Supporting Collaborative Applications Using Dynamically Adaptive Multicast Trees

by

XINLIANG ZHOU

A Thesis
Submitted to the Faculty of Graduate Studies
in Partial Fulfilment of the Requirements
for the Degree of

MASTER OF SCIENCE

Department of Computer Science
University of Manitoba
Winnipeg, Manitoba

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION PAGE

SUPPORTING COLLABORATIVE APPLICATIONS USING DYNAMICALLY ADAPTIVE MULTICAST TREES

BY

XINLIANG ZHOU

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University

of Manitoba in partial fulfillment of the requirements of the degree

of

Master of Science

XINLIANG ZHOU © 2003

Abstract

This thesis presents "active dynamically adaptive multicast", an active network based multicast multicast service for real-timie multimedia transmission. Traditional network infrastructure has a fixed layered architecture that restricts the approaches that may be taken to supporting data-intensive multimeedia transmissions. In particular, the current Internet makes it very difficult to optimize transmission based on "application layer" characteristics without the deployment of specialized protocols which requires costly (and therefore undesirable) network infrastructure upgrades.

While multicasting (sending a single message or media stream to multiple recipients rather than just one) does permit greater efficiency than using unicasting, it also introduces a number of problems. Foremost among these is the fact that, by definition, the data sent to all multicast recipients is identical. This ignores the fact that different receivers have different interests and that the network infrastructure from the sender to the various receivers may have different capabilities.

This problem can be addressed by allowing the multicast streams to "adapt" dynamically to the needs of their receivers and to the capabilities of the netwrok segments they will travel over. Thus, network bandwidth can be saved (or lower available bandwidth tolerated) if multimedia streams are modified during transmission through specific branches of the multicast transmission tree to reflect both the requirements of the receivers on that branch and the capabilities of the underlying network links that constitute the branch.

The thesis presents an Active Networks based prototype implementation of such a dynamically adaptive multicast protocol which reacts to user-interface hits (e.g. iconifying a window displaying a particular media stream) by sending "requirements" messages in the form of active network capsules back up the multicast tree. The received information is then combined with other such received information and is used to determine the necessary adaptation which is then performed, all without having to dsitribute a new protocol supporting such adaptation.

Acknowledgements

I would like to thank Dr. Peter Graham and Dr. Rasit Eskicioglu for being my advisors. They have always been enlightening, patient and supportive throughout the project, and I owe a great lot of gratitude to them for having lead me this far.

I dedicate this thesis to my parents and my dear wife, for their unconditional love and support.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

The advances in Internet technologies have led to the emergence of collaborative applications, such as tele-conferencing, and tele-medicine, which use the Internet as the communication infrastructure to achieve group-based cooperation. Unlike traditional applications like email or file transfer, collaborative applications utilize various kinds of real-time media (e.g., audio, video, graphics and plain text) between participants to communicate timely and efficiently. The key technical foundations of these applications are multicasting and real-time services, in particular, real-time transport protocol.

A collaborative application naturally has multiple participants. Multicasting provides a more efficient mechanism to replicate messages as they travel along the network, and saves the sender from replicating the messages itself. Instead, the messages are duplicated as needed within the network itself during message routing. Such "multicast routing" must construct a tree of routers that is rooted from the source node and contains all multicast destinations. This tree is known as a "multicast tree". An optimal multicast tree is a tree that spans the multicast group members and has minimum cost, where the cost might be evaluated using the number of hops, the transmission delay, or some other metric(s). Optimal algorithms for constructing multicast trees efficiently are still an open research topic in the network community, although effective techniques for constructing good multicast trees are known. Until now, most work on multicasting has been based on legacy network infrastructure in which the functionality of the network itself is restricted. This prevents multicast trees from adapting to the changing needs of applications.

The real-time transmission is the crux of collaboration of continuous media [AGH90] communication. Continuous media involves those data generated at a given (not necessarily fixed) rate, independent of the network load, and which should be delivered to the user with the same rate as they were generated. Video, including variable-rate video and audio are examples of continuous media. Although the TCP/IP protocols are successful in most Internet applications, such reliable transport protocols are not appropriate for real-time transmission. Besides the lack of multicast support, the nature of reliable transmission requires that there must not be any restricting delay constraints set between senders and receivers. Also the "slow start" congestion control mechanism used by TCP could interfere with the playout rate of audio and video streams. To facilitate real-time transmission, the IETF developed the Real-time Transport Protocol (RTP) [HSJ98], which provides end-to-end network transport functions suitable for applications transmitting real-time data, such as audio, video or simulation data, over multicast or unicast network services. RTP primarily defines the packet formats and data transmission issues, and also has its own control protocol, called the Real-Time Control Protocol (RTCP) [HSJ98], to be used in conjunction with RTP. The primary function of RTCP is to provide feedback on the quality of the data delivery. The RTCP control packets are periodically transmitted between participants, so that applications can use the feedback information to monitor and control the quality of service (QoS).

As multicasting and RTP set the cornerstone for real-time multimedia and collaborative applications, research has been prosperous in these areas. Implemented at Lawrence Berkeley Laboratory (LBL), vic [MJ95b] and vat [MJ95a] are the most widely used RTP-based video and audio conferencing tools used over the MBONE [Eri94]. Also with the help of RTCP's QoS feedback mechanism, end-to-end adaptive QoS control was proposed [BDS96]. Since RTP/RTCP are actually end-to-end protocols residing at the application layer, adaptation can only be carried out at the sender side. Thus, without support from the underlying network infrastructure, the adaptation could be subject to scalability problems in multicast distribution: when there are multiple receivers, the sender could either adjust bandwidth according to the reports of the worst-positioned receiver, leaving other receivers to share the same poor quality, or do the adjustment in favor of a certain fraction of receivers and leave the remainder in a congested state.

This thesis proposes using active network technology [Wet99] combined with multicasting and real-time transport services to provide better adaptations for collaborative applications. Active networks advocate a flexible and extensible network infrastructure by introducing programmability into the network layer. Such a novel architecture allows user-defined network services to be injected into the network, so that the network layer can provide application-specific adaptations based on the awareness of changing user requirements. This research addresses better adaptations for collaborative applications through two mechanisms: (1) The design of the active network protocols that allow multicast trees to adapt to the needs of their users by dynamically modifying the data streams being sent through the various branches of the tree in response to user activity; (2) Provision for in-network QoS control of real-time media transmission realized with active-network-based network layer services.

To solve the problem involving real-time data-intensive data transmission, most existing approaches focus on either providing new network layer services or having the application handle the transmission issues by themselves. With active network bridging the network and application layer, the active dynamically adaptive multicast service described in this thesis comes up with solution by incorporating the multicast network service with user-requirement-driven adaptation. It could benefit collaborative applications by providing a finer-grain and prompt adaption scheme which will lead to a better bandwidth usage.

This thesis is organized as follows: Chapter 2 summarizes other works related to this thesis. Chapter 3 describes the problem and the motivation behind the presented solution. Chapter 4 presents the solution strategy towards the problem, and the implementation and experiment details are covered in Chapter 5. Chapter 6 concludes with a summary and discusses future directions.

Chapter 2

Related Work

This chapter surveys the concepts and research related to this thesis. Multimedia networking is reviewed in section 2.1 focusing on transmission issues like multicasting and real-time transport. Section 2.2 introduces an emerging multimedia toolkit for Java, the Java Multimedia Framework (JMF), which allows the Java community to manipulate, process, and transmit continuous media. Section 2.3 reviews active networks technology generally.

