases; for example, when streaming data to embedded devices, and/or while performing computationally-intense transformations, such as digital facial recognition or motion analysis.

Acknowledgements

Thanks to Michael Marsh for helping us discover the usefulness of the harmonic mean. Thanks to Cyclone development team members Greg Morrisett and Dan Grossman for their rapid response to our Cyclone-related problems. Thanks also to Scott Nettles, Jonathan T. Moore, Bobby Bhattacharjee, Trevor Jim, and Jonathan Smith for helpful comments on earlier versions of this paper.

References

[1] Elan Amir, Steve McCanne, and Zhang Hui. An application level video gateway. In *Proceedings of the Third ACM International Multimedia Conference and Exhibition (MULTIMEDIA)*, pages 255–266, November 1995.

[2] Elan Amir, Steve McCanne, and Robert Katz. An Active Service framework and its application to real-time multimedia transcoding. In *Proceedings of the ACM SIGCOMM Conference*, pages 178–189, September 1998.

[3] David G. Andersen, Hari Balakrishnan, M. Frans Kaashoek, and Robert Morris. Resilient overlay networks. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*. ACM Press, October 2001.

[4] Benjamin Atkin and Kenneth P. Birman. Evaluation of an adaptive transport protocol. In *Proceedings of the IEEE INFOCOM Conference*, April 2003. To appear.

[5] Samrat Bhattacharjee, Ken Calvert, and Ellen Zegura. On Active Networking and congestion. Technical Report GIT-CC-96-02, College of Computing, Georgia Tech, 1996.

[6] John Markus Bjørndalen, Otto J. Anshus, Tore Larsen, L.A. Bongo, and B. Vinter. Scalable processing and communication performance in a multi-media related context. In *Proceedings of the EUROMICRO Conference*, September 2002.

[7] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An architecture for differentiated services. Technical Report RFC 2475, IETF, December 1998.

[8] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource reservation protocol (RSVP). Technical Report RFC 2205, IETF, September 1997.

[9] Robert Braden, David Clark, and Scott Shenker. Integrated services in the Internet architecture: an overview. Technical Report RFC 1633, IETF, June 1994.

[10] John Byers, Jeffrey Considine, Michael Mitzenmacher, and Stanislav Rost. Informed content delivery across adaptive overlay networks. In *Proceedings of the ACM SIGCOMM Conference*, 2002.

[11] Prashant Chandra, Allan Fisher, Corey Kosak, T. S. Eugene Ng, Peter Steenkiste, Eduardo Takahashi, and Hui Zhang. Darwin: Customizable resource management for value-added network services. In *Proceedings of the IEEE International Conference on Network Protocols (ICNP)*, pages 177–188, October 1998.

[12] Sumii Choi, Jonathan Turner, and Tillman Wolf. Configuring sessions in programmable networks. In *Proceedings of the IEEE INFOCOM Conference*, April 2001.

[13] James Curtis and Tony McGregor. Review of bandwidth estimation techniques. In *New Zealand Computer Science Research Students' Conference*, volume 8, April 2001.

[14] Emulab.net, 2001. `http://www.emulab.net`.

[15] Xiaodong Fu, Weisong Shi, Anatoly Akkerman, and Vijay Karamcheti. CANS: Composable, adaptive network services infrastructure. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)*, March 2001.

[16] Steven .D. Gribble, Matt Welsh, Rob Van Behren, Eric A. Brewer, David Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, R.H. Katz, Z.M. Mao, S. Ross, and B. Zhao. The Ninja architecture for robust Internet-scale systems and services. *Computer Networks*, 35(4):473–497, March 2001.

[17] Ningning Hu and Peter Steenkiste. Estimating available bandwidth using packet pair probing. Technical Report CMU-CS-02-166, School of Computer Science, Carnegie Mellon University, September 2002.

[18] Jon Inouye, Shanwei Cen, Calton Pu, and Jonathan Walpole. System support for mobile multimedia applications. In *Proceedings of the Workshop on Network and Operating System Support for Digital Audio and Video*, pages 143–154, May 1997.

[19] Chalermek Intanagonwiwat, Deborah Estrin, Ramesh Govindan, and John S. Heidemann. Impact of network density on data aggregation in wireless sensor networks. In *Proceedings of the Twenty-Second Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS)*, July 2002.

