An Application-level Quality of Service Architecture for Internet Collaboratories

G. Robert Malan and Farnam Jahanian
University of Michigan
Department of Electrical Engineering and Computer Science
1301 Beal Ave.
Ann Arbor, Michigan 48109-2122

Phone: (313) 763-5243

E-mail: {rmalan,farnam}@eecs.umich.edu

1 Introduction

The availability of ubiquitous network connections in conjunction with significant advances in hardware and software technologies have led to the emergence of distributed collaboratories over wide-area networks. In these collaboratories, people from a global pool of participants can come together to perform scientific work or take part in meetings without regard to geographic location. A distributed collaboratory provides: (1) human-to-human communications and shared software tools and workspaces; (2) synchronized group access to a network of data and information sources; and (3) remote access and control of instruments for data acquisition. A collaboratory software environment includes tools such as whiteboards, electronic notebooks, chat boxes, multi-party data acquisition and visualization software, synchronized information browsers, and video conferencing to facilitate effective interaction between dispersed participants.

Collaboratories are inherently real-time applications. The participants collaborate over data in real-time, and expect to see and propagate their ideas and actions as if they were working in the same physical space. In a wide-area Collaboratory, specifically an Internet collaboratory, hard real-time delivery guarantees cannot be met; however, soft guarantees can be provided for a collaboratory's data delivery that provide the illusion of a simultaneous meeting.

Participants in a wide-area collaboratory vary in their hardware resources, software support and quality of connectivity. In an environment such as the Internet, these participants are interconnected by network links with highly variable bandwidth, latency, and loss characteristics. In fact, the explosive growth of the Internet and the proliferation of intelligent devices is widening an already large gap between these clients. Collaboratory participants share many types of media: video, audio, text, pictures, data from real-time instruments, etc. The throughput required to simultaneously support these multimedia streams can very often surpass slower participants' available bandwidth. A significant problem, is bandwidth allocation for these poorly connected participants. A collaboratory's bandwidth allocation scheme must effectively support group collaboration. We argue that due to the high-level semantic constraints on this bandwidth, an application-level mechanism for bandwidth allocation should be used.

We have implemented a middleware architecture that supports the allocation of this scarce resource by providing application-level semantic-based Quality of Service policies. This work is motivated by our experiences with the UARC distributed system during the last four years. The Upper Atmospheric Research Collaboratory (UARC) is an experimental distributed testbed developed at the University of Michigan to examine issues in supporting collaborative scientific work over wide-area networks [1, 6, 5]. The UARC testbed connects a geographically dispersed community of space scientists via the Internet. These scientists perform experiments on remote instruments, evaluate their work, and discuss experimental results in real-time over the Internet. The main media that the UARC scientists share are: real-time data sets from remote scientific instruments, periodic still photographs, and text-based chat messages. Notice that all of these media types cannot suffer arbitrary packet losses in their transmission. This illustrates a split between data that must be totally received versus data that can suffer loss. Specifically, we make the distinction between these two types of real-time data by placing them in one of the following categories: continuous or discrete data. Continuous data are those characterized by a stream of continuous packets. Examples of continuous data are video and audio streams. The semantic information in these streams is spread over many network packets, all of which don't need to be received to provide useful information. Discrete data are those that provide a separable and discrete semantic in a specific number of network packets. Examples of discrete data are: pictures, postscript files, instrument data sets, Java code, etc. In general, all of a discrete data's network packets must be received to provide useful information. Since the majority of the UARC media are discrete, we focused our collaboratory architecture on supporting bandwidth allocation for discrete real-time data. However, we made sure that our architecture could easily accommodate continuous real-time data.