2.1 Multimedia Networking

After years of evolution and huge increases in popularity, the Internet has expanded dramatically in function, from a facility used primarily for file transfer and email exchange mostly by academic researchers to a promising infrastructure for commercial and other multimedia services. Though literally, multimedia just means two or more types of media, here it refers to the combination of two or more continuous media sources. Continuous media services generate media streams at a given (not necessarily fixed) rate, independent of the network load, and are expected to be delivered to the user at the same rate as they were generated. In practice, most continuous media are either audio or video streams. The technical foundations in transmission for multimedia applications are multicasting and real-time media services.

Most multimedia applications using continuous media (e.g. collaborative applications like tele-conferencing, electronic whiteboard, and video on demand) are, by nature, multi-participant applications. It is thus potentially more efficient for the network to provide a multicast service, where replication of messages can be done at optimal points in the network rather than to have the sender transmit multiple copies of the media data individually to each receiver.

For continuous media, timely delivery is often more important than exact data accuracy, since a small fraction of data loss/error won't be as noticeable as the jitter often caused by delayed delivery. Thus, multimedia communication requires real-time delivery (i.e. an upper bound on the transmission delay has to be imposed between the sender and receiver). An Internet protocol for transporting data including audio and video in real-time is the Real-time Transport Protocol (RTP).

2.1.1 Multicasting

Multicasting is a form of group communication in which messages from a single sender are transmitted simultaneously to many receivers. In multicasting, a group is defined to be a set of receivers interested in the same message(s). Multicast differs from broadcasting. Broadcasting floods traffic all through the network unselectively while multicasting reduces the load imposed on the network by only sending traffic to the hosts in a multicast group. An efficient multicast protocol should provide data delivery to a group of hosts with lower network and host overhead than broadcasting to all hosts or unicasting to each host in the group individually [DC90].

In multicasting, a group is assigned a logical group ID that uniquely identifies the corresponding collection of hosts. The logical group ID is then mapped to a physical multicast address for a certain network. Class D addresses, ranging from 224.0.0.0 to 239.255.255.255, are the multicast addresses in an IPv4 network. The mapping takes the lower 28 bits of the class D address to specify the group ID while the whole 32-bit address forms the multicast address.

One of the important functions of a multicast protocol is group management which takes care of group membership and each member's joining and leaving activities. A multicast router learns about the group membership of the hosts in the directly attached subnet by exchanging Internet Group Management Protocol (IGMP) [ABG97a] messages. Each router then propagates that information using the multicast routing protocol so that each multicast router knows what to do if it receives a packet destined to a given multicast address [Ste98].

Multicasting can be easily implemented on some Local Area Networks (LANs), such as Ethernet, due to the availability of the broadcast delivery and, for the Ethernet and other LANs conforming to IEEE 802 standards, a large space of multicast addresses [DC90]. However, for most packet-switched, Wide Area Networks (WANs) without broadcasting support, special purpose multicast routing algorithms and protocols are needed to deliver multicast messages to group members which are sparsely distributed. The key to multicast routing in a WAN environment is to construct the message delivery path and then forward the multicast messages accordingly as they pass through the routers. Such a delivery path is called a multicast tree because it is always in the topology of a tree that roots from a certain node and spans all group members. Source based trees [WPD88] and shared distribution trees [BFC93] are the two major types of multicast trees employed by most multicast routing protocols.

2.1.1.1 Source Based Multicast Trees

A source-based tree is a minimal multicast tree spanning from the sender (i.e. the "source") to all receivers in the group. The cost of the tree may be evaluated using path length, transmission delay, or some other metric. Thus a source tree is also known as a shortest path tree (SPT). Though a source tree provides the most efficient delivery path for multicast forwarding, a source tree does not scale well to large networks. To refer to a source tree, both the address of the sender and the multicast group address are required. So for each source-group pair, a source tree must be constructed and maintained. For a large network, with large numbers of senders and groups, the storage requirement for the routers to hold the forwarding pointers could be overwhelming.

Among existing multicast protocols, all dense-mode multicast protocols[1] use source trees as their multicast trees despite using different construction algorithms. Typical protocols in this category are: the Distance Vector Multicast Routing Protocol (DVMRP) [WPD88], Multicast OSPF (MOSPF) [Moy93], and Protocol Independent Multicasting-Dense Mode (PIM-DM) [DEF+95].

DVMRP builds source trees by flooding and pruning Truncated Broadcast Trees (TBT) [WPD88]. A TBT for a certain subnet represents a shortest path tree rooted at that subnet and spanning all other routers in the network. A TBT is

[1]In dense mode, the members of multicast groups are densely distributed across a network, and are usually interconnected via high bandwidth links.

constructed by periodic routing updates between DVMRP enabled routers based on the Bellman-Ford distance vector algorithm [Bel58]. Traffic is initially flooded down TBTs. Routers are then pruned off the TBT if they have no group members.

MOSPF is built on top of the Open Shortest Path First (OSPF) Version 2 routing protocol (RFC 1583). Each MOSPF router maintains a complete network view by gathering the link-state information flooded between routers. Each router builds source trees by using this information and the group membership information obtained using IGMP. As the name suggests, these trees are the shortest-path trees constructed on demand for each (source, group) pair.

PIM was designed to provide flexibility, scalability and efficiency for both densely populated networks and large heterogeneous inter-networks. It is protocol-independent because it does not depend on a specific underlying unicast routing protocol. For this reason, PIM has been implemented widely in production environments.

PIM-DM is based on the assumption that when a source starts sending, all downstream systems want to receive the multicast messages. Thus multicast messages are initially flooded (i.e. broadcasted) to the entire network. For those who do not want to receive the multicast messages, prune messages are sent upstream from them to remove themselves from the distribution tree. Such intrinsics of PIM-DM make it suitable for densely populated environments.

2.1.1.2 Shared Distribution Multicast Trees

In the case of shared distribution trees, all traffic to the receivers of a certain group are forwarded down a shared tree according to just the group address that they are destined for. As the name indicates, such distribution trees are shared by all senders to a certain group. Thus, to refer to a shared tree, only the group address is needed, regardless of the source address. To construct a shared tree, a "center" node has to be designated as the root of the tree, down which all multicast traffic from various senders will flow to the receivers. The name of the center point varies in different protocols. For example, it is called the "Core" in Core Based Trees (CBT) [BFC93] and "Rendezvous Point" (RP) in Protocol Independent Multicasting-Sparse Mode (PIM-SM) [DEF$^+$95].

When a receiver wishes to join a multicast group, a join message is sent to the root. When a sender starts sending, traffic must somehow first be sent to the root of the tree, and from there, it can then be propagated down the tree to all receivers. Shared distribution trees mainly aim to solve the scalability problem of dense-mode multicast protocols. With the avoidance of flooding, multicast trees can be constructed without overly burdening the network, even if there are just a few sparsely scattered receivers. Shared trees may provide a sub-optimal delivery path, but require significantly less router memory.

Protocols in this category mainly differ from one another in the mapping from a group address to a center address, the selection of the center node, and the tree construction algorithm used. CBT and PIM-SM are two typical center-based multicast protocols.

Proposed as a solution to scaling problems with sender-based multicast trees, CBT was the first multicast protocol to use a shared tree. In CBT, one router is designated as the core, around which the multicast tree is constructed. For each multicast group there is a single multicast tree rooted from the core for that group which is shared by all senders. When a receiver joins a group, it sends IGMP messages to its last-hop CBT router (i.e. routers directly connected to receivers) and the latter sends a Join message to the core router. As the Join message is forwarded all the way to the core, or until it reaches an on-tree router, new branches of the tree are formed and eventually spliced onto the existing tree. Once the tree is built, the core is no longer important for forwarding.