[20] Manesh Jain and Constantinos Dovrolis. End-to-end available bandwidth: Measurement methodology, dynamics, and relation with TCP throughput. In *Proceedings of the ACM SIGCOMM Conference*, August 2002.

[21] JBI - Joint Battlespace Infosphere. `http://www.rl.af.mil/programs/jbi/default.cfm`.

[22] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *Proceedings of the USENIX Annual Technical Conference*, June 2002.

[23] David A. Karr, Craig Rodrigues, Joseph P. Loyall, Richard E. Schantz, Yamuna Krishnamurthy, Irfan Pyarali, and Douglas C. Schmidt. Application of the QuO quality-of-service framework to a distributed video application. In *Proceedings of the International Symposium on Distributed Objects and Applications*, September 2001.

[24] Baochun Li and Klara Nahrstedt. Dynamic reconfiguration for complex multimedia applications. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, pages 165–170, June 1999.

[25] Baochun Li and Klara Nahrstedt. QualProbes: Middleware QoS profiling services for configuring adaptive applications. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware)*, pages 256–262, 2000.

[26] Steven McCanne, Van Jacobson, and Martin Vetterli. Receiver-driven layered multicast. In *Proceedings of the ACM SIGCOMM Conference*, Stanford, CA, August 1996.

[27] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *Twenty-Third ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 271–283, St. Petersburg, Florida, January 1996.

[28] Klara Nahrstedt and Jonathan M. Smith. Design, implementation and experiences of the OMEGA end-point architecture. Technical Report MS-CIS-95-22, Department of Computer and Information Science, the University of Pennsylvania, 1995.

[29] Klara Nahrstedt and Jonathan M. Smith. The QoS broker. *IEEE Multimedia*, 2(1):53–67, 1995.

[30] Akihiro Nakao, Larry Peterson, and Andy Bavier. Constructing end-to-end paths for playing media objects. In *Proceedings of the IEEE Conference on Open Architectures (OPENARCH)*, April 2001.

[31] B.D. Noble and M. Satyanarayanan. Experience with adaptive mobile applications in Odyssey. *Mobile Networks and Applications*, 4:245–254, 1999.

[32] Wei Tsang Ooi, Robbert van Renesse, and Brian Smith. Design and implementation of programmable media gateways. In *Proceedings of the Workshop on Network and Operating System Support for Digital Audio and Video*, June 2000.

[33] Joseph C. Pasquale, George C. Polyzos, Eric W. Anderson, and Vachaspathi P. Kompella. Filter propagation in dissemination trees: Trading off bandwidth and processing in continuous media networks. *Lecture Notes in Computer Science*, 846:259–269, 1994.

[34] Ranga S. Ramanujan and Kenneth J. Thurber. An active network-based design of a QoS adaptive video multicast service. In *Proceedings of the Workshop on Network and Operating System Support for Digital Audio and Video*, pages 29–40, July 1998.

[35] Reza Rejaie, Mark Handley, and Deborah Estrin. Quality adaptation for congestion controlled video playback over the Internet. In *Proceedings of the ACM SIGCOMM Conference*, pages 189–200, 1999.

[36] Luigi Rizzo. Dummynet: A simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1), January 1997.

[37] Stefan Savage, Andy Collins, Eric Hoffman, John Snell, and Thomas Anderson. The end-to-end effects of Internet path selection. In *Proceedings of the ACM SIGCOMM Conference*, pages 289–299, September 1999.

[38] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A transport protocol for real-time applications. Internet RFC 1889, 1996.

[39] David L. Tennenhouse and David J. Wetherall. Towards an Active Network architecture. *ACM Computer Communication Review*, 26(2):5–18, April 1996.

[40] Thierry Turletti and Jean-Chrysotome Bolot. Issues with multicast video distribution in heterogeneous packet networks. In *Proceedings of the Packet Video Workshop*, pages F3.1–3.4, September 1994.

[41] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Bard, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, December 2002.

[42] Mark Yarvis, Peter Reiher, and Gerald J. Popek. Conductor: A framework for distributed adaptation. In *Proceedings of the IEEE Workshop on the Hot Topics in Operating Systems (HOTOS)*, pages 44–49, March 1999.

[43] Nicholas Yeadon, Andrew Mauthe, David Hutchison, and Francisco Garcia. QoS filters: Addressing the heterogeneity gap. *Lecture Notes in Computer Science*, 1045:2271–243, 1996.