2 System Architecture

The system is comprised of both an underlying architecture and a middleware services interface. These support the framework for providing application-based quality of service to real-time collaboratory data streams. The collaboratory's architecture consists of three pieces: data suppliers, servers, and clients. An example system is shown in Figure 1. This figure shows three server clouds with associated suppliers and clients. In general, the suppliers feed data to the set of servers. In turn, these servers disseminate the data to client subscribers, the collaboratory's participants. In the UARC application, the data suppliers are located near the atmospheric instruments in Greenland. They package the raw data and send it over the Internet to a network of distributed servers. These servers then reliably redistribute the data to all the connected clients. The middleware at the supplier currently consists of C libraries. The client support takes the form of both Java classes and C libraries. The server architecture is currently a federation of C programs; however, work has begun on porting the code to Java.

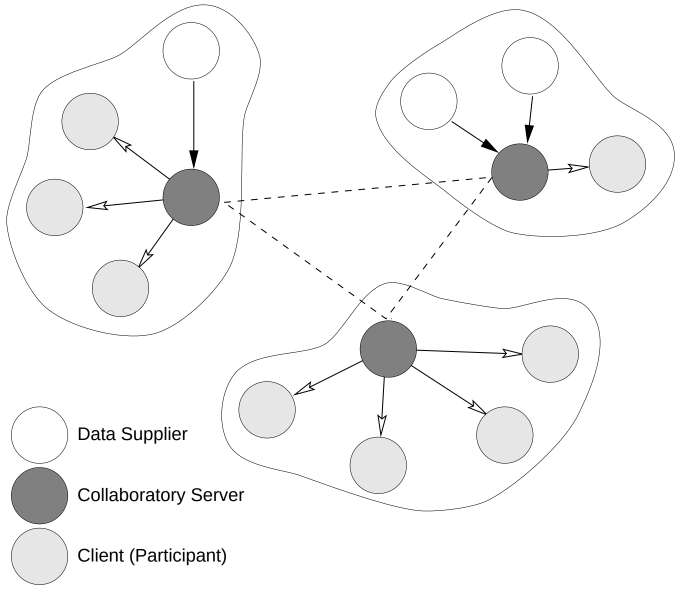


Figure 1: An example collaboratory with three servers, several data suppliers, and scattered participants. The real-time data suppliers feed information to a single server that can then propagate the information to other servers across the Internet. These servers then distribute the data to subscribed participants using application-level quality of service policies.

The collaboratory system's architecture is utilized as a middleware service interface: the Collabratory

Middleware Services. This middleware is a collection of components for the distribution and dissemination of shared data to support groupware applications over wide-area networks. As illustrated in Figure 2, the key services in this middleware are: a *transport service* which allows easy integration of application-level quality of service to adapt to network and resource variability, a *group membership service* for maintaining the list of active participants in each collaboration group and the health of its connections, a *shared state service* for maintaining the replicated state of a groupware application, and a *synchronization service* for managing shared access to resources. The remainder of this paper focuses on application-level quality of service policies and the collaboratory's transport services.

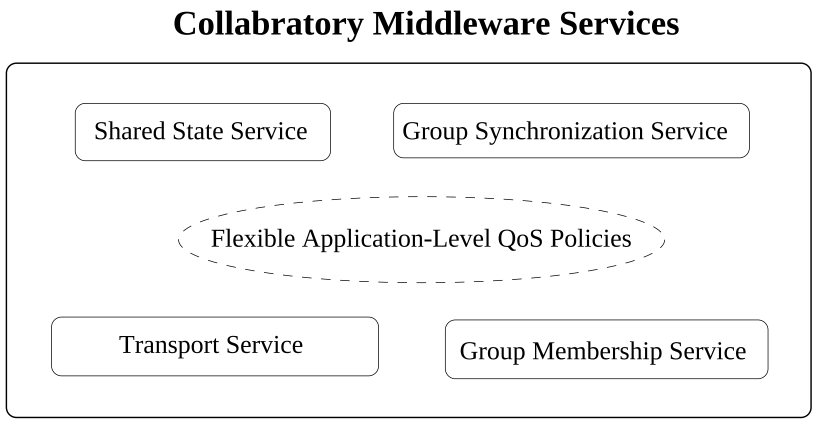

Collabratory Middleware Services
Shared State Service
Group Synchronization Service
Flexible Application-Level QoS Policies
Transport Service
Group Membership Service

