Audio/Video Messaging for Multiple Devices
(A Work-in-Progress Report)

Jill Boyce, Mauricio Cortes, and J. Robert Ensor

700 Mountain Avenue, Room 2D-522, Murray Hill, NJ 07974

Bell Laboratories, Lucent Technologies

{jillb,mcortes,jre}@dnrc.bell-labs.com

Abstract

This paper describes an infrastructure that supports data transforms and transmissions as part of multimedia communications. The infrastructure is made up of a specialized resource manager—called media flow manager—which creates and manages media flows. A media flow is the movement of a set of data—through a collection of media processing units—from a set of sources to a set of sinks. The paper describes the programming interface presented to applications by the media flow manager and the programming interface between the media flow manager and media processing units. It also describes how the media flow manager is being used to build an audio/video messaging application, and it describes the design and function of a particular media processing unit—a video transcoder—used by the messaging application.

1. INTRODUCTION

More and more devices can play a role in multimedia communication. For example, some workstations, PCs, personal digital assistants (PDAs), and telephones can display the contents of Web pages as well as send and receive real-time audio/video data streams. Different devices may use different networks to communicate with each other. For example, a person might compose a message on a PC and send it to a message server via the Internet, while the recipient might use a telephone to retrieve the message via the public switched telephone network (PSTN).

Each user of a communication application accesses information through one or more combinations of endpoint devices and network interfaces. Such a combination can send and receive data using only particular data formats and transport protocols. Hence, a user can access a communication application with a given device/network combination only if the application receives and sends data with appropriate formats and protocols. For example, there are message servers that can deliver messages as text via TCP/IP and also (after converting text to speech) as audio streams over PSTN circuit connections. Hence, users can retrieve their messages from these servers using both PC/Internet and telephone/PSTN combinations.

Our current work focuses on development of communication applications that are accessible from a variety of endpoint devices and network interfaces. To support many such applications, we have developed an infrastructure, middleware, to manage the transport and transformation of data. We have developed a specialized resource manager—called the *media flow manager*—which creates and manages media flows. We define a *media flow* to be the movement of a set of data—through a collection of media processing units—from a set of sources to a set of sinks. As illustrated in Figure 1, an application is built from collections of these processing units. The application uses the media flow manager to establish flows that incorporate specified media processing units (MPUs). The media processing units and the network connections are selected according to the data formats and transport protocols required by the application users, i.e. the device/network combinations through which people access the application.

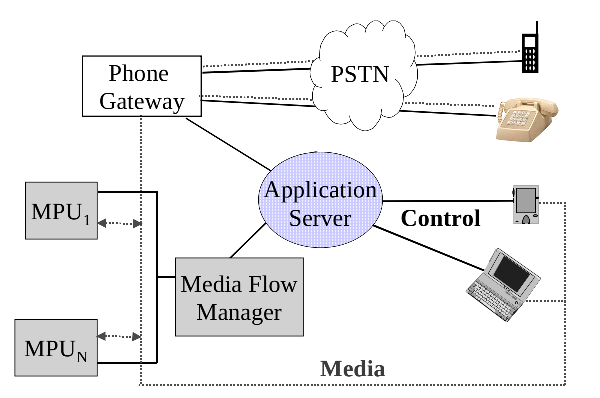

Figure 1.

In the next section of this paper, we survey some closely related work. We use this survey as an opportunity to list some of the fundamental design goals of our system. The paper continues by introducing the audio/video messaging application that we are currently building. It next describes media flow managers and their interactions with media processing units. The paper then briefly describes a particular processing unit—a video transcoder—that is part of our messaging application.

2. RELATED WORK

Most basically, the system we are now building is an addition to the body of work concerned with network based media processing efforts (e.g., [2], [8], [13], [14]). This body of work addresses the common goal of providing users with data that is appropriately formatted and delivered according to the requirements imposed by endpoint devices and their network connections. For example, Fox et.al. [8] discuss the need to add processing entities in the network called distiller to adapt multimedia traffic to available bandwidth and end-device capabilities. We want to make processing units available for multiple applications. We want these units to be distributed, available for targeted use in different geographical sites.

