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

Michael Hicks, Robbert van Renesse,
Mark Bickford, Robert Constable, Christoph Kreitz, and Lori Lorigo
Department of Computer Science
Cornell University, Ithaca, NY 14853
{mhicks,rvr,markb,rc,kreitz,lolorigo}@cs.cornell.edu

Abstract

In disaster and combat situations, mobile cameras and other types of 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 prefer. This paper describes *MediaNet*, a distributed multi-media processing system designed with the above scenarios in mind. MediaNet makes three contributions. First, unlike past approaches, MediaNet's users can specify how the system should adapt to scarce resources, based on their needs; MediaNet uses intuitive specifications called *Continuous Media Networks* for this. Second, MediaNet uses both local and on-line global resource scheduling to improve user performance and network utilization. Third, MediaNet is completely adaptive, requiring no underlying support for resource reservations. Performance experiments show that our scheduling algorithm is reasonably fast, and that user performance and network utilization can both be significantly improved.

1 Introduction

Consider a dangerous setting, such as collapsed buildings caused by an earthquake or terrorist attack. 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 in such systems.

A number of projects have explored how to provide improved quality of service (QoS) for streaming media in resource-limited conditions. These systems place computations in the network, either within routers themselves (e.g., [4, 7, 22]), or at the application-level using an overlay network (e.g., [2, 20]). These systems rely on 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 and/or priorities in making QoS decisions.

To study whether these problems can be overcome, we have been developing a system called *MediaNet* that takes a comprehensive view of streaming media delivery. Like past approaches, we use an overlay network that schedules local adaptations. However, MediaNet differs from past approaches in three key ways:

1. Rather than make QoS adaptation user-agnostic, we allow users to specify how to 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 first author was supported by the AFRL-IFGA Information Assurance Institute under grant AFOSR F49620-01-1-0312.

the preference may be the other way around. The primary goal of MediaNet is to maximize each user's utility.

2. In addition to using local scheduling, MediaNet employs a global scheduler to divide tasks among network components. Different from other projects that use global schedulers (e.g., [7, 10]), MediaNet's scheduler is continuously looking for improvements based on monitoring input.

3. MediaNet is fundamentally *adaptive*: it does not rely on hardware or software support for capacity reservations, whether for networking or computation. MediaNet runs on standard COTS components, each of which may have background loads not under the control of MediaNet.

4. MediaNet intends to exploit formal methods and tools to check the properties of generated configurations, and the correctness of configuration switching protocols (as in [18]). In the near future, we would also like to use formal tools to generate highly optimized configurations directly, for example, such as done in [17].

While there is more work to be done, our prototype and experimental measurements are promising, showing that user-based quality-of-service and improved network utilization can be achieved. On the other hand, while overheads appear low for the small networks we have considered, our system does exact a higher cost for its global adaptations. We consider our work as a step to exploring the synergy between global and local adaptation.

In this paper we present the design and implementation of MediaNet, a framework for experimenting with user-directed, QoS adaptation for streaming media. We begin by describing the MediaNet architecture (Section 2), its global resource allocation scheme (Section 3), and our prototype implementation (Section 4). We then present some experimental evidence that analyzes MediaNet's benefits (Section 5), and finish up with related work (Section 6) and conclusions and future work (Section 7).

2 MediaNet

MediaNet distinguishes three types of components (see Figure 1):

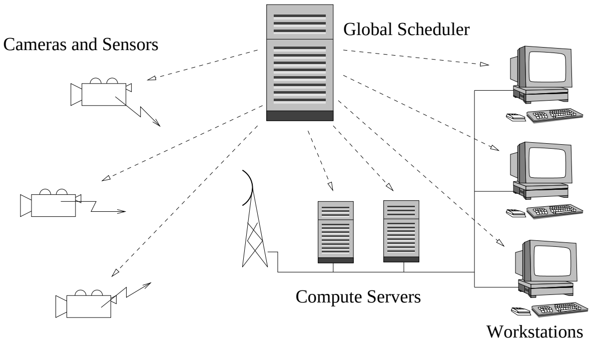


Figure 1: MediaNet architecture.

1. **Compute nodes**: these include cameras, sensors, workstations, and compute servers. Compute nodes are highly heterogeneous, in that they have different computational power, available memory, hardware support for video operations, etc. We use the terms "compute node" and "host" interchangeably.

2. **Network links**: these include both the network that makes up the wired infrastructure, and wireless links that connect robotic cameras and sensors to the wired infrastructure. These links are also highly heterogeneous. Moreover, the underlying network topology may change at run-time as components physically move around or new parts of the infrastructure are deployed.

3. **Global scheduler**: this receives all user specifications, network topology information, and various monitoring data, and assigns tasks to individual host nodes. The scheduler operates on-line, adapting in real time.

An important role in MediaNet's architecture is played by the concept of a *Continuous Media Network* (CMN). A CMN is a specification of a set of operations on a set of data sources. Users communicate their requirements by submitting CMNs to the global scheduler, while the global scheduler sends commands to the hosts in the form of CMNs. We are designing various tools that operate on CMNs, including a graphical tool for the construction and display of CMNs, and formal tools that can help verify various properties of CMNs, such as real-time constraints. In this section we will first take a closer look at these CMNs, and will then describe the scheduling approach.

2.1 Continuous Media Networks

A Continuous Media Network is a directed acyclic graph (DAG). Each node represents an operation on a set of frames (packets of data such as video frames, audio clips, etc.) that produces zero or more output frames. Ignoring the details of timing, an operation is therefore like a function that maps input frames of one type to output frames of another (possibly the same) type. Connecting operations together in the DAG is tantamount to composing functions, where the range of one function must match the domain of the one(s) it's connecting to. In this sense, operations must be well-typed.

Examples of operations include video frame cropping, frame dropping, changing the resolution or color depth, ”picture-in-picture” effects, compression, encryption, and audio volume control. Certain operations receive input and send output external to the DAG, to perform I/O with devices like video cameras and players. Operations can maintain internal state. Many complex stream-processing computations can be expressed as CMNs.