Figure 2: Collabratory Middleware Services

3 Application-level QoS and Transport Services

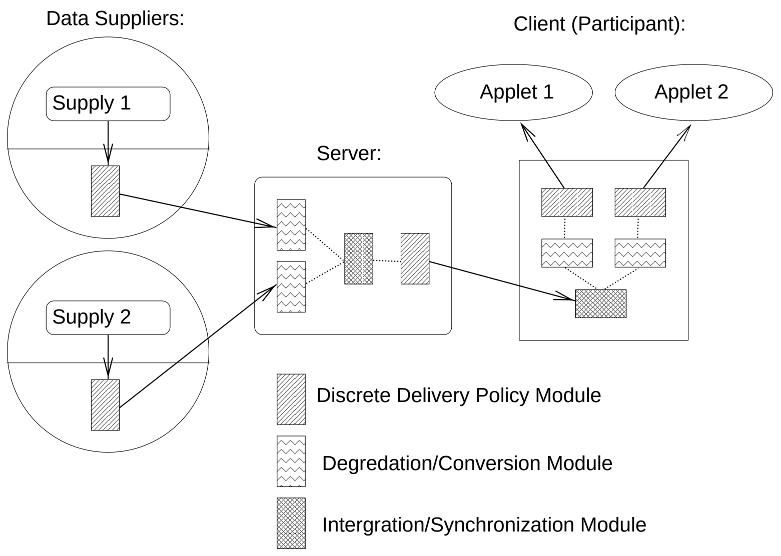
Our architecture is designed to provide real-time data delivery over the Internet without relying on network-level quality of service guarantees. Instead, application-level quality of service policies are used to allocate the available bandwidth between the subscribed streams. These policies provide a participant with an effective throughput based on semantic thresholds that only the application and user can specify. If network-level guarantees are available [10], they can of course be exploited.

These application-level qos policies are provided by utilizing dynamically placed data processing modules in the real-time data stream paths. Figure 3 shows an example topology with several types of modules. These modules generalize into several categories: filters; synthesizers and harmonizers; and schedulers. Filters degrade data streams based on their specific semantic types. Synthesizers are used to coalesce different streams into a single logical stream; while harmonizers are used to synchronize several real-time data streams. Finally the schedulers provide a mechanism for the direct allocation of the available network bandwidth. Providing application specific protocol handlers is not a new idea. They are used in [2, 9] to provide application-specific protocol handlers in an operating system kernel. Active network proponents [8] argue that Internet routers should be enabled to run arbitrary application-level code to assist in protocol processing. Our work takes the middle ground: an explicit topology that an application can administer to allow for protocol processing at any point along the datapath.

The collaboratory's server architecture is next introduced to illustrate how various collaboratory data streams can be managed using dynamic data processing modules. Afterwards, discrete data delivery policies and on-demand data degradation are described. These provide concrete examples of mechanisms for supporting application-level quality of service policies.

3.1 Server Architecture

The collaboratory server architecture was designed to explicitly manage the data throughput to each of its participants. Figure 4 describes the architecture for a single server. Generally, a collaboratory will be



Data Suppliers:
Supply 1
Supply 2
Server:
Client (Participant):
Applet 1
Applet 2
Discrete Delivery Policy Module
Degredation/Conversion Module
Intergration/Synchronization Module

Figure 3: Example collaboratory topology with dynamic data processing modules at suppliers, servers and clients. The architecture allows for the placement of data processing modules at any point in the distributed datapath. This example shows the path data from two suppliers to two client viewing applets.

supported by set of servers that share data between each other. When a supplier gives data to one server, the others eventually receive it, providing a broadcast mechanism from suppliers to all of the servers.