Other work is more closely related to ours. [1], [10], and [6] not only share common high-level goals, but also approach system design in similar ways. These systems differ from ours in the details, which we consider important.

Amini et.al.[1] describe a control server and graph managers that, together, are comparable to our media flow manager. Graphs and data exporters correspond to our flows and media processing units, respectively. However, while the graph manager invokes data explorers, control for explorers is passed up graphs from end users. In contrast, we want a more direct control path to appropriate media processing units. We feel that application servers can manage the control paths more efficiently. Furthermore, our approach is not limited to media streams, it can be used in asynchronous applications such as traditional messaging applications.

Jonas et.al. [10] discuss how network access points (NAPs) can search for processing and network resources throughout the network. Our media flow manager selects from a pool of registered resources. We feel that this is more efficient. Again, we do not limit our concern to media streams.

The Ninja Path [6] project sets up flows called paths. The processing nodes have to find each other, which we feel is less efficient. In Ninja, the source node needs to know capabilities of the sink. We feel that this limitation restricts the utility and re-use of servers. Chandrasekaran et.al. report that only linear paths are supported by Ninja Path. Our approach allows arbitrary graphs to be deployed dynamically on behalf of an application server.

3. EXPERIENCE

We have built an audio/video messaging application that allows users to send and receive audio and video messages through a variety of device and network combinations. The application was constructed by integrating a collection of specialized media processing units and an early version of the media flow manager. Figure 1 illustrates the overall structure of the system. The application server uses MPUs (managed by the media flow manager) to transcode and transport the audio and video portions of messages according to the capabilities of each user's device/network endpoint.

The primary characteristics of the application can be illustrated by the life of a typical message. A user may capture audio and video with any suitable device; we often use a Cassiopeia PDA, with its built-in microphone and an attached camera. The application client then uploads the message to an application server, which invokes media processing units to produce standard audio and video encodings of the message body. For example, audio/video messages created with the Cassiopeia are processed by units that transcode the video from a proprietary format to MPEG-1 or MJPEG. The message recipient may retrieve audio/video messages though a file download or media stream to a PC or PDA. Audio portions of messages are available to users through telephones.

4. ARCHITECTURAL OVERVIEW

Our architecture includes several components included in Figure 1 to support media flows:

- A **media processing unit (MPU)** receives, transforms, and/or delivers multimedia information. Examples of such units include transcoders, media streamers and text-to-speech transformers. They can exchange media with each other and with external entities, such as standard servers or clients.
- A **media flow manager (MFM)** is a resource manager that controls a set of MPUs. Application servers send requests to media flow managers for transformations on sets of multimedia data. These requests can vary from basic transcoding to complex cross-media transformations. This manager maintains a list of available MPU factories (see below), showing the MPUs they are able to instantiate. It uses this list to find and allocate MPUs to the requested tasks.
- An MPU **factory** (not shown in Figure1) creates one or more MPU instances locally. When a factory registers with a MFM, it lists the MPUs that it can create. Each factory monitors the resource usage of its host machine and reports these measurements to its MFM.
- An **application server** handles application-specific protocols. It is a client of a media flow manager and it specifies the functions that should be applied to sets of data. MFM acts on these requests by assigning

MPU resources, and establishing media flows. Once the resources have been allocated, application server controls these MPUs, under the supervision of MFM.

- A **client application** runs in user devices such as a PC, laptop, or PDA. It sends and/or receives data via media flows. Notice that a client can be a source and/or a sink of a media flow. The media flow manager includes MPUs within flows to receive, process and deliver the data according to the characteristics (*e.g*., bit rate, resolution) required by each sink.

Figure 1 shows several links connecting the different components. Solid lines represent control channels, while dotted lines represent media communication channels. The application server and the media flow manager exchange control messages using the *Media Flow Control Protocol (MFCP)*. This protocol allows application servers to request processing resources to a media flow manager. This protocol is described in Section 5.

Furthermore, MFM and factories exchange control commands that allow the former to control local resources as well as to coordinate the connections between any pair of MPUs. The MPU Control Protocol command set is described in Section 6.