We describe CMNs using XML. In particular, a user-specification is a list of CMNs, each tagged by their utility value. A utility value is a number between 0 and 1, where 1 means most desirable. Using the utility value, a user can specify his or her relative preferences for the various specifications in the list. An example is shown in Figure 2. Each CMN is specified as a single `alt` element, with each CMN node given by a uniquely named XML `comp` element. Sub-elements describe the associated operation (`op`) and inputs of the node (`inputs`), as well as hints and constraints for the global scheduler, such as the expected inter-frame arrival interval (`interval`), and the operation's required location (`location`). The `size` attributes indicate the output frame size.

This example specifies two CMNs, one with utility value 1.0 and the other with value 0.5. In both cases, the CMN indicates a video stream is received on port 5001 at location `pc1` at an effective rate of 30 *fps* and will ultimately be streamed to the user's player on `pc2` port 5000. In the second CMN, the frame rate is reduced when B frames are dropped by the intervening `dropF` node (in this case, B frames constitute two-thirds of all frames). Note that since this node has operation `cmd="MPEG_drop"` the global scheduler can deduce that its inputs and outputs must both be MPEG streams. The `size` attributes indicate the average frame size; before B frames are dropped, this is 4833 bytes, but afterwards it is 8800 bytes, since B frames are much smaller than other frame types.[1]

We do not expect users will author these XML documents directly. Graphical interactive tools may

```
<spec>
  <alt utility=”1.0”/>
    <comp name=”recv1”>
      <op cmd=”recv” port=”5001”
        size=”4833”/>
      <location>pc1</location>
      <interval>.0333</interval>
    </comp>

    <comp name=”send1”>
      <op cmd=”send” host=”pc2”
        port=”5000”/>
      <location>pc2</location>
      <interval>.0333</interval>
      <inputs>
        <input name=”recv1”/>
      </inputs>
    </comp>
  </alt>

  <alt utility=”0.5”/>
    <comp name=”recv1”>
      <op cmd=”recv” port=”5001”
        size=”4833”/>
      <location>pc1</location>
      <interval>.0333</interval>
    </comp>

    <comp name=”dropF”>
      <op cmd=”MPEG_drop”
        frametypes=”B” size=”8800”/>
      <interval>.01</interval>
      <inputs>
        <input name=”recv1”/>
      </inputs>
    </comp>

    <comp name=”send1”>
      <op cmd=”send” host=”pc2”
        port=”5000”/>
      <location>pc2</location>
      <interval>.01</interval>
      <inputs>
        <input name=”dropF”/>
      </inputs>
    </comp>
  </alt>
</spec>
```

Figure 2: Example CMN

[1] Both MPEG and this particular specification are explained in more detail in Section 5.1.

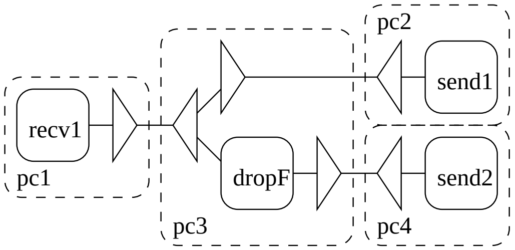


Figure 3: A global configuration

be developed that allow users to describe what they want to do in a user-friendly fashion, while they output XML documents that can be used as input to MediaNet. Furthermore, the example in Figure 2 includes scheduling details that may not (and need not) be available to the user, such as the rate and size of frames. These all depend on the particular stream being subscribed to, so we expect them ultimately to be merged into the user's initial specification.

2.2 MediaNet Scheduling

Once a user provides a specification, it must be scheduled on the network. MediaNet uses a two-level scheduling approach. A *global scheduler* (GS) combines the CMNs of individual users into a single CMN, and assigns each operation to a host. On each MediaNet host runs a *local scheduler* (LS) that implements the schedule provided by the GS. The GS will insert additional operations into the combined CMN, for example to transport frames between nodes, and to monitor resource availability. The GS works in real-time: it is regularly updated with the availability and capacities of the various hosts and resources, and new versions of the user specifications. On each update, the GS runs a scheduling algorithm (see Section 3), and then sends a job description, in the form of a CMN described by XML, to each MediaNet host.

As an example, say the system is using a Y-shaped network, where the base of the Y is host `pc1`, and the ends of its two arms are `pc2` and `pc4`, with its center at `pc3`. Let the CMN shown in Figure 2 specify the preference of user `rvr`, while user `mwh` has the same CMN, but with `send1` replaced by `send2` at location `pc4`. Figure 3 shows how these configurations might be scheduled so that user `rvr` gets utility 1.0 while `mwh` gets utility 0.5. This scheduling could arise if the link to `pc4` is congested but the other links are not.

Two things are noteworthy here. First, the GS has inserted a number of components to simply forward the data, so-called *send* and *receive* operations, depicted by right-pointing and left-pointing triangles, respectively. These are implemented by `send` and `recv` operations, like those shown in Figure 2. Second, the GS has exploited the commonality of the two specifications, creating a multicast-like effect, but generalized to CMNs. In particular, rather than run the two configurations in parallel, the scheduler collapsed the common part of the configuration to be shared by both users. In this way, the scheduler is able to better utilize the system's resources. Note that the `dropF` component in this example is functioning as a sort of RTP-style *mixer* [24]: it provides a local resource adaptation on a shared stream.

It is fairly straightforward for the LS to implement the provided CMN. It first creates data structures that represent the CMN nodes, and 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, following the topological ordering. To avoid timing and queuing overhead, when operations are data-driven the LS simply runs the operations when frames arrive, while ensuring that the frame rate approximates the prescribed interval.