The shared tree constructed in CBT is bi-directional, which means traffic moves both up and down the tree. CBT is considered to be the simplest shared tree because of its simple forwarding rule. When a packet reaches a CBT router, it is forwarded out on all interfaces that are part of the tree, except the one it arrived on. CBT provides better scalability to support sparse distribution of multicast receivers. By using a shared tree, CBT reduces the amount of multicast state information that must be stored in routers, and traffic is aggregated onto a smaller subset of links.

Unlike PIM-DM, PIM-SM uses a shared multicast tree built by having all receivers send explicit Join messages through their last-hop PIM router (i.e. routers directly connected to members) to a designated root node called the "Rendezvous Point" (RP). The last-hop router should be properly configured with the information of the RP. If it does not have RP information, it is considered an error. The way that the tree is built is similar to CBT: as the Join message travels hop-by-hop from the receiver's directly-connected router to the RP, a branch of the tree extending the whole path is added. Unlike CBT, the tree in PIM-SM is unidirectional, so traffic only flows down the tree from the RP to the receivers, but not back up.

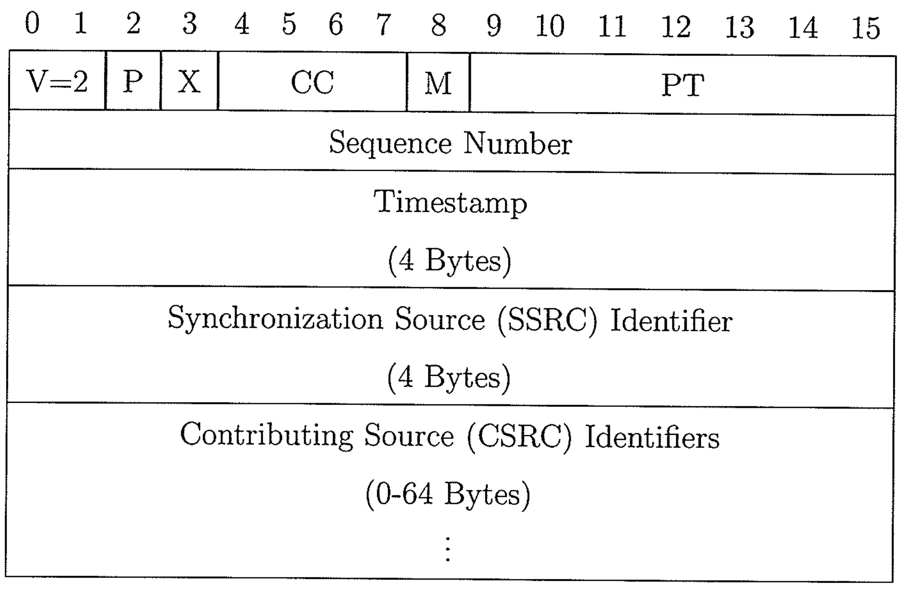
When a sender transmits multicast data to a group, its first-hop router (i.e. the router directly connected to the sender) maps the group address to the address of the RP, encapsulates the data in a Register message, and unicasts the message to the RP. Once the RP receives the Register message, it first de-encapsulates the multicast data out of the Register message and then forwards it down the tree to the receivers. The RP also sends another Join message back to the sender's sub-network to set up a packet delivery path from the sender to the RP. From that point on, multicast traffic is sent natively (i.e. without encapsulation) through the path to the RP, and from there, down the shared tree to all receivers.

PIM-SM also has the capability for last-hop routers to switch to the Shortest Path Tree and bypass the RP if the traffic rate is above a set threshold called the SPT-Threshold. In that case, a join message is sent by the last-hop routers hop-by-hop up to the first-hop router to construct an SPT between the first-hop router and the last-hop router. Once the SPT is built and data start to arrive along it, a prune message is sent up the original shared tree to prune off the branch going through the RP.

2.1.2 Real-Time Transport Protocol (RTP)

RTP [HSJ98] is the Real-time Transport Protocol which provides end-to-end delivery services for data with real-time characteristics, such as interactive audio and video. Despite the misleading name, RTP actually runs at the application layer on end systems (i.e. the applications that generates the content to be sent in RTP packets and/or consumes the content of received RTP packets [HSJ98]). RTP sets no constraints on the capability of lower layer protocols, except that they must support framing. Currently, RTP is mostly commonly used together with UDP on IP networks or with AAL5 on ATM networks.

RTP is actually composed of two protocols: the Real-time Transport Protocol is the protocol used to carry the data, and the Real-Time Control Protocol (RTCP) is used to monitor the quality of service and to convey information about the participants



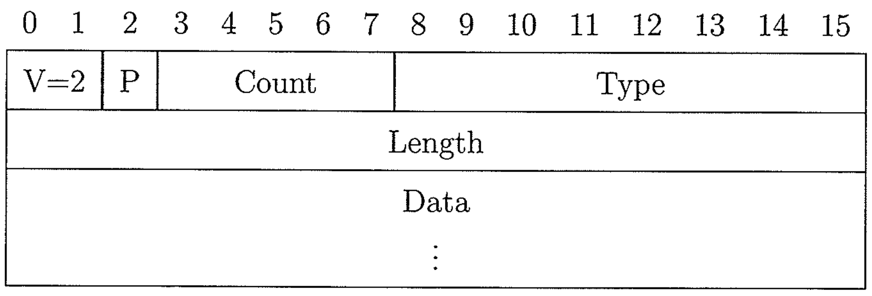
V: Version P: Padding X: Extension CC: Contributor Count

M: Marker PT: Payload Type

Figure 2.1: RTP Header Format

in an on-going session. RTP provides basic packet format definitions to support real-time communication but does not define control mechanisms or algorithms. The major components of an RTP header (Figure 2.1) include a timestamp, a sequence number, a payload type, a synchronization source (SSRC) identifier, and a marker bit.

Each RTP packet bears a timestamp to provide the timing information necessary to synchronize the media data for presentation. It also contains a sequence number to be used to determine whether packets have been lost or have arrived out of order. The payload type field is used to indicate the kind of payload data contained in the packet. For each source of an RTP session, there is a unique identifier called the synchronization source identifier (SSRC) that is carried by all packets originating



V: Version P: Padding

Figure 2.2: RTCP Header Format

from that source. The interpretation of the marker bits varies with media types. For example, in an audio packet, the marker bit defines the start of a talk spurt, while in a video packet it is used to indicate the end of a video frame.

RTP does not provide any mechanism to ensure that the data packets are delivered timely or in order. Neither does it provide reliable delivery or any QoS guarantee. It is augmented by RTCP to enable monitoring the quality of the data transmission, and to provide control and identification mechanisms for RTP transmissions [HSJ98].

Being a control protocol used in conjunction with RTP, RTCP does not transfer data. It just transfers information related to an RTP data stream by periodically exchanging control messages between participants. Besides providing control information, RTCP messages also serve as a liveness indicator for session members, in the absence of transmitted media data. An RTCP message consists of a number of "stackable" packets (a.k.a. information elements), each with its own type code and length indication. Using the RTCP message and its information elements, RTCP performs the following four functions:

1. Provide feedback on the quality of the data distribution

The most important RTCP information elements contained in RTCP messages are the sender report (SR) and the receiver report (RR). The sender report is generated by a participant that has recently sent data packets and contains the number of packets and bytes already transmitted as well as information that can be used for synchronization purposes. The receiver report contains statistics about the data stream such as the highest sequence number received so far, the inter-arrival jitter, fraction of lost packets since the last report, and cumulative number of lost packets. This is necessary for QoS monitoring. The sender may exploit the loss and jitter information contained in receiver reports to adjust its data rate, thereby adapting to the conditions experienced by the major fraction of receivers.