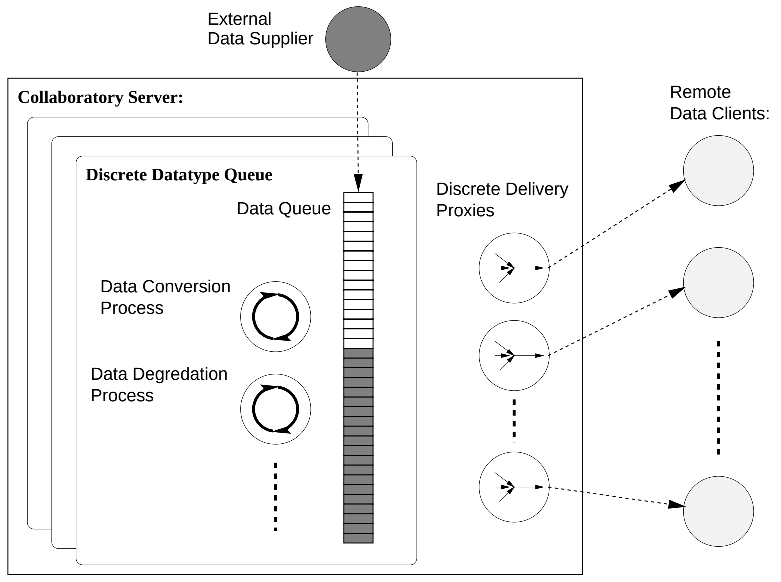


Figure 4: Collaboratory's Server Architecture

Each collaboratory server consists of several components that work together to provide flexible data delivery policies. As shown in Figure 4, external real-time data is stored for a bounded amount of time at the server in a data-specific queue. Each supplier datastream is stored in a separate server queue. Clients subscribe only to the data that they want. They do this by sending messages to a client proxy at the server. This proxy explicitly manages the bandwidth between the server and a single client. Notice that in our system, most of the data flows from the servers to the clients. If the clients were to additionally supply a significant amount of data without using some type of network reservation protocol, the utility of the system would degrade. In our scientific collaboratory, the only flow of data from the clients is a small amount of text. Since audio and video throughput is generally bounded–although bursty–the inclusion of these continuous data suppliers at the client hosts could be accommodated.

Associated with each of the server's data queues are sets of processes that can dynamically manipulate the real-time data streams. The server was designed so that these processes can be added or removed from the system while running. Several dynamic processes have been used during UARC experiments to dynamically convert and filter Radar data sets, degrade JPEG images on-the-fly, compress and uncompress data, etc. These processes act as internal suppliers that generate new data streams that are associated with their own server queues. The servers can specify which of these data streams are shared between each other, and which are only for distribution to clients. For example, in the UARC system, a server in greenland compresses the raw radar dataset for sending to the other servers; additionally, it converts the raw radar into a type easily manipulated by the client viewers. The former is shared between the servers, while the latter is not. This choice was made to minimize the amount of traffic over a low-bandwidth link between between greenland and the remainder of the Internet.

The current version of the server uses multiple unicast TCP connections to distribute data between servers and their clients. We have avoided using a reliable multicast protocol, such as SRM [3], due to the fine-grain control our system provides to the collaboratory's users. A reliable internet multicast protocol would introduce unwanted semantic dependencies between the data streams. For our target applications, the bandwidth near the servers is generally not the problem. The bandwidth constraints come from links close to the participants. If a link between two well-connected islands is the bottleneck, our system provides the solution by just making one the nodes in the second well-connected cloud a server that supports the rest of the island. One collaboratory server has supported up to 50 simultaneous clients during UARC experiments. Another reason for choosing a unicast distribution mechanism is the availability of Internet resource reservations [10]. There is currently no comparable multicast reservation scheme.