3 Global Resource Allocation

At the heart of MediaNet is the algorithm used by the GS to assign operations to hosts. The algorithm works by generating possible assignments of user operations to network hosts, along with the necessary intervening send and receive operations, and assigning to each assignment a *score*, based on how effectively the user specifications are met and how efficiently the network is utilized. The assignment chosen is the one with the maximal score. While our algorithm is not guaranteed to generate an optimal solution, it is fast and should be considered a proof of concept; the performance analysis in Section 5 is based on this algorithm. We first discuss the scoring algorithm, and then indicate how assignments are chosen.

3.1 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 model tracks each host h and network link l, assigning it a *capacity* $C(h)$ for hosts in terms of instructions per second, and $C(l)$ for links in terms of bytes per second. In addition, network links are given a *latency* (in seconds), and are associated with the set of hosts that are connected by the links. With each operation o are associated cost functions $f(o)$ and $s(o)$ that return approximately the number of instructions the operation takes, and the average size of the output frame, respectively. In our implementation, cost functions are parameterized by frame inputs, architecture type, etc. Using this information, we 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.[2]

The total score is the minimum of three separate scores: the *host score*, the *network score*, and the *operation score*. These scores measure, resp., 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 as follows. We first calculate a *local score* $ls(x)$ for each of n *entities* x; for example, in the host score, we calculate the computational load $l(h)$ (the local score) on each host h (the entity). We then determine the *scaled leftover capacity* $slc(x)$ by subtracting the local score from, and dividing it by, the *local capacity* $c(x)$:

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

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 $slc(x)$ non-negative. If any $slc(x)$ is ≤ 0, we use the smallest individual value:

$$l = \begin{cases} \forall x.\ slc(x) > 0\text{:} & \dfrac{1}{\displaystyle\sum_{\text{all } x} \frac{1}{n \times slc(x)}} \\ \text{otherwise:} & \forall x.\ \mathbf{min}(slc(x)) \end{cases} \tag{1}$$

Using this technique, we calculate the three scores as follows. 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 capacity $C(h)$. The computational load is, for every operation o on host h, the cost of the operation $f(o)$ divided by its specified minimum interval $i(o)$:

$$L(h) = \sum_{o \in \mathbf{ops}(h)} \frac{f(o)}{i(o)}$$

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 $C(l)$. The bandwidth $B(l)$ is calculated by summing the relevant output frame sizes $s(o)$ divided by their intervals.

Finally, for the operation score, the entities are the operations, the local score is the operational delay $D(o)$, and the capacity is the user's maximum acceptable delay $mad(o)$. 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:

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

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 gets to run twice once an operation p becomes runnable. We then determine the maximum delay on each network in a similar manner, but account for the network's latency. 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):

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

Here, $\mathbf{net}(p, o)$ is the delay of the network connecting operations p and o (which will be zero if o is not a receive operation), and h is the host on which o is scheduled. 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 $mad(o)$.

3.2 Creating 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,

[2] In the future we intend to use more detailed monitoring so that the GS can improve the approximations over time.

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 n^m possible assignments, where m is the number of operations, and n is the number of nodes. This means a brute force enumeration of assignments is infeasible.[3] Therefore, to construct a tractable algorithm requires a way to prune the assignment space. We present an algorithm here that is roughly $O(mn\ \mathbf{log}\frac{1}{\epsilon}\ +\ n^3)$, where ϵ relates to the utility space (defined below). Though it can be improved, it shows that a reasonable algorithm exists that works well in the scenarios we have considered.

We create an assignment of nodes 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 inner phase tries to find a reasonable assignment for a given global CMN (as opposed to a user-specified CMN). To do this, it first 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. Before beginning the outer phase, we calculate the all-pairs "shortest" paths of the network, using maximum *bottleneck bandwidth* (also referred to as the *path bandwidth*) as the metric of optimality; this takes time $O(n^3)$. From this we choose the most bandwidth-plentiful links to create a tree of paths originating from each operation to its immediate downstream neighbors in the CMN. At each intermediate node in this tree, the scheduler inserts receive and send operations to forward the data. It can then calculate the score.

At this point, it tries to improve the score by relocating each (movable) operation to each possible host, remembering the location that improves the score most. This process continues until no more operations can be moved.

If MediaNet users have different priorities with respect to resource usage, the scheduler may use the following simple strategy (not yet implemented). First, the scheduler runs a binary search on the CMN lists of the highest priority users. Then it adds the CMN lists of the next highest priority users, and runs the binary search again, but using the chosen CMNs for the first group of users. The scheduler continues this until all users have been dealt with, or the algorithm fails. In the latter case, the scheduler notifies the leftover users, indicating that it is available to service them. Other prioritization schemes are also possible.

3.3 Discussion

Here we discuss some aspects of the algorithm we have presented, including a rough characterization of its running time, as well as shortcomings and potential enhancements.

Running Time As mentioned, our algorithm runs in roughly $O(mn\ \mathbf{log}\frac{1}{\epsilon}\ +\ n^3)$ time. The n^3 component comes from the all-pairs, shortest paths computation which precedes the outer phase. The mn component comes from the fact that for each of the m operations, we greedily choose the best of all available nodes. Finally, the calculation occurs $\mathbf{log}\frac{\mathbf{1}}{\epsilon}$ times due to the use of binary search. This characterization does not include the optimization of improving single users' utilities.

As a sanity check, we set up a simple experiment to measure the algorithm's performance in practice (see Section 5 for our experimental setup). Using the BRITE topology generator [6], we created a 50 node, 150 link network and fed it into the algorithm, along with specifications from ten users, each using roughly the specification from Figure 2 (see Section 5.1 for an exact description) subscribing to one of five data sources. This setup is an attempt to model our motivating scenarios, e.g. a disaster scenario explored by camera-carrying robots. In this case, the algorithm was able to complete in roughly $20ms$. As expected, the majority of the time was spent in the shortest path computation, comprising about $18ms$. We plan to perform more detailed sim-

[3] It is likely that this scheduling problem is NP-complete, though we have not yet proven as much.

ulations and analyses, but we are encouraged that it handles reasonable scenarios.