In the following subsections, we describe the functionality of media processing units, MPU factories, and media flow managers.

4.1. Media Processing Unit

Media processing unit (MPU) are objects that can receive, transform, and deliver multimedia data. MPUs can process multimedia information coming from a file or some other MPU. The file can be either a local file or any web-accessible file. MPUs can also handle real-time streams and non real-time multimedia transfers coming from other MPUs. We currently support files, unreliable real-time streams, and reliable multimedia transfers.

Each MPU defines a list of methods. We plan to use IDL[12] syntax to advertise the MPU interface. (The Ninja project[6] uses XML to describe comparable interfaces.)

4.2. Factory

A factory must register with a MFM in order to let its MPUs be included in a media flow. Factories find a MFM at a well-known address and port. The registration process requires each factory to send to MFM the list of MPUs it can support. This information is used by MFM to select MPUs that meet application server requests.

Factories determine the network, CPU, and memory resources needed to process incoming data and to generate outgoing multimedia data. This information is reported to MFM. In the following subsection, we describe how MFM makes use of this information to manage network and computational resources.

4.3. Media Flow Manager

A media flow manager (MFM) stores the list of MPUs supported by each registered factory. This information allows MFM to control resources by creating, deleting, or modifying MPUs. For example, MFM can allocate CPU, memory, and network resources to an application server that needs to transcode an MPEG1 stream to a H.263 stream. The factory reports the amount of resources it spends in performing this transformation.

MFM represents a media flow as a graph with MPU as its nodes, and incoming or outgoing files, unreliable streams, or reliable multimedia transfers as its links. Application servers can request the creation of arbitrary graphs that will process multimedia data on its behalf.

In order to control the resources within a factory, MFM keeps track of its resource usage. As mentioned earlier, each factory reports back to the MFM the amount of resources needed by an MPU to process a request. This information is used by MFM to control network and computational resources. For example, MFM can perform load balancing if two or more factories offer the same MPU.

5. MEDIA FLOW CONTROL PROTOCOL

This protocol contains commands for the management of media processing units and their connections. These commands are exchange between server applications and MFM.

5.1. MPU-related commands

The first set of commands allows applications to request the creation, modification, and destruction of MPUs. MPUs are classified as adaptors, transcoders, or crossmedia. An adaptor unit generates the same media encoding as its input media parameter. For instance, an audio adaptor is a virtual unit that changes the sampling rate of an audio stream. A transcoders unit generates a different media encoding than its input parameter encoding. For example, a unit that transforms MPEG1 to H.263 movies is a video transcoder. A cross media units translates a input media parameter into a different media such as a text-to-speech engine.

The number of input and output media parameters can also characterize virtual units. For example, a splitter unit allows one input media parameter and multiple media output parameters. In contrast, a merger unit will accept two or more input media parameters but only one output media parameter. Finally, a bridge allows multiple input and output media parameters.

A. CreateMPU(unit, MPUname, parameters): An application uses this command to request the creation of a new multimedia processing unit. The application specifies a factory name, an MPU name, and a list of parameters. The string denoting the unit name should include the company and product name as well as the version (e.g. com.lucent.mpg2avi.ver3). The MPU name specified as the second parameter should be registered by at least one factory. The application can specify a list of name-value pairs to initialize the MPU.

When an MFM receives a createMPU command, it selects a factory implementing this MPU and requests the factory to instantiate it. Internally, MFM creates a data structure representing the newly created MPU to store control information. The factory will assign a unique identifier, which in turn is returned to the application.

B. Destroy(mpuId): When an application requests the destruction of an MPU, the MFM directs the MPU to disconnect all incoming and outgoing connections. Then, MFM asks the factory controlling the MPU to shut the MPU down, and the manager deletes the local description of the MPU.

C. Modify(mpuId, parameters): Applications can change the configuration of an MPU by specifying a list of name-value pairs. Some names are MPU-dependent, thus MFM forwards these pairs to the unit without further interpretation. In contrast, a few keywords, such as bitrate, stop, and pause, are first interpreted by MFM. The name-value pair is forwarded to the virtual unit, once it has being cleared by MFM. Notice that MFM uses these keywords to control and coordinate the virtual units affected by the command. Once the name-value pair is forwarded to an MPUt, the unit must interpret and send a positive or negative response back to MFM.