3.2 Discrete-Data Delivery Policies

The collaboratory's transport service provides a flexible and powerful delivery mechanism by explicitly managing the connections from the server to the clients. The collaboratory server provides control over its delivery policies by directly controlling the size and content of each client's outbound buffer. By multiplexing the subscribed streams, the client proxies can be used to prioritize, interleave, and discard discrete real-time data. We have constructed a flexible interface that allows a collaboratory's users to both: determine their current performance level with respect to other participants, and to gracefully degrade their quality of service from an application-level standpoint so that it best matches their individual network's service level. These quality of service parameters are taken directly from the user in the current system. When a user discovers that he is oversubscribed, he can use the interface to determine priorities among the subscribed streams, and assign drop policies for the individual streams. For example, for important streams, the user can specify a first-in-first-out (FIFO) delivery policy to assure that all of the data is transmitted. Alternatively, a *skipover* policy can be used to specify fine-grained drop orders. Current skipover policies consist of both a FIFO *threshold*, and a *drop distance* parameter. The threshold is used to allow for transient congestion; once this threshold of data has accumulated in the proxy, the drop distance parameter is used to determine which data are discarded, and which are delivered.

Another issue is the creation of underlying mechanisms to export the ability to allow applications to specify how different data streams–such as video, text, or interpreted code–interact. These mechanisms allow the collaboratory to specify how the different data streams are interrelated. For example, if the system elected to not send an image to a slower member, it might want to discard related material such as audio as well; or it might replace it with something like descriptive text.

3.3 On-Demand Degradation and Conversion

In addition to discrete delivery policies, the collaboratory's transport system provides for on-demand data degradation and conversion of discrete real-time data. In general, these mechanisms are used to convert one real-time discrete data stream into another. In order to support real-time collaboration between heterogenous clients, some mechanism for graceful data degradation must be made available to provide useful data to the slower participants. At the application level, we understand something about the semantics of the shared data. We can exploit this knowledge, and provide a graceful degradation of the data based upon these semantics. The collaboratory's transport service uses on-demand (lossless and lossy) compression on semantically typed data, tailoring contents to the specific requirements of the clients. Lossless compression techniques are used for those data that cannot be degraded, such as raw text, binary executables, and interpreted program text. Lossy compression techniques are applied to data that can suffer some loss of fidelity without losing their semantic information, examples of which are: still and moving images; audio; and higher level text like postscript or hypertext markup languages. We give the application layer control over this quality by providing an interface that adjusts the fidelity of the real-time data stream on a per-client basis. Our architecture allows stream-specific modules to be plugged into the supplier, server, or client middleware to achieve this functionality.

The uses of this data manipulation technology in the UARC system include: on-demand degradation of JPEG image streams and lossless compression and decompression of Radar data. Figure 5 represents the effects of on-demand image degradation on message latency for a real-world UARC datafeed. In this example, the UARC system is feeding real-time camera images of atmospheric phenomena from a server at Michigan to a participant in Alaska. By reducing the fidelity of the image to 35% (very readable), the mean latency for delivery was effectively halved. This is intuitive: we're sending less data, but we're still sending the same level of semantic information. The space scientist can adjust his setting to a useful level (down to 20% for these images) dynamically.

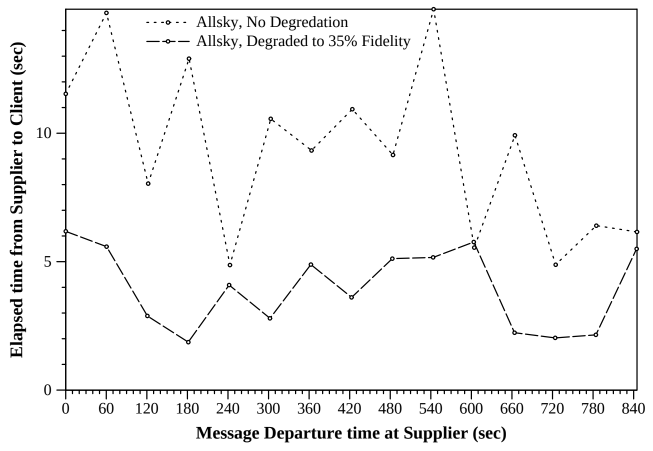


Figure 5: Experiment compares the end to end latencies for undegraded and degraded UARC Allsky camera images (degraded to 35Michi gan and the remote Client `UISR` in Alaska. Degradation of the data streams provide a 57% reduction in the mean latencies between these two experiments.