2. Identify RTP Source

   Besides the 32-bit SSRC contained in RTP packets, an information element called the source description (SDES) must be included in all RTCP messages to provide more detailed identification and description of each participant. This element describes the source in more detail, providing information such as the canonical name (CNAME) of the source (an RFC 822 email address style identifier), the real user name, email address, telephone number, geographic information, application name, and/or private extensions in text. Such information may be displayed in participants' applications and/or used for contact or debugging purposes.

3. Control the RTCP Transmission interval

   Since all participants in a session are required to send RTCP messages, as the number of participants increases, the control message traffic could grow too large relative to ordinary data transmissions. To prevent control traffic from overwhelming network resources and to allow RTP to scale up to a large number of session participants, control traffic should be limited to a certain percentage of the overall session traffic. Each participant can discover other participants by listening to incoming RTCP messages for source descriptions. The interval for generating RTCP messages is then scaled according to the number of participants involved in the session. Since information from senders, in particular their canonical name and synchronization information, is more important than information from receivers, RTCP divides the control bandwidth so that all active senders, usually just one or two, get a quarter of the control bandwidth, while the passive receivers share the rest equally [Sch95].

4. Convey minimal session control information (optional)

   This function uses RTCP messages as a carrier for some minimal amount of information. It's mainly aimed at so-called "loosely controlled" sessions where participants can enter and exit the session informally without explicit membership control and set-up. In such a scenario, information like user identification can be conveyed through RTCP messages without resorting to the use of the RTP protocol.

In addition to end systems, RTP supports the notions of "mixers" and "translators", which are considered to be "intermediate systems" built on top of RTP. The application-level media gateway (described later in Chapter 3) is an example of such a system. A translator may convert a media stream between two encoding schemes without modifying the stream's synchronization or SSRC identifier. Examples of translators include video converters, media gateways and firewalls. A mixer receives one or more media streams from different sources, combines them together, possibly also converts the data encoding, and puts its own SSRC identifier on the output media stream. Thus the output stream is identified as having the mixer as its synchronization source.

RTP is really a protocol framework that deals with application-independent issues. To have a complete specification of RTP for a particular application, one or more companion documents: "application-dependent profiles" and "payload format specifications" are required. For example, Schulzrinne *et al.* [HSJ98] defined a profile for audio and video conferencing with minimal session control, as well as a set of standard encodings and their names for use with RTP.

2.2 Java Media Framework (JMF)

Due to recent increased interest in multimedia applications, efforts have been made to develop supportive application frameworks that can facilitate multimedia programming through consistent interfaces. With regard to Java, the ability to incorporate continuous media (also known as time-based or streaming media) into Java applications is provided by the Java Media Framework (JMF) [JMF99]. To enable enriched media, such as voice, music, and video, to be used in Java-based applications, JMF defines a hierarchy of Java classes and interfaces that encapsulate the essentials of multimedia sessions. In addition, to support extensibility, JMF also allows advanced developers to customize and extend JMF functionality via a plug-in API and by re-implementing related interfaces.

The design of JMF centers on time-based media because a continuous stream of multimedia data must be delivered and presented to the user within a stringent deadline for the media to be intelligible. Figure 2.3 illustrates the general media processing

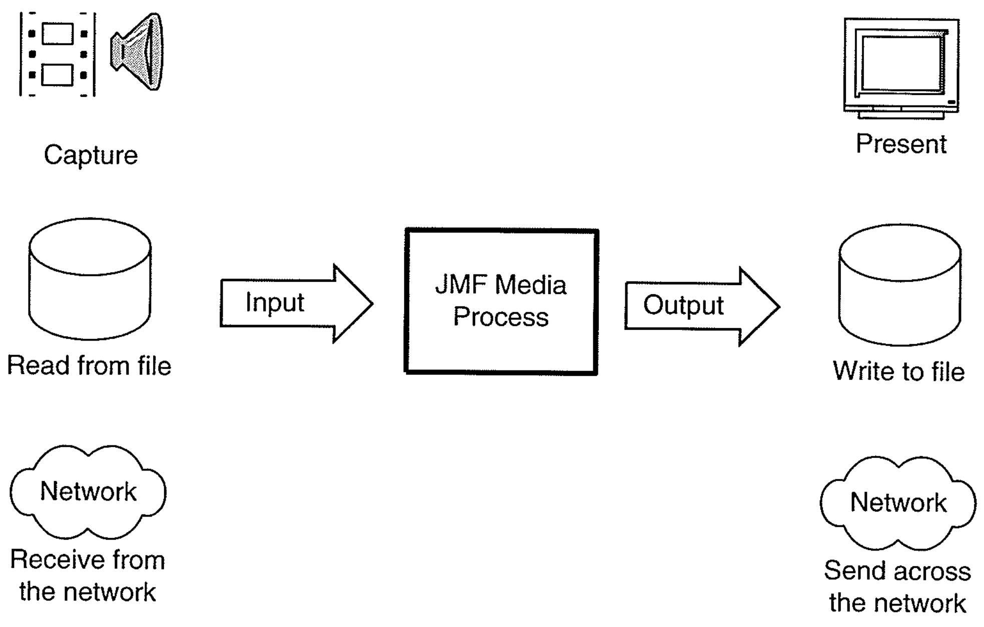


Figure 2.3: JMF Media Processing Model

model of JMF. A JMF media process includes applying effect filters, compressing/decompressing, and converting betweens different formats. The process may take inputs from a variety of sources, such as capture devices, local files, and/or network streams. The media data output may be destined for display devices, local files, and/or the network for real-time transmission. The major components in JMF are *DataSources*, *Processors*, *Players*, and *DataSinks*, which are created, coordinated and managed by a *Manager* object in JMF. A *DataSource* represents an object that can produce one or more media streams. It encapsulates the location of the media, the protocol to be used to write or read the data, its format, and so on. A *DataSource* can be either a *PushDataSource* or a *PullDataSource*, depending on who initiates and takes control of the data transfer. For a *PushDataSource*, it's the server who starts and controls the transfer, while a *PullDataSource* requires the client to start and control the transfer.

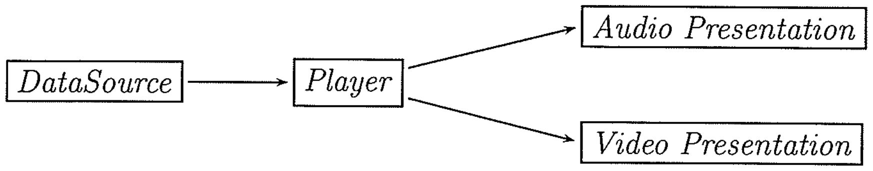


Figure 2.4: JMF Player Model

As shown in Figure 2.4, a *Player* takes a media stream from a DataSource as input and renders it to users. The destination it renders it to might be a screen and/or speakers, depending on the type of media. A *Player* provides the basic functionality of media presentation, but does not expose any control over the processing it performs or the process of rendering media data. For those developers who want to customize such processing, or capture media streams from a device, a *Processor* should be used instead. A *Processor* provides all the capabilities of a *Player*, and it also enables media processing customization and media capture. Besides taking a *DataSource* as input, a *Processor* can also output the manipulated data to another *DataSource* for further manipulation, presentation, or storage in a file (Figure 2.5).

To render media streams that are contained in *DataSources* to destinations other than a presentation device, a *DataSink* must be used to connect to the *DataSource*, either automatically or manually. A *DataSink* can read media data from a *DataSource* and write it to destinations such as a file, or a network.