5.2. Media flow commands

The second set of commands allows applications to request the creation, modification, and destruction of media flows. A media flow is a graph structure that processes and generates one or more files, unreliable real-time streams, or reliable multimedia data.

These commands allow the creation of such graph by adding and deleting MPUs (nodes) and connecting and disconnecting these nodes. Notice that application servers can issue MPU commands even after adding a MPU to a media flow.

A. CreateFlow(): This command creates a new media flow. An MFM responds to this command by initializing a media flow description, assigning a new unique identifier to it. The MFM returns this number to the requesting application.

Internally, MFM creates a local object that maintains and controls the media flow state. This state is represented by a graph whose nodes are MPUs. Each link represents the communication between two MPUs. For each node, MFM keeps track of its creation time, the application that requested the MPU, resources, and the list of actual input and output multimedia streams.

B. Add(mfId, mpuId): Applications can request MFM to include an MPU in a media flow. An MPU can be included in one and only one flow. MFM returns a positive response, if the MPU already exists and it can be added to the specified media flow.

C. Delete(mfId,mpuId): MFM can remove a MPU from a media flow on behalf of an application. MFM directs this MPU and all its neighboring units to tear down their interconnections. Then, MFM updates the control state of all the involved MPUs. Notice that the MPU is not destroyed, allowing the application to add it to another media flow at a later time.

D. Connect(mfId, mpuId1, mpuId2): This command allows applications to hookup two MPU. These connections can be implemented through local files, reliable transmissions, or unreliable real-time streams. The media stream can be generated by some MPU within the same media flow or by an external source. If a real-time or reliable connection is requested by an application, MFM will coordinate the initial handshake between the two MPUs.

E. Disconnect(mfId, mpuId1, mpuId2): An application uses this command to tear down a connection between two MPUs. MFM requests both units to shutdown their connection. MFM updates the local state of both MPUs

5.3. Miscellaneous commands

The query command allows application servers to retrieve general information of factories and MPUs that are being control by MFM. On the other hand, the transaction command helps programmers developing application servers to specify atomicity constraints. For instance, an application can request to build a complex end-end media flow. If one MPU cannot be allocated, the media flow is discarded.

A. Query(type, id): An application can issue this command to retrieve the state of a media flow, a factory, or a MPU. Applications can retrieve the graph associated with each media flow, ask a factory for its resource usage, or get the state of a particular MPU.

B. Transaction: A simple transaction mechanism has been included in MFCP. This mechanism allows an application to group basic MPU and media flow commands into atomic units. Nested transactions are not permitted in this environment. The transaction mechanism introduces two new commands, **trans_begin** and **trans_end**. Any basic command between these two transaction commands is considered a transaction step. MFM assures that the transaction is executed atomically, i.e. all transaction steps are executed or no step is executed. Since every basic command returns an error status, MFM must rollback all the previous steps when a basic command fails within a transaction.

We believe that this set of commands is sufficient for applications to manage media flows. In Section 7, we will discuss some issues that we have encountered in controlling factories and MPUs.

6. MPU CONTROL PROTOCOL

This control protocol enables factories and media flow managers to exchange control information. The interaction between a factory and a MFM starts with a registration command. Once the factory has successfully registered, it informs MFM of its own capabilities by declaring the MPUs it can create. Thereafter, MFM can request the factory to create and destroy MPU instances. Factories will report resource usage information back to MFM, enabling the manager to control the network and computational resources.

A. Register(unit): This command allows a factory to register its name with an MFM. Recall that the name should include the company and product name as well as the version (e.g. com.lucent.mpg2avi.ver3). MFM responds with a unique identifier. This identifier will be used throughout the interaction between the two entities. It is important to point out that one or more factory instances can run simultaneously even with the same unit name. MFM can distinguish each instance by this unique identifier.

B. Define(factoryid, idl-method): A factory can declare one or more MPUs whenever it is ready to support them. This allows each factory to make MPUs available at any time. For example, factories can make new MPUs or new versions of an MPU available after months of running the factory process.