The UARC system also uses these plug-in modules for data conversion. During an atmospheric experiment, some clients needed raw Radar data, while others need more refined data. A natural solution was to share the raw Radar data between the servers, and allow those servers that needed to distribute refined data, to generate it on demand. This saved in inter-server bandwidth, which in the case of the UARC system was very important.

A problem with on-demand degradation is that it is not free. Figure 6 shows the cost of degrading a UARC allsky camera image. Clearly, the system should not give a knob to every participant. If the server needed to degrade a single image to forty or fifty different fidelity levels, it could take a significant amount of compute resources. A mechanism for bounding the number of fidelities should be used at the server to aggregate those that are close in size. A crude mechanism is currently in place for the UARC system's JPEG data fidelity: radio buttons that are set to low, medium, or high fidelity. Work is proceeding in providing a more sophisticated method for aggregating the various requests for fidelity at a finer granularity.

The architecture allows for these data manipulation modules to be placed at either the supplier, server, or client's middleware layer. This became increasingly important in a recent UARC experiment in which

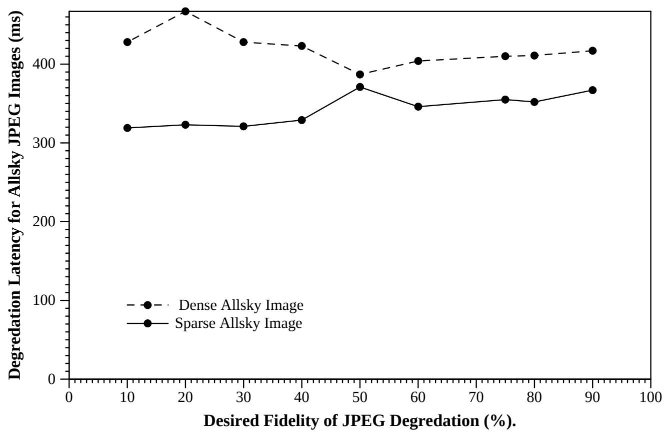


Figure 6: JPEG Degradation latency on an Ultrasparc 1 workstation for both a Dense and a Sparse Allsky image. Allsky camera images are monochromatic colorspace images that consist of the night sky over Greenland (Stars, Aurora, and Clouds). Their dimensions are always 388 by 467 pixels.

the clients were all Java-based viewers. It turned out that there was enough bandwidth to allow for almost all of the participants in the experiment to receive all of the data simultaneously. However, the compute resources for each Java client varied considerably. In fact, the slower Java clients could not paint the data fast enough to keep up with the incoming supply of data. In this case it made sense to place a module on the client where it could use application knowledge close to the user to determine which data were displayed and which were discarded.

Other work in on-demand data manipulation includes the work by Fox et.al. [4]. Their work focuses on data distillation as a means for supporting resource poor clients. They propose a general proxy architecture for the support of these clients. However, our work differs in that it allows for the dynamic placement of data manipulation modules at any point in the system, including suppliers and clients. Our system allows for the integration and allocation of multiple real-time data streams, whereas the work in [4] supports the effective delivery of a single type of data.

4 Conclusions and Future Work

By explicitly managing the scarce bandwidth of slow collaboratory participants at the application level, our architecture provides useful and effective support for real-time scientific collaboration. The combination of discrete real-time data delivery policies and on-demand data degradation can be used to provide a semantic throughput of multi-media real-time data. These application-level quality of service policies have been demonstrated in the UARC system.