Shortcomings While fast and useful in our experience, our algorithm has some shortcomings when considering multiple users. One serious shortcoming is that it considers the utility values that users provide as globally absolute, rather than relative to each user. Users may exploit this by choosing low utility values for operations that use extensive resources. To fix this, we need to scale a CMN's utility value by the resources the CMN requires. At the moment, we rely on users to choose their utility values fairly based not only on preference but also on resource usage.

Another problem is that when generating paths between CMN nodes in the actual network, we do not consider interference from competing users. In particular, when inserting send and receive operations between any two nodes in the network, the algorithm will always choose the same path (based on the shortest path computation). This means that if multiple users have non-collapsable computations on the same two nodes, both data streams will be sent along the same path, even if redundant paths are available. One simple way to fix this would be to allow moving of inserted send and receive computations after they are chosen, much in the way we move user operations, and take the best performing location.

Potential Enhancements Though we do not currently take advantage of it, our system allows each score (i.e. host score, network score, and/or operation score) to be weighted when combining them into the single score by taking each to a particular 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. This way, we can put more emphasis on one facet of system resource use versus another. This weighting could even be performed for particular local scores that combine to make up any of the three scores.

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 may prove intractable to implement algorithmically.

4 Implementation

We have implemented a MediaNet prototype. The LSs are written in a type-safe systems language called Cyclone [15], comprising roughly 13,000 lines of code. Cyclone is simply C at its core, but with restrictions to ensure type-safety (e.g. no unsafe casts, arbitrary pointer arithmetic, etc.) and enhancements for greater flexibility (e.g. exceptions, tagged unions, garbage collection, a variety of safe pointer types, etc.). The GS is written in C, consisting of about 10,000 lines of code, of which roughly 2,800 lines implements the scheduling algorithm. For the remainder of this section we present the relevant details on how we implement local scheduling, monitoring, and reconfigurations.

4.1 Local Scheduling

Each LS is implemented around a single-threaded event loop in conjunction with `select` on non-blocking sockets. For each operation in a CMN, the local scheduler allocates a structure that has callbacks for 'received packet' events and 'deadline expiry' events, used as described in Section 2.2. To forward data, an operation invokes the callback(s) of its downstream nodes, following an upcall-style approach.

By default, we implement send and receive operations using TCP. For example, in Figure 2, operation `send1` is specified to connect to host `pc2` on TCP port 5000. The main benefit of TCP for us is that it readily communicates network congestion to the application. In particular, when the TCP send buffer fills due to congestion, the application receives an `EWOULDBLOCK` error. We use this in our implementation to "block" the socket and queue the packets in the application until the congestion subsides (i.e., when `select` indicates the socket is once again writable). Once the application queue is filled (currently size 10), packets are dropped based on priority. User-supplied operations are used to set the priority, which is high by default. The chief disadvantages of TCP are greater overhead and loss of control, and that it performs poorly over lossy links. For our study, TCP was an expedient choice, and we could imagine using something more appropriate, such as RTP [24] over UDP.

4.2 Monitoring

Each `send` operation is associated with a network link by the GS, and packet throughput is collected for each link, including both packets sent and packets dropped. On regular intervals (currently every second), this information is aggregated and reported to the GS. For simplicity, we current use XML messages embedded in HTTP POSTs.

In the worst case, the number of monitoring messages grows with the number of nodes, since all link reports are placed in one message, and the number of links per node is typically small. While in the experiments we present in Section 5 monitoring overhead is low, due to having only a few nodes, the global scheduler will start becoming taxed as the network size increases. One way to deal with this is to reduce the monitoring frequency, and/or to perform hierarchical aggregation or gossiping An easy modification would be to avoid sending messages in non-overload conditions (i.e., unless frames are being dropped), where the GS assumes that things are as it expects.

For each link, the GS maintains an estimate of the link's available bandwidth. This variable is maintained as follows:

1. If the reported available bandwidth exceeds the estimate, or if the measurement is less than the estimate and packets are being dropped (indicating the link is at peak capacity), then the estimate is changed to the reported value. Note for broadcast links, reports from different LSs must be combined.

2. At regular intervals (currently every second), the GS "creeps" its bandwidth estimates, assuming that additional bandwidth might now be available; this is in the spirit of TCP's additive increase. While more experimentation is needed, we currently increase the estimate by a constant $w = 3\%$ each second. The higher the value of w the more frequently the GS will attempt to reconfigure due to greater perceived resources, making it more adaptive. The tradeoff is that when it is wrong, the reconfiguration will quickly fail, resulting in a potential disruption. A dynamic determination of w (perhaps based on round-trip times) would help.

We have experimented with using other means to estimate available bandwidth on underutilized links, but none has so far proven to be more robust than bandwidth creeping. In particular, we have implemented an estimator using packet-pair measurements [8], but found that its estimates can be wildly inaccurate, more closely approaching the link bandwidth than the available bandwidth resulting in premature reconfigurations. Available bandwidth detection is an ongoing area of research with no clear solutions as yet [8], though some recently proposed work may be promising [13].

4.3 Reconfiguration

Each time new user specifications, network topology updates, or monitoring information arrives at the GS, the GS runs the scheduling algorithm. If the algorithm generates a configuration that is different from the old configuration, MediaNet initiates a reconfiguration phase.

To initiate a global reconfiguration, the GS sends new CMNs to the LSs via HTTP POST. Because different LSs may receive these messages at different times, we require a form of synchronization to ensure that data associated with an old version is not incorrectly forwarded to a new version, which would be disastrous if that data changed type. To prevent this, the global scheduler assigns different TCP port numbers to its inserted send and receive operations on each reconfiguration.

Naively, the LS could close all existing connections when notified of a reconfiguration, and wait for its neighbors to connect on the new ports. However, this results in lost data, both from the application queues and the socket send and receive buffers. To prevent this, we execute the following simple synchronization protocol that piggybacks on normal dataflow.