C. Invoke(factoryid, mpu-name): MFM can request a factory to start an MPU on behalf of an application server. If the factory can create a new MPU instance, a unique identifier within this factory representing the newly created MPU is returned to MFM. This identifier will be used in further interactions between MFM and the factory.

D. SendCommand(factoryid, mpuid, name_value): When an application server issues a modify command, MFM translates it into a SendCommand message, and sends it to the appropriate factory. Recall that MFM interprets the name-value pair if it contains a keyword. Otherwise, MFM forwards the name-value pair to the corresponding factory.

E. OpenStream(factoryid, mpuid, url, type): The Connect command issued by an application server gets translated into an OpenStream command by MFM. This command is sent to the corresponding factory, directing an MPU to open a multimedia stream. The third parameter specifies if the stream is an input or output stream. Depending on its local resources, the factory will return a positive or negative response to MFM. If the factory can allocate the requested resources to handle this stream, it will issue a ResourceUsage command to inform MFM of the new values.

F. CloseStream(factoryid, mpuid, streamid): This command closes an input or output stream. The factory will issue a ResourceUsage command modifying the amount of resources used by this unit.

G. Revoke(factoryid,mpuid): MFM sends a revoke command whenever an application servers sends a destroy command. MFM examines if the MPU is still active before sending this command.

H. ResourceUsage(factoryid, mpuid, projected[]): This command allows factories to report back to MFM the projected CPU, network bandwidth, and memory utilization. MFM will use this information to manage resources across the registered factories. Notice that factories rather than MPU report resource usage back to MFM. In this way, factories can summarize the report for all the MPUs running under its control. In any case, this approach has the potential to overflow media flow managers with resource usage commands. We are considering asking factory programmers to restrict the number of resource reports per minute.

We believe this is a minimal command set to control a significant number of MPUs and their resources.

7. SOME ISSUES

Our goal is to build a media flow system with the following characteristics: extensibility, scalability, and resource monitoring. In this section we examine these issues in some detail.

Our team is addressing other important issues, such as startup congestion control, fault tolerance, and security. For example, the initial factory registration can become a bottleneck since each factory needs to register and declare a list of supported MPUs with MFM. Since a significant amount of data could be exchanged during these registration period, we plan to use a version of the bakery algorithm to reduce the impact of the startup congestion problem.

7.1. Extensibility

Each new media processing function extends the system's capabilities. A new function allows MFM to accept new requests from user applications and to control these new MPUs. New MPUs can be added to an existing factory or can be part of a newly created factory. In either case, the factory needs to register the new MPU with its corresponding MFM.

It is important to point out that factories can report the availability of an MPU at any time. This mechanism encourages the development of factories that can load new MPU definitions at runtime. Factories can declare new MPUs well after its initial registration. This feature allows programmers to build non-interruptible media flow services.

7.2. Scalability

We have designed MFM to handle thousands of requests from simultaneous users. Note that multimedia processing is only done by MPUs. Each factory runs in their own process space and even on different machines.

The system can accommodate a wide range of configurations, *e.g.*, a single MPU, multiple MPUs running in one machine, and a large number of MPUs—each running on its own machine. The system configuration will depend on the frequency (or distribution) and available resources for an MPU.

The performance of each MPU depends on its task (*e.g.*, transcoding, mixing) and its actual implementation. A MPU can be further characterized by the computational and network resources needed during each function invocation. Each MPU provides this resource usage information to its factory. In this way, factories can collect this information and send a report to MFM. In turn, MFM can balance the load among MPUs using this resource information.

Scalability could be compromised by network traffic between MPUs and their corresponding external sources or sinks. Notice that MFM can only control the network elements within a media flow. Since MFM cannot control the external network, network congestion can still be present.

7.3. Resource Monitoring

We have described how applications can request the creation of MPUs and media flows. However, MFMs and factories must exchange control messages to let media flow components accommodate unexpected changes in network or computational usage. For example, MFM should be able to inform one or more factories to change its bit rate because of internal or external network congestion. Resource monitors are needed to keep MFM inform of actual resource usage information. These resource monitors can trigger events in MFM to send a SendCommand to one or more MPUs.