JMF incorporates RTP as its default protocol for streaming media transmission[2]. For each media type transmitted over the network, there is a correspond-

[2]JMF handles the reliability of streaming media transmission by using the control mechanism

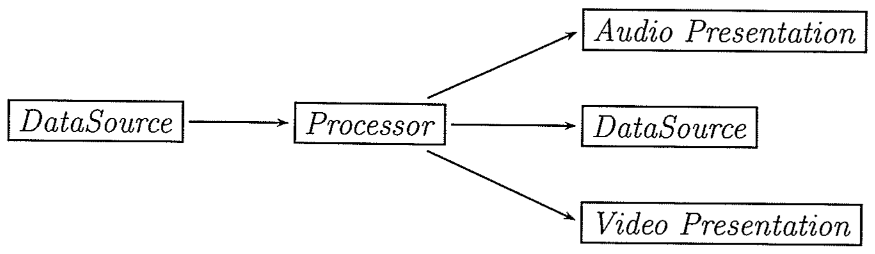


Figure 2.5: JMF Processor Model

ing RTP session. In JMF, an *RTPSessionManager* (RTPSM) manages an RTP session. The major objects managed by an RTPSM include the session participants (*RTPParticipant*) and session streams (*SendStreams* and *ReceiveStreams* in JMF, respectively), as well as the statistical information conveyed by RTCP messages. Each RTP stream has a buffer data source that is JMF compliant associated with it. Thus, with the help of a data source, RTP streams can be easily accessed or presented using the facilities provided by JMF (e.g. the RTP data source could be connected to a *Player* for presentation, or to a *Processor* for manipulation).

Although JMF uses UDP as the underlying protocol for its RTP implementation by default, RTP support in JMF is designed to be transport protocol independent. Since RTP can run over other transport protocols, JMF allows developers to write new data handlers for RTPSM to work over other specific transport layers. This extensibility can be achieved by customizing *RTPSocket*. *RTPSocket* is a data source providing both input and output streams to stream data to and from the underlying network. RTPSM uses *RTPSocket* to interface to the network. To deal with specific transport protocols, a developer could create a new *RTPSocket*

priovided by RTCP

incorporating the developer's own input and output streaming implementation, and pass it to RTPSM.

2.3 Active Networks

Active Networks propose that, by providing programmable interfaces in network nodes and more intelligent "active packets" that can use resources within the network nodes, users can easily construct or refine new network services. The network is "active" in two ways [TW96]:

1. Routers and switches within the network actively process (i.e. perform computations on) the user data flowing through them. The traditional network transports data from end to end as passive packets without modifications. Most network services only focus on transmission issues and the computation done in the network is limited to protocol processing. As computing power becomes cheaper and the network applications become more varied and demanding, it becomes attractive and possible to deploy functionality in the network itself. Doing so offers the potential to provide users with better services. By introducing the ability to execute user-defined programs, the function of the network is no longer restricted to packet delivery, and may act as a general computation provider doing such things as, for example, bit rate adjusting on media streams.

2. Individual users and/or administrators can inject customized programs into the network, thereby tailoring router processing to be user and/or application specific. The emergence of new distributed applications has increased demands for new and better network services. However, with the exponential

> increase in scale of the Internet, deploying new network services is becoming more and more difficult. Active networks may be seen as a "revolutionary" step from the existing, legacy network architecture to one which allows users to extend the functionality of the network by programming active nodes and injecting the required code into the network. This opens a window towards dynamically and automatically deploying user defined services and computations.

Tennehouse and Wetherall [TW96] first proposed the concept of active networks. Currently, there are several active network prototypes under development, including ANTS [WGT98], CANES [CBZS98], Netscript [Yd96], and Switchware [AHK+98]. These prototypes share the fundamental design issues of execution environment, API extension, code distribution, and resource management. Users are also allowed to construct their own services in these prototypes. ANTS uses Java byte-codes as the basis for its platform-independency. Capsule routines in CANES are simply represented by ordered sequences of node primitive operations. Netscript provides a flow-oriented language for service composition. Switchware provides a proprietary programming language, PLAN, for service composition. PLANet [HMA+99], which is implemented in Switchware, is a pure active internetwork[3], while ANTS, CANES, and other prototypes tend to encapsulate their active packets within TCP or UDP.

Since active networks advocate a revolutionary architecture that overthrows the traditional network services model, it is crucial for active networks' success that the technology can co-exist with legacy networks and that active nodes can be incrementally deployed.

[3]The network layer services in PLANet are implemented directly on top of link layer, and don't reply on TCP/IP or other kind of network infrastructure.

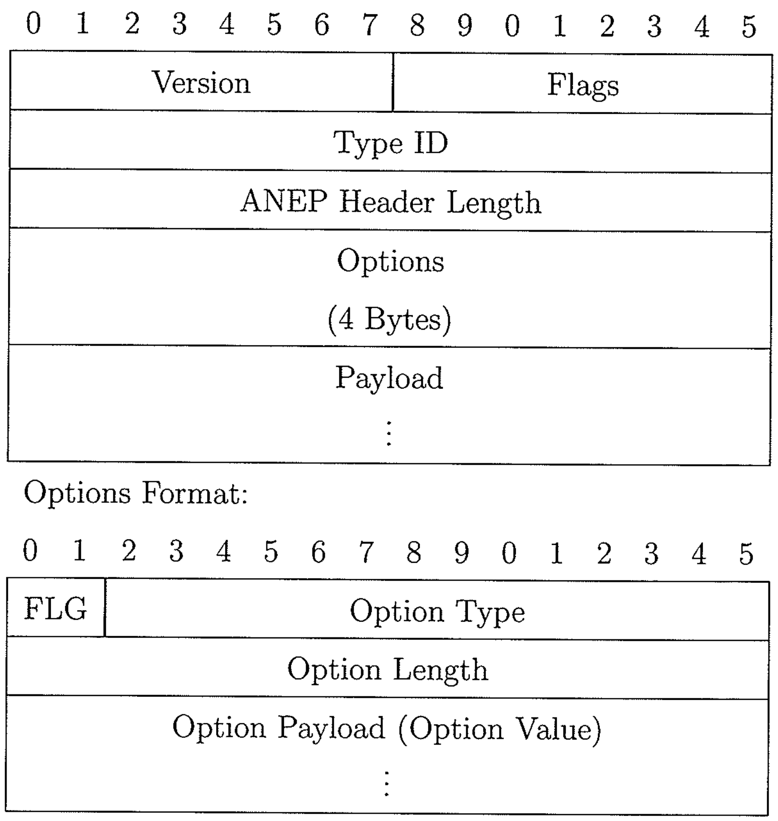


Figure 2.6: ANEP header format

ANEP [ABG+97b] specifies a mechanism for encapsulating Active Network packets for transmission over different media, and thereby making them transparent to network nodes that do not support active packets. The Option Type field in an ANEP header (Figure 2.6) essentially identifies the code that active network nodes should use to handle this active packet. If the service corresponding to a delivered Type ID does not exist at a given active node, automatic code distribution (deployment) for the service program will be invoked. ANEP is also the fundamental transmission protocol for the large-scale active network testbed, called the ABONE, which provides a practical test environment for active network experiments. The ABONE includes diverse OS platforms distributed across many organizations. Each