When a reconfiguration message is received, the LS sets a flag and caches the message. Next, the LS closes any receive operations whose input does not originate from MediaNet itself (i.e., video stream sources) immediately after grabbing any data from the socket receive buffer. This data is processed, and then the LS sends a 'flush' packet to the downstream operations. When the packet arrives at a MediaNet receive operation, the receiver closes the connection, which closes its upstream send as well. When the flush packet arrives at a non-MediaNet send operation (i.e., for the user's video player), the connection is closed and the flush packet is dropped. Once all of the send connections are closed on a given LS, the reconfiguration takes place.

An example is shown in Figure 4. Here, node A receives a reconfiguration message, so it sends a flush packet downstream from its `recv` operation, since its source is from outside MediaNet (i.e. the camera). This is forwarded on to node B, where the MediaNet `recv` connection notices the packet, and so closes its connection, thereby closing node A's `send` connection as well. Note that the flush packet is queued behind any packets in the A's send queue, so it will not be received at B until all of these pack-

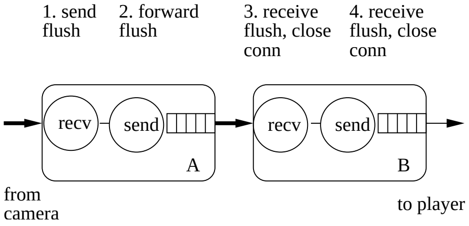


Figure 4: An example reconfiguration

ets are processed. Finally, the flush packet reaches node B's `send` operation, so the connection is closed and the packet is dropped.

This protocol does not eliminate dropped packets entirely because some or all of the data grabbed from the receive buffer may constitute only a partial frame, and will thus be dropped, and because data can arrive just before the connection is closed. We can eliminate all unprocessed data by not closing input stream connections, but rather associating them with the appropriate operations in the new configuration; we would probably need to perform some buffering during the reconfiguration. This should be straightforward to implement.

5 Experiments

In this section, we present experiments that measure MediaNet while delivering an MPEG video stream under various topologies and load conditions. We compare MediaNet's performance to two baseline configurations: one in which no adaptation takes place, and one in which adaptation takes place purely locally, without benefit of global coordination. Both approaches are implemented using MediaNet's LSs without the global scheduler.

These experiments were conducted on a 1 GHz Pentium III with 250 MB of RAM running Linux kernel 2.4.7-10 as part of RedHat Linux 7.2. We simulated the presented network topologies by running the GS and multiple LSs on the same machine. Within the LSs, a shim layer below the application simulates the effect of dropped packets; if sending the packet would exceed the (dynamically alterable) bandwidth quota, the shim layer returns an `EWOULDBLOCK`. The main disadvantage here is that this approach does not accurately model TCP's backoff behavior, and it ignores some of the cost for transmitting the data over a network. For the experiments, each simulated CPU has capacity 10^7 and each network link has capacity 300 KB/s, simulating a wireless link. In the near future, we plan to run the same experiments on Emulab [9].

5.1 Configuration

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

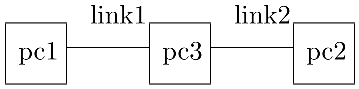
Recall that I frames are essentially JPEG pictures, while P frames and B frames exploit temporal locality, including "deltas" from adjacent frames. P frames rely on the most temporally-recent P or I frame, and B frames rely on the prior I or P frame, and the next appearing I or P frame. Therefore, I frames are the most important, and next the P frames, while B frames are the least important.

For configuring MediaNet, we used the user CMN depicted in Figure 2, with two differences. First, each `alt` configuration includes an additional `MPEG_prio` operation for setting the drop priority of MPEG frames. Second, we duplicate the utility 0.5 configuration at utility 0.1 but dropping both P and B frames (the `MPEG_drop` operation has attributes `frametypes="PB"` and `size="13500"`, and interval 0.5).

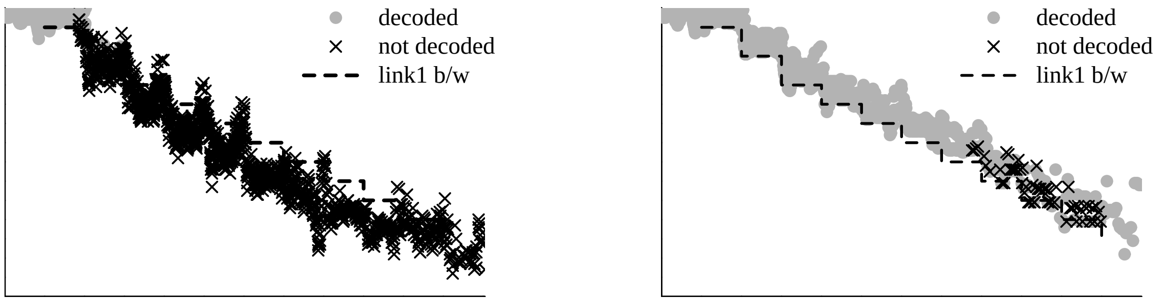
The `size` and `interval` annotations on the user specification arise from the particular video we are considering. In the configuration with utility 1.0, the `recv` computation notes the arrival interval between frames as 0.0333; this comes from the fact that the movie is 30 fps, and thus the inter-frame interval is 1 30 = 0.0333. The `size` annotation is simply the average framesize of the movie: 4833 bytes.[4] For utility 0.5, the `ecv` computation is labeled as before, but the `dropB` computation has an arrival interval of 0.1 (i.e. 10 fps) and an average framesize of 8800, since now only I and P frames are being sent. Finally, for utility 0.1, we label `dropPB` with an arrival interval of 0.5 and an average framesize of 13500.