The current version of our collaboratory architecture provides data delivery that centers on a single participant's priorities and policies. Future work will focus on providing delivery synchronization between several participants. Specifically, we want to be able to synchronize mixed media delivery with several constraints:

- Synchronization of a single stream by providing a guarantee that the data will be displayed by all participants within some user-specified delta.
- Inter-Stream synchronization on the same machine, so that different streams are represented at roughly the same point in time.

The synchronization of a single stream between a set of participants requires the use of a much more sophisticated group awareness protocol than the architecture currently supports. Providing real-time delivery guarantees over the Internet for a wide spectrum of clients with differing network and processor support is a challenging problem. There is a range of solutions, from slowing everyone down to the speed of the poorest participant to using discrete delivery policies to guarantee that those data that are delivered to slow members are delivered within a certain time window with the fastest members. Application-level quality of service policies will be instrumental in achieving this goal.

A natural progression would be to integrate the multistream synchronization policies with RTP's [7] synchronization of continuous media. There are significant problems here to be solved, especially with the intergration of single connection resource allocations (RSVP) and RTP streams. Load balancing also becomes a problem with CPU allocation to provide effective synchronization through on-demand degradation of the various streams.

We believe that high-level quality of service policies are an effective way to provide meaningful delivery to a broad spectrum of collaboratory participants. By giving the power over data fidelity and delivery to the application layer, an effective semantic throughput can be achieved that cannot be matched by network-level QoS architectures.

References

[1] Clauer, C.R., Kelly, J.D., Rosenberg, T.J., Rasmussen, C.E., Stauning, E., Friis-Christensen, E., Niciejewski, R.J., Killeen, T.L., Mende, S.B., Zambre, Y., Weymouth, T.E., Prakash, A., Olson, G.M., McDaniel, S.E., Finholt, T.A., and Atkins, D.E. "A New Project to Support Scientific Collaboration Electronically." In EOS Transactions on American Geophysical Union, June 28, 1994 (75).

[2] Fiuczynski, M., Bershad, B. "An Extensible Protocol Architecture for Application-Specific Networking." In Proceedings of the 1996 Winter USENIX Conference, San Diego, CA., pp. 55-64, January 1996

[3] Floyd, S., Jacobson, V., Liu, C., McCanne, S., and Zhang, L. "A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing." In Proceedings of ACM SigComm '95, pp. 342-356, October 1995.

[4] Fox, A., Gribble, S.D., Brewer, E.A., Amir, E. "Adapting to Network and Client Variability via On-Demand Dynamic Distillation." Proceedings of the ACM ASPLOS VII, Oct 1996.

[5] Hall, R.W., Mathur, A.G, Jahanian, F., Prakash, A., Rasmussen, C. "Corona: A Communication Service for Scalable, Reliable Group Collaboration Systems." Proceedings of the Sixth ACM Conf. on Computer Supported Cooperative Work (CSCW '96), November 1996.

[6] Malan, G.R., Jahanian, F., Rasmussen, C., Knoop, P. "Performance of a Distributed Object-Based Internet Collaboratory." University of Michigan EECS Department. CSE-TR-297-96, July 1996

[7] Schulzrinne, H., Casner, S., Frederick, R., Jacobson, V. "RTP: A Transport Protocol for Real-Time Applications." IETF RFC 1889. January 1996.

[8] Tennenhouse, D.L., Wetherall, D.J. "Towards an Active Network Architecture." Computer Communication Review, (26)2, April 1996.

[9] Wallach, D.A., Engler, D.R., Kaashoek, M.F. "ASHs: Application-Specific Handlers for High-Performance Messaging." In Proceedings of the ACM SigComm '96, pp. 40-52, August 1996.

[10] Zhang, L., Deering, S., Estrin, D., Shenker, S., and Zappala, D. "RSVP: A New Resource ReSerVation Protocol." IEEE Network, September 1993.