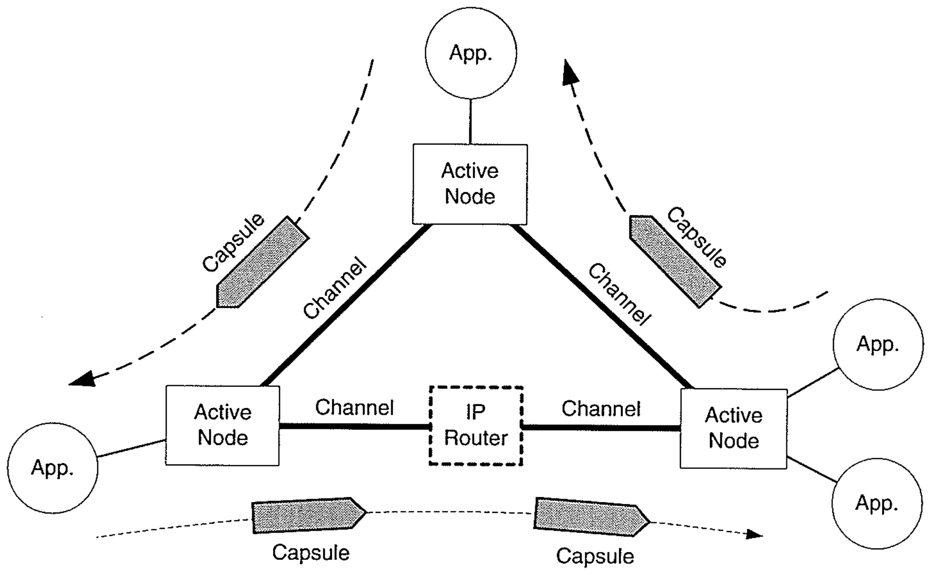


Figure 2.7: Entities in an ANTS Active Network [Wet99]

ABONE node is managed by Anetd, an experimental daemon specifically designed to support the deployment, operation and control of active networks [DMRS00].

2.3.1 The Active Node Toolkit System (ANTS)

ANTS was the first active network prototype, with which Tennehouse and Wetherall proposed the concept of Active Networks. Since then, ANTS has received significant attention in the network research community. ANTS provides an experimental environment for testing active network concepts, such as active processing and dynamic code distribution. To do so, ANTS abstracts a set of entities that compose a communication network (Figure 2.7). Active nodes replace traditional routers, switches, and end nodes to serve as the constituent element of an ANTS-based network. Active nodes are interconnected with link layer channels directly or with conventional IP routers in between. ANTS supports two types of link channel. In standalone mode, capsules are carried within UDP datagrams. ANTS may also run using Anetd, where it exploits the services of Anetd to encapsulate active capsules using ANEP for transmission.

The major role of an active node is to provide a restricted execution environment for user-defined programs without sacrificing programmability and flexibility. ANTS achieves this by exposing a set of node primitives [WGT98] for both applications and user-defined network services to use. These include:

1. Environment access: to query the node location, state of links, routing tables, local time and so forth;
2. Capsule manipulation: with access to both capsule header fields and payload;
3. Control operations, to allow capsules to create other capsules and forward, copy or discard themselves;
4. Node Storage: to manipulate a soft store of application-defined objects that are held for a short interval.

Passive data packets found in legacy networks are replaced with active "capsules" that carry not only data but also a reference to a forwarding routine to be invoked when a capsule arrives at an active node. The forwarding routine is typically carried as a reference to keep the capsule lean, since for commonly used forwarding routines, that are likely to already be resident at the active nodes, it is more efficient to just send a reference than to send the actual code. The code for such forwarding routines can normally be assumed to be at every active node. For other user-defined programs, which are mostly application-specific, the code distribution mechanism can be invoked when a node identifies the absence of the needed code. A collection of related capsule types can also form a "code group", whose forwarding routines are transferred together during code distribution.

Dynamic code distribution is one of the principle benefits provided by active networks. ANTS proposed an in-band[4] code transport mechanism to propagate user-defined programs to the active nodes where they are needed. A capsule is sent into the network assuming that the required active code already exists at the nodes through which the capsule will pass. This is why only a reference to the forwarding routine needs to be carried in the capsule initially. A code cache at each active node is used to store the forwarding routines that have recently been used by capsules and to help avoid redundant transmission. When a capsule arrives at a node, the code cache is checked for the required code. A "miss" in the code cache will cause a load request for the missing code to be sent to the upstream node (i.e. the node from which the capsule arrived). Since the capsule comes from the upstream node, it must have just been executed on it. Thus it is more likely that the required code is still residing in its code cache. Similarly, if the upstream node found the required code missing in its code cache, it will ask its upstream node for the code. Such process is repeated until the code is found present or the sender of the capsule is reached. The execution of capsule code is suspended until the node receives the requested code contained in a load response. When the node receives such a load response, it puts the code into the code cache, locates any suspended capsules waiting for that capsule type, and then resumes the capsules' execution. This type of "code-on-demand" distribution scheme enables new network services to be de-

[4]The in-band code transport approach couples the transfer of code with the transfer of data so that the code can be distributed and executed along the data path automatically. In contrast, using an out-of-band approach, the code distribution is separated from data transmission, and normally, happens in advance of data transmission.

ployed in an automatic and flexible way. A user may deploy the new service code simply by injecting the corresponding capsules into an active network, and the code will be automatically propagated to all active nodes along the transmission path incrementally.

Users program ANTS by developing their own active protocols and applications. ANTS allows users to construct their own applications by extending the base *Application* class which provides primitives for its subclasses to use to access the local node and its services. The two essential primitives support sending a capsule into and receiving a capsule from the network. In ANTS, a protocol is defined as the unit by which the network as a whole is customized by applications [WGT98]. Practically speaking, a protocol is just a collection of related code groups to be treated as a single unit of protection by active nodes. Since the code group which is the unit of transfer during code transport is comprised of related capsules, implementing a protocol actually consists of implementing all the code group's constituent capsules. To activate a new protocol, an application must register it with the active node on which the application resides before sending or receiving any of the protocol's capsules.

Chapter 3

Problem Description and Motivation

3.1 Environment

Remote collaborative applications allow a group of users or software systems that are geographically dispersed to work together by exploiting telecommunication and computer technologies. To achieve better collaboration, various forms of continuous media (e.g. video and audio) are normally exchanged between collaborating users. For example, with face-to-face collaboration as in video conferencing, those continuous media may be the live video and audio of the participants, while in some specific collaborative applications, other application-specific media streams may be transmitted via the network. An example of a collaborative application is tele-medicine, which can help to deliver improved healthcare services especially to rural areas. Typical applications of tele-medicine include tele-mentoring, tele-consultation, tele-diagnosis, and even tele-surgery, all of which require the real-time transmission of various forms of multimedia data between two or more locations. As one of the applications, tele-surgery inherits all the characteristics of tele-medicine, and is also more demanding in terms of real-time communication requirements than most other applications[5]. The primary goal of tele-surgery is to enable remote viewing and consultation during surgical procedures. The "digitized operation" is transmitted from an operating theatre to collaborating surgeons, consultants, administrators, students and other observers (most likely via multicast technology). A digitized operation must capture virtually every aspect of the surgery. These include:

1. Video streams from the operating theatre to remote sites: Video streams will be the most bandwidth-consuming type of data flowing in the network. For monitoring an operation remotely, a video stream from just one camera is not enough. Surgeons at remote sites normally require the ability to switch between different video sources (e.g. different camera angles) according to their interests or needs to improve their understanding of how the surgery is proceeding. Thus, several cameras are typically needed in the operating theatre, which will generate simultaneous video signals. To save bandwidth, such video streams are commonly compressed for transmission over a network. To be acceptable and useful to viewers at remote sites, the decompressed video must be of sufficiently high quality (both in terms of such characteristics as resolution and being free of jitter).