5.2 Single Path Adaptation

We first consider a simple configuration in which three nodes are placed in a line, with the movie sender on `pc1` and the player on `pc2`:



[4] 13500 × 2 + 7625 × 8 + 2850 × 20)/30 = 4833.



(a) No adaptivity

(b) Local adaptivity

Figure 5: User-perceived performance as available bandwidth diminishes

5.2.1 Local Adaptation

For our baseline cases, “local adaptation” consists of tagging video frames with priorities, so that the least important frames are dropped during overload. Because our implementation is in user-space, we only drop those frames that are queued within the LS, and not those already in the kernel send buffer. The experiment measures the video player’s performance, in terms of the received bandwidth and the decodable frames, as we lower `link1`’s available bandwidth over time.

Each of the graphs we will show in this section has the same format. Each circle in the figure is a correctly-decoded frame, while each × 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 is shown as a dashed line. Dropped frames are not shown.

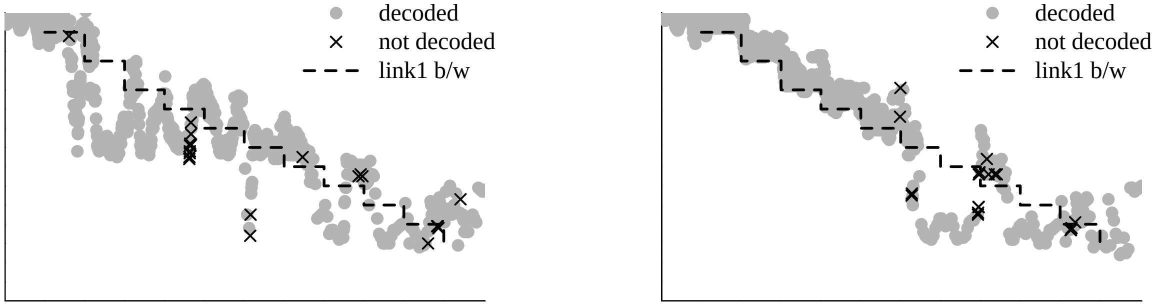
Figure 5(a) shows the no adaptivity case. We can see that the video quality is quite poor: once congestion kicks in, the application cannot decode any of the frames it receives because temporally important frames (I and P frames) are being indiscriminately dropped.

In contrast, when using local adaptation, the performance improves significantly, as shown in Figure 5(b). Until roughly time 75, the player can decode all of its received frames. Then from time 75 to 120, a number of frames cannot be decoded properly. This is because at this point we are only sending I or P frames, so any dropped P frame could prevent downstream P frames from being decoded. During playback, this manifests as a “glitch” noticeable by the user. In this case, the large 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 (i.e. until a circle is reached in the figure).

5.2.2 Global Adaptation

The MediaNet performance is shown in Figure 6. In 6(a), the GS performs all adaptation by proactively dropping B frames when it detects that there is not enough bandwidth (at about time 20), and later dropping both P and B frames (at about time 110). During the run, the GS optimistically tries to return to the higher bandwidth configuration but fails, accounting for the spikes upward at times 20, 30, and 40 (among others). The undecoded frames arise from packets dropped during reconfiguration, as was described in Section 4.3.

Figure 6(a) illustrates that MediaNet’s GS is able to provide good overall viewing quality, whatever the available bandwidth, whereas the local case has poor performance while dropping P frames during times 75-110. While MediaNet reduces the bandwidth, particularly during times 20-60 when it is dropping *all* B frames, user quality is not significantly reduced (10 *fps* as opposed to 30 *fps* is hardly noticeable to most users). Even so, we could combine local and global adaptation to improve the situation: provide priority-based frame dropping of B frames, and then perform global adaptation to proactively drop all P frames. This approach is taken in Figure 6(b). Until roughly time 60, MediaNet has the same performance as the local case, but



(a) Global adaptivity

(b) Global/Local adaptivity

Figure 6: MediaNet performance under diminishing B/W

then gets better viewing quality because it starts dropping all of its P frames.

While end-user performance is improved in Figure 6(b), this style of adaptation can reduce network utilization. In particular, if congestion were occurring on `link2` rather than `link1`, then the configuration used in Figure 6(b) would send the full stream along `link1`, only to have some of its frames dropped on `link2`; thus some of the bandwidth sent on `link1` would be wasted. In contrast, using the global-only adaptation shown in Figure 6(a) would result in the `dropF` component being inserted on `pc1` as soon as congestion was detected on `link2`, preventing wasted bandwidth on `link1`. More work is needed to crystallize these tradeoffs and incorporate appropriate metrics into the scheduler.

5.3 Finding Alternate Paths

Perhaps the most significant benefit of global adaptations is that they can better utilize redundant paths, which are not uncommon in the wide area [23]. To illustrate this, we set up an experiment in which there is an additional node between the sender on `pc1` and the player on `pc2`, forming a diamond:

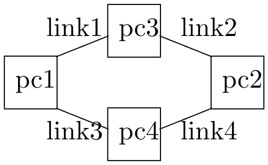
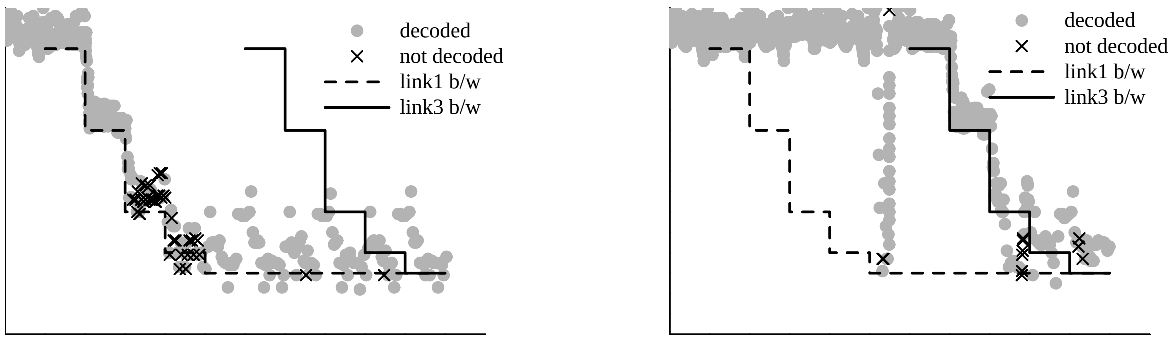