8. EXAMPLE - VIDEO TRANSCODING

In this section, we present device, bandwidth, and network error issues associated with an application's request for the appropriate transcoding MPUs.

8.1. Device and network bandwidth

In [11], images were transcoded to different bit rates, to adapt to available bandwidth. Image quality was adjusted, but image pixel dimensions remained fixed, and a single image compression standard was used. Video quality was additionally adapted by adjusting the video frame rate. In [8], transcoding between video compression standards was also done, based on network variation, and device hardware and software variation.

Similarly, the audio/video messaging application described in Section 2 transcodes video content between video compression standards, but using different compression standards than those used in [8], including MPEG-1, Motion-JPEG, H.263, and a proprietary Casio format. Multiple MPUs are used to perform the video standard transcoding function. These transcoders also perform additional transformations to accommodate the different pixel dimensions, frame rate, and bandwidth of different endpoint devices. For example, the PDA has a small image/video display screen, which may not be capable of displaying all of the pixels than are present in a video message created on a PC. Similarly, the PDA may not be capable of decoding a given video compression standard at the bandwidth of the original video message.

8.2. Network error characteristic

In order to achieve our goal of supporting a variety of network connections, we must consider not only the differing available bandwidths of different networks, but also the differing channel loss characteristics, which has not been done in the previous works [8], [11]. For example, PSTN, IP, and wireless networks all have different loss characteristics. Better quality video can be received in the presence of channel loss by targeting the specific channel loss characteristics. Not only is a video source coding transcoder MPU necessary, but also a transcoder for channel coding, or joint source/channel coding. We are building MPUs with source/channel coders that are targeted to a user's particular wired IP or wireless IP channel.

The UDP protocol, which is commonly used for multi-media applications over IP, specifies that if any portion of a transmitted packet is not properly received, the entire packet is discarded. This works well for wired IP networks, where bit errors are uncommon, and loss occurs in general because entire packets are lost in router queues. In wireless networks, however, bit errors are common. When an IP packet is sent over a wireless network, in general a relatively large IP packet (< 1500 bytes) is divided into several small wireless frames of ~40 bytes, depending on what wireless standard is used.

In both types of lossy networks, channel coding can be used to improve received quality, by adding overhead prior to transmission. For wireless networks, convolutional coding is used for protection against bit errors. For IP networks, block coding (i.e. Reed Solomon) can be applied across packets for protection against packet loss [5]. Channel coding adds significant overhead to the transmission thereby reducing the amount of information that can be transmitted over a limited bandwidth network. For example, for rate 1/3 convolutional coding, for each 1 byte of information, 3 bytes are transmitted.

Combined source/channel coding can be used to improve performance if the particular error characteristics of the channel are known and targeted. A technique known as Unequal Error Protection (UEP) can be used with scalable video coding for transmission of video over unreliable networks. Several of the popular video coding standards, such as H.263++, MPEG-2, and MPEG-4, provide tools for scalable video coding. In scalable video coding, the video encoder creates a base layer and one or more enhancement layers. The base layer contains the most important video information, and video can be decoded using only the base layer. When enhancement layers are also decoded, the video quality is improved [4].

For networks with prioritized transmission, such as ATM, scalable video coding can be used to improve performance in the presence of channel errors, by transmitting the base layer at a higher priority than the enhancement layer [3].

Because the standard wireless networking protocols [9] do not support prioritized transmission, Unequal Error Protection with scalable video coding is accomplished by applying higher levels of channel coding to the base than to the enhancement layer(s), so that the base layer is more likely to be received by the decoder. Applying higher overhead channel coding rates only to the most important portions of the video allows more efficient use of the limited wireless bandwidth.

We developed a video transcoder MPU that converts input video to an H.263++ SNR Scalable bitstream using UEP, for transmission over a UMTS 3rd Generation Wireless system. Compared to separating the source and channel coding, considerable bandwidth savings can be achieved.