2. Audio streams in both directions: Audio stream transmissions should be bi-directional to support interactions between surgeons and consultants. For audio streams, the only parameter of concern is quality. Since audio streams

[5]As such, tele-surgery makes an excellent "motivating example" to consider in this thesis.

are not as bandwidth intensive as video streams, they are normally easier to deal with and the quality is also easier to guarantee.

3. Other data: Besides audio and video streams, there is also other information that could be essential to a remote viewer's overall grasp of the surgical process. These data include some time-critical information such as the patient's blood pressure, pulse rate and other vital signs, less time-critical information such as patient records, diagnostic images and pathology data, as well as data that may be generated by interactive facilities such as distributed whiteboards and chat systems (the latter two being bidirectional in nature).

Remote collaborative applications call for powerful, yet flexible network infrastructure to meet the diversity and requirements of the various data streams that must be transmitted in realtime between the participants. Developements in multimedia networking, such as multicasting, video/audio CODECs and real-time transmission, have pushed the evolution of remote collaborative applications forward. However, given the high throughput requirements such applications place on the network infrastructure, the capabilities provided by current network technology can provide are limited and may sometimes be inadequate. This problem will be exacerbated as more and more aggressive network applications are deployed. Futhermore, the heterogeneity of the network environment (i.e. the huge variety of applications and end systems interconnected using various network technologies) makes it difficult to efficiently deliver large volumes of data to all participants in a collabarative application. Additionally, the network link conditions in such heterogeneous environments may vary enormously. Some portions of the network may be connected with relatively slow links (particularly in remote areas), while other portions are connected with high-speed links.

Since multicasting facilitates one-to-many transmissions by having the network do the data replication, all receivers in a multicast session commonly share the same source data stream sent at the same quality and rate. For applications using multicast to transmit data streams to heterogeneous participants, the source normally has to run at a rate that matches the most constrained receiver, even though some receivers with high bandwidth connection are capable of receiving correspondingly higher quality data streams (this is undesirable). Additionally, not all data streams are of equal value to all observers at the same time. Thus it may be neither necessary nor efficient to simply replicate and forward the full-quality media streams to all participants, given the outgoing link conditions and preferences of the receivers.

Approaches to providing better network support for data-intensive real-time multimedia applications are still a hot topic in the research community. Resource reservation is one possible solution which explicitly reserves network resources such as link bandwidth and buffers along all the paths to be traversed by media streams to ensure timely delivery and guaranteed Quality of Service (QoS). The resource reservation protocol (RSVP) [ZDE93] is a receiver-driven protocol that works in conjunction with network layer protocols like IP. Applications can invoke RSVP to request desired unidirectional QoS (e.g. peak/average bandwidth and delay bounds) from sender to receivers. RSVP supports both unicast and multicast protocols and scales well for large multicast groups. However, there are still some problems with using RSVP on a WAN. First, reserving an entire path may not be possible in a heterogeneous network [RT98]. One reason is there may be some shared link (e.g. an Ethernet segment) that makes it impossible to set aside a fraction of the bandwidth of the link. Also, if there is a link on the path between the sender and receiver with low bandwidth, a reservation for resources beyond the upper bound of that link will always fail. Another issue is the sub-optimal use of

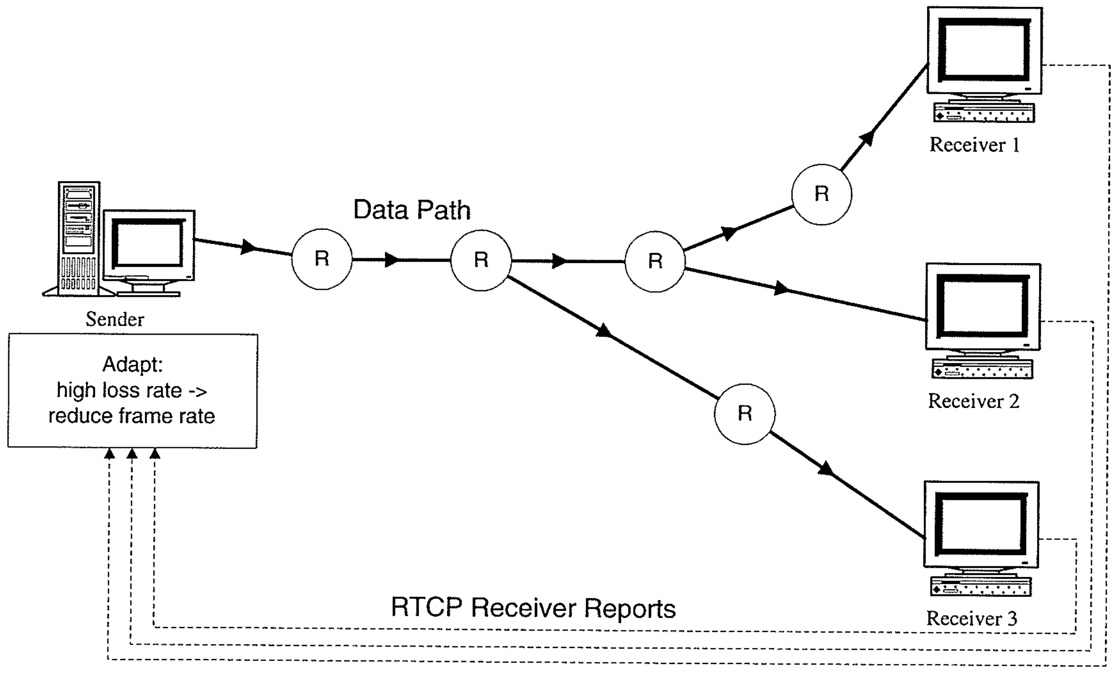


Figure 3.1: Sender-Initiated Adaptation Using RTCP Reports

network resources caused by resource reservation. Since the receiver has no knowledge about the current network load, there exists the possibility of over aggressive or conservative reservation of resources based on the receiver's estimation. Furthermore, a reservation always holds for the entire session, and the network load may vary significantly during that time. Thus, if the receiver makes its reservations during a busy period, the network can only provide limited resources that will leave the receiver with poor quality media streams even if the load is mitigated later. Further, if the receiver makes an aggressive reservation at the beginning but the network gets busy later, it may prevent other users from getting their desired QoS despite the fact that sufficient resources may actually be available.

Another approach to supporting realtime multimedia transmissions is application adaptation that lets the user application adapt dynamically to changes in available network resources and application needs. Instead of explicitly reserving network resources, this approach adapts to changes by making a tradeoff between quality and data rates. Changes in available network resources may be detected in various ways. The approach described by Busse *et al.* [BDS96] uses RTCP receiver reports to collect information, like the loss rate, from receivers (Figure 3.1) so the application can adapt its stream quality if necessary. In this scheme, it is assumed that packet loss is induced by network congestion. To avoid the situation where a receiver connected via a low bandwidth link may report high loss rate, and therefore force the sender to provide low quality to all receivers, the sender calculates the loss rate statistics from the proportion of unloaded, loaded and congested receivers. If the loss rate statistics are high, which means the current network load is high, the sender reduces the frame rate of the video being transmitted.

Another category of schemes initiates the adaptation at the receiver side. An example is the approach using layered video described in [MJV96]. Layered video is a composite of multiple layers with a base layer providing the lowest quality of video signal, and each additional layer enhancing the quality of the base layer. Layered video is propagated over multiple multicast sessions, with each session corresponding to one layer. A receiver may subscribe to all the sessions to get the full quality video, and un-subscribe to a subset of the enhancement layers when the loss rate experienced exceeds a certain threshold. When the receiver un-subscribes itself from a multicast session corresponding to a certain layer, the underlying multicast service will prune its link from the multicast tree of the session, thus saving bandwidth. This approach does the adaptation entirely at the receiver side. Figure 3.2 provides an example of layered video multicast over a heterogeneous

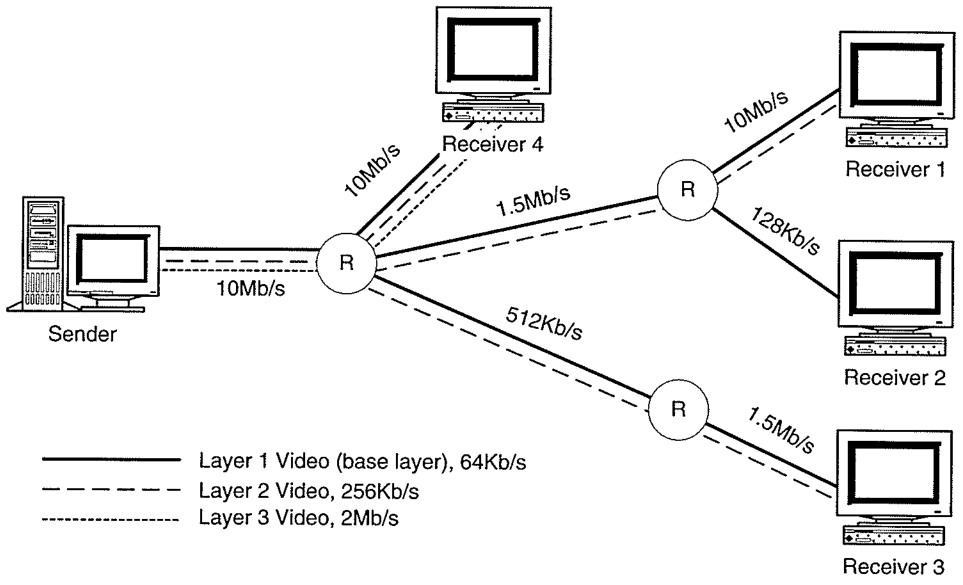


Figure 3.2: Receiver-Initiated Adaptation Scheme Using Layered Multicast

network. In the example, the source video stream is sent using 3 layers.

There are also drawbacks to the use of application level adaptation. Sender-initiated schemes are still subject to the network heterogeneity problem. Since the sender multicasts the same stream to all receivers, if the sender decides to lower the media quality for the sake of a fraction of the receivers who are experiencing high packet loss, all recipients in the same multicast group will be forced to accept the reduced quality. Layered multicasting avoids this problem at the cost of employing multiple multicast sessions. This requires building multiple multicast trees and using multiple multicast addresses for transmitting one logical stream, which may contribute to faster depletion of limited multicast addresses. Also, the media stream used in such schemes has to be layer encoded which makes the layered multicast scheme restrictive and incompatible with most existing video applications.

3.2 Motivation

This thesis is motivated by the desire to address the efficient transmission of real-time multimedia data in heterogeneous networks using in-network techniques (e.g. Active Networks concepts). In considering the drawbacks of approaches like resource reservation and traditional application level adaptation, in-network adaptation approaches offer potential benefits by being able to provide better network level support for multimedia services that can differ at various points in the network. The work described in the rest of this thesis builds on the following specific research work that has been done by others.

The Video Gateway [AMZ95] is an application-level scheme used to mitigate bandwidth heterogeneity. In this scheme, a video transmission is decomposed into multiple sessions with different bandwidth requirements in a fashion similar to layered video. Application-level video gateways are then deployed within the network to bridge different sessions by performing bandwidth adaptation through transcoding and rate-control. Receivers may convey their "receiver interests" into the network by employing application-specific protocols to dynamically control the behavior of video gateways. Although the gateway scheme may address the heterogeneity problem with video gateways physically situated at appropriate nodes within the network, the deployment of such application agents is still an issue due to the lack of native support in the Internet service model[6].

Building on the video gateway approach, Amir *et al.* [AMK98] further proposed an *active service* architecture to address the deployment problem. An active service

[6]The Internet service model does not allow automatic deployment of new network services. That means, the video gateways have to be deployed and configured beforehand manually, and once the network topology is changed, those video gateways have to be re-depolyed.

architecture enables clients to download and run service agents at strategic nodes in the network. Those service agents, called "servents", may perform user-defined computation like what the video gateway does. Compared to active networks, active services are more conservative since all computations are restricted to the application layer, while active networks advocate a programmable network layer. Thus servents are restricted from manipulating routing tables and forwarding functions that would contravene IP-layer integrity. Amir *et al.* also present a detailed example service, the Media Gateway service (MeGa) [AMK98], implemented by exploiting such an architecture. The MeGa service may be seen as an evolution of the application-level video gateway. In this service, the servents embody the ideas of JPEG/H.261 [CCI90] transcoding and rate control and act as an application-level agent that transparently bridges two RTP sessions carrying multimedia data.

The active service based media gateway is an in-network adaptation scheme that mitigates the bandwidth heterogeneity problem. Although it is argued that the active service framework adequately and effectively supports applications like video gateways at the application-level, there are still some points that make an active network (or other in-network, network layer approach) based approach preferable. An active network based approach can benefit distributed collaborative applications in the following ways:

1. Active networks can bridge the network and application layers. Clark and Tennenhouse [CT90] pointed out that multimedia applications could be simplified and both application and network performance enhanced if the network protocol reflected the application semantics. The innovative active network model has pushed this further. Unlike traditional network layer services, the services in active networks can process not only the packet header fields, but

also the application-specific payload. This makes it possible for user-defined network services to perform functions usually only feasible at the application layer. Such network layer services may do media transcoding, rate control, and quality control over the media data to accommodate various network conditions using information only available at the network level. On the other hand, for the tranditional routers to perform such application-oriented functions, additional service layers other than network layer will have to be implemented. Also in order to utilize such higher layer services, packets have to be encapsulated in some higher layer protocols. This will add more overhead to the network transmission. Active networks aims to solve the problem at the network layer and thus can avoid the overhead involved with having many additional layers in routers.

2. Since services in Active Networks run at the network layer, they can exploit knowledge of the network (e.g. topology and load conditions) to achieve effectiveness and efficiency. Being isolated from transmission issues, application-level adaptation schemes can only exploit indirect metrics like loss rate to speculate on network conditions. Active network services may acquire network information directly since they are running at the network layer. They may also cooperate with other network services to form a more flexible, efficient, and application oriented network infrastructure. For example, a rate control service, implemented with active networks technology, can be easily combined with other services such as multicast routing to provide better in-network adaptation. This is one feature of the proof-of-concept prototype described later.

3. Active Networks also allow interaction between end application and network

services through specially designed capsules. Besides being used for carrying data, capsules can also be used to convey configurations into network nodes, or can be generated by network services to report current network conditions to end applications. The ability to closely integrate host activities with in-network behaviours is also used in the prototype described later to optimize bandwidth use beyond what is possible with MeGa.

For the benefits discussed above, Active Networks has made itself a promising candidate in our problems by providing a network infrastructure, with which a in-network receiver-initiated adaption can be implemented. This will benefit data-intensive real-time multimedia transmissions with a better QoS support, more flexible and finer-grained adaptation scheme. Such active network based solution will be described in the next chapter.

Chapter 4

Solution Strategy

4.1 Active Dynamically Adaptive Multicast

4.1.1 Active Multicast

The active network based dynamically adaptive multicast service described in this thesis is based on the active multicast prototype presented in [WGT98], which employs a source multicast tree for message routing[7]. In active multicast, receivers interested