Figure 7 shows the performance of the MediaNet as compared to just local adaptation. The experiment starts with the path going through `pc3`, and the bandwidth is incrementally reduced on `link1`. For the local case, frames are dropped as the bandwidth is reduced, while for MediaNet, the system is reconfigured so that the traffic goes through `pc4`, utilizing the idle `link3` as opposed to the congested `link1`. Later, `link1`'s bandwidth is reduced as well, which causes MediaNet to start dropping frames until it reaches the same level as the local case.

Again, MediaNet must guess as to the possible available bandwidth on unmaximized links, so it unsuccessfully attempts to reconfigure the system a number of times, most obviously at time 50. Here, enough time has passed since it last used `link1` that it thinks the bandwidth of the two links is the same, so it tries to switch back to its preferred configuration through `pc3`. This fails, as noted by the quick drop in bandwidth, and so the configuration is restored to going through `pc4`. Reconfiguration attempts also occur at times 90 and 105. In these cases, however, the GS attempts to upgrade the user configuration on the same path (going from dropping P and B frames to dropping B frames adaptively).

We currently exploring ways to prevent the possiblity of "route-flapping" suggested by the reconfiguration attempt at time 50. One possibility is to only reconfigure if the total score (see Section 3.1) exceeds the current score by some Δ, where the larger the Δ, the greater the preference for the current configuration. Another possibility is to maintain "confidence" measures for link bandwidth estimates, so that mostly estimated links are not weighted as highly in the shortest path computation.



(a) Local adaptivity

(b) Global/Local adaptivity

Figure 7: Local vs. Global adaptivity with redundant paths

6 Related Work

The idea of multi-media processing in the network was first inspired by the problems of digital video broadcasting in heterogeneous networks [26, 21]. The growing popularity of the World Wide Web inspired many follow-up projects. A first practical implementation of such ideas was described in [28]. Users could configure various filters on network servers according to the users' needs. Within the context of Active Networking [25], various other projects also have tackled the problem of multicasting in heterogeneous networks, for example [4, 22]. Other projects have targeted the dissemination to mobile, wireless workstations, such as Quasar [12] and Odyssey [19].

The MeGa Media Gateway [2] is a server that allows users to upload and instantiate so-called *service agents* that can process on video streams. A collection of these gateways can be set up in a cluster configuration, called an Active Service [3]. This allows services to be composed much like in MediaNet. Another such project is Degas [20]. Degas contains decentralized protocols for task distribution and load balancing. Neither project allows user specified adaptation, however.

In CANS [10], a centralized plan manager constructs a data path in a network of wide-area services, and uses heuristics to maximize the minimum bandwidth along each path. Other projects that do automatic path creation include Conductor [27], Ninja [11], and PATHS [5]. The Darwin project [7] uses a hierarchical resource management strategy that includes resource reservation and Active Networking techniques. Again, MediaNet differs from these systems in that it allows users to specify how they wish to adapt to varying resource availability.

BBN's UAV Open Experimental Platform [16] allows users to specify how they wish to adopt to changes in available resources. The adaptation specification is very low-level: the users have to specify how they wish to react to particular resource changes, such as the bandwidth dropping on a particular link. In MediaNet, we expect most users to not even be aware of what links exist. Since the UAV OEP currently allows only one compute server (called the *distributor*), no global scheduling is necessary. The MediaNet project was strongly inspired by the UAV OEP, but provides a much higher-level specification of adaptation, and supports multiple compute servers.

7 Conclusions

MediaNet is a system for experimenting with user-specified, globally-adaptive quality of service in distributed streaming media applications. It has three clear benefits:

1. **User-specified Adaptation**. Users specify how adaptation should take place, and this is considered in system-wide scheduling decisions.

2. **Improved user performance**. By combining both global adaptations (e.g. by utilizing redundant paths) and local adaptations (e.g. by using reactive, priority-based frame dropping), users obtain higher utility.

3. **Efficient resource utilization**. Because of the use of an on-line global scheduler, MediaNet can more efficiently utilize system resources, in three ways:

   (a) It aggregates user-specified continuous media networks, thus removing redundancy in a multicast-like fashion (e.g. as in Figure 3).

   (b) It utilizes redundant resources, such as alternative, uncongested routing paths.

   (c) It adapts proactively to prevent wasted resources, for example by dropping frames closer to the source when there is downstream congestion.

7.1 Future Work

While our work is a promising first step, many questions remain. These can be broken down in terms of *scalability*, *accuracy*, and *applications*.

Scalability The largest outstanding questions concern scalability. For example, how will the scheduling algorithm perform as more users and nodes are added? How can we ensure configuration stability in a highly volatile network setting? How much overhead will monitoring impose? How can we eliminate the GS as a central point of failure?

We plan to seek answers to these questions by pursuing a number of research directions. First, we plan to use formal tools, in particular Nuprl [1], to check the properties of our current components and protocols (as in [18]) and to automate the process of generating and/or annotating specifications (as in [17]). We are currently developing a formal semantics of CMNs and proving relevant properties.

Second, we plan to explore a more hierarchical approach to scheduling, such as used in Darwin [7]. In particular, rather than have the GS determine a single CMN for each LS, we could have it communicate a number of possible CMNs (derived from different utility-levels), along with criteria for choosing between them. These new specifications would allow adaptive scheduling to be more distributed, thus reducing monitoring overhead and the impact of a GS failure. This approach could be thought of as automatically generating local scheduling algorithms (perhaps resembling the QuO contracts used by the BBN OEP [16]) from a global algorithm and user specifications. The challenge here is balancing the reduction in overhead due to imposed hierarchy with the utilization and performance benefits of a global network view.

Accuracy One of the problems with using a global scheduler is that it will never have completely accurate information, due both to reporting delays or measurement inaccuracies. Using a more hierarchical approach will help somewhat with the first problem. To deal with the second, we are considering more accurate monitoring code. In particular, we are interested in developing on-line, available bandwidth estimations, and combining these with the bandwidth creeping approach we currently employ. As mentioned, some work in this direction has been done by Jain and Dovrolis [13].

Another inaccuracy with our current setup is our use of TCP. Since we imagine scenarios involving wireless transmission of streaming media, we need to consider something that better deals with lossy links.

Applications Finally, while we have focused on multimedia streams for our application domain, we believe MediaNet is general enough to apply to more general publish/subscribe applications, particularly ones that employ streaming data. For example, we could readily support streaming stock quote data with corresponding operations, like in-network filtering. More aggressively, we are interested in applying MediaNet to the Air Force's Joint Battlespace Infosphere (JBI) [14]. The JBI is a general infrastructure for information dispersement, along with user-defined aggregation/filtering computations carefully located in the network, called *fuselets*. We believe MediaNet is a good fit for scheduling JBI streams and computations, but the large open question is how to make it scale to upwards of 1000 nodes, as might be required in a large combat setting.

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, and Bobby Bhattacharjee for helpful comments on earlier versions of this paper.

References

[1] S. Allen, R. Constable, R. Eaton, C. Kreitz, and L. Lorigo. The Nuprl open logical environment. In D. McAllester, editor, *17th Int. Conf. on Automated Deduction*, volume 1831 of *Lecture Notes in Artificial Intelligence*, pages 170–176. Springer Verlag, 2000.

[2] E. Amir, S. McCanne, and Z. Hui. An application level video gateway. In *ACM MULTIMEDIA'95*, pages 255–266, November 1995.

[3] E. Amir, S. McCanne, and R. Katz. An Active Service framework and its application to realtime multimedia transcoding. In *ACM SIGCOMM'98*, pages 178–189, September 1998.

[4] S. Bhattacharjee, K. Calvert, and E. Zegura. On Active Networking and congestion. Technical Report GIT-CC-96-02, College of Computing, Georgia Tech, 1996.

[5] J.M. Bjorndalen, O. Anshus, T. Larsen, and B. Vinter. Paths – integrating the principles of method-combination and remote procedure calls for run-time configuration and tuning of high-performance distributed applications. In *Proc. Norsk Informatikk Konferanse*, pages 164–175, November 2001.

[6] BRITE: Boston university Representative Internet Topology gEnerator. `http://www.cs.bu.edu/brite/`.

[7] P. Chandra, A. Fisher, C. Kosak, T. S. E. Ng, P. Steenkiste, E. Takahashi, and H. Zhang. Darwin: Customizable resource management for value-added network services. In *IEEE ICNP'98*, pages 177–188, October 1998.

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

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

[10] X. Fu, W. Shi, A. Akkerman, and V. Karamcheti. CANS: Composable, adaptive network services infrastructure. In *USITS'01*, March 2001.

[11] S.D. Gribble, M. Welsh, R. Van Behren, E.A. Brewer, D. 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 (Special Issue on Pervasive Computing)*, 35(4):473–497, March 2001.

[12] J. Inouye, S. Cen, C. Pu, and J. Walpole. System support for mobile multimedia applications. In *ACM NOSSDAV'97*, pages 143–154, May 1997.

[13] M. Jain and C. Dovrolis. End-to-end available bandwidth: Measurement methodology, dynamics, and relation with TCP throughput. In *ACM SIGCOMM'02*, August 2002. To appear.

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

[15] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, June 2002. To appear.

[16] D.A. Karr, C. Rodrigues, J.P. Loyall, R.E. Schantz, Y. Krshnamurthy, I. Pyarali, and D.C. Schmidt. Application of the QuO quality-of-service framework to a distributed video application. In *Proc. of the International Symposium on Distributed Objects and Applications*, September 2001.

[17] X. Liu, C. Kreitz, R. van Renesse, J. Hickey, M. Hayden, K. Birman, and R. Constable. Building reliable, high-performance communication systems from components. In *Proc. of the 17th ACM Symp. on Operating System Principles*, Kiawah Island Resort, SC, December 1999.

[18] X. Liu, R. van Renesse, M. Bickford, C. Kreitz, and R. Constable. Protocol switching: Exploiting meta-properties. In *Int. Workshop on Applied Reliable Group Communication*, Phoenix, AZ, April 2001.

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

[20] W.T. Ooi, R. van Renesse, and B. Smith. Design and implementation of programmable media gateways. In *ACM NOSSDAV 2000*, June 2000.

[21] J. C. Pasquale, G. C. Polyzos, E. W. Anderson, and V. 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.

[22] R.S. Ramanujan and K.J. Thurber. An active network-based design of a QoS adaptive video multicast service. In *ACM NOSSDAV'98*, pages 29–40, July 1998.

[23] S. Savage, A. Collins, E. Hoffman, J. Snell, and T. Anderson. The end-to-end effects of internet path selection. In *ACM SIGCOMM'99*, pages 289–299, September 1999.

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

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

[26] T. Turletti and J. Bolot. Issues with multicast video distribution in heterogeneous packet networks. In *Packet Video Workshop*, pages F3.1–3.4, September 1994.

[27] M. Yavis, A. Wang, A. Rudenko, P. Reiher, and G.J. Popek. Conductor: A framework for distributed adaptation. In *IEEE HotOS'99*, pages 44–49, March 1999.

[28] N. Yeadon, A. Mauthe, D. Hutchison, and F. Garcia. QoS filters: Addressing the heterogeneity gap. *Lecture Notes in Computer Science*, 1045:2271–243, 1996.