A single layer H.263 video message using rate 1/3 convolutional coding was compared with a message that was transcoded by the MPU into two layers, with rate 1/3 coding applied to the base layer and rate 1/2 coding applied to the enhancement layer. As shown in the table, a 25% saving in bandwidth was achieved by using the video transcoding MPU, over the case when no source/channel transcoding were done. The difference in received video quality was small, for several channel model simulations, including the UMTS physical layer standard's Vehicular A and Indoor-to-outdoor/Pedestrian models [7].

We are now working to determine the video coding standards, tools, and parameter choices that are best suited for the source/channel transcoders for IP networks and various wireless networks, including CDMA, GPRS and UMTS.

Case	Source coding rate		Convolutional channel coding rate		Total rate	Savings
	base	enhan	base	enhan		
1 layer	64 kbps	-	1/3	-	192 kbps	-
2 layer	16 kbps	48 kbps	1/3	1/2	144 kbps	25%

9. FUTURE DIRECTIONS

This paper has provided a brief description of our current research—development of communication applications that can be used from a large number of device/network combinations. To support such applications, we are building a specialized resource manager, the media flow manager, that allocates resources to gather, process, and distribute media. We are also building new, specialized media processing units.

Our current experiments will help us measure the quality of two standard interfaces—the media flow manager's interface, MFCP, and MPU control protocol that includes function descriptions. Future work will allow us to test the performance of media flows within additional network environments. We will also measure the performance of new media transcoders, those that are sensitive to the performance characteristics of the networks.

References

[1] L. Amini, J. Lepre, M. Kienzle, "Distributed Stream Control for Self-Managing Media Processing Graphs," pp. 99-102, ACM Multimedia '99, Orlando FL., October 1999.

[2] E. Amir, S. McCanne, R.Katz, "An Active Service Framework and its Application to Real-time Multimedia Transcoding," pp. 178-189, Sigcomm '98, Vancouver.

[3] R. Aravind, M. Civanlar, A. Reibman, "Packet Loss Resilience of MPEG-2 Scalable Video Coding Algorithms," IEEE Transactions on Circuits and Systems for Video Technology, Vol. 6, No. 5, October 1996.

[4] J. Boyce, "Packet Loss Resilient Transmission of MPEG Video over the Internet," Signal Processing: Image Communication, Vol. 15/1-2, September 1999.

[5] D. Budge, R. McKenzie, W. Mills, and P. Long, "Media-independent Error Correction using RTP," Internet Engineering Task Force Internet Draft, May 1997.

[6] S. Chandrasekaran, S. Madden, M. Ionescu, "Ninja Paths: An Architecture for Composing Services Over Wide Area Networks", http://ninja.cs.berkeley.edu/dist/ papers/path.ps.gz, Berkeley, CA.

[7] ETSI Umts Universal Mobile Telecommunications System: ES 201 385 V1.1.1 (1999-01).

[8] A. Fox, S. Gribble, E. Brewer, E. Amir, "Adapting to Network and Client Variability via On-Demand Dynamic Distillation", pp. 160-170, ACM ASPLOS '96, MA, October 1996.

[9] T. Hodes, R. Katz, E. Servan-Schreiber, L. Rowe, "Compasable Ad-hoc Mobile Services for Universal Interaction", ACM, MOBICOM '97, Budapest, Hungary, 1997.

[10] K. Jonas, M. Kretschmer, J. Modeker, "Get a KISS-Communication Infrastructure for Steaming Services in a Heterogeneous Environment," pp. 401-410, ACM Multimedia '98, Bristol UK.

[11] B. Noble, M. Satyanarayanan, D., Narayanan, J. Tilton, J. Flinn, K. Walker, "Agile Application-Aware Adaptation for Mobility," SOSP, Saint-Malo, France, October 1997.

[12] OMG, "CORBA/IIOP 2.3.1 Specification", October 1999.

[13] D. Tennenhouse and D. Wetherall, "Towards an Active Network Architecture," *Computer Communication Review* 26,2 (April 1996), 5-18.

[14] S. Wray, T. Glauert, A. Hopper, "The Medusa Applications Environment", International Conference on Multimedia Computing and Systems, Boston MA, May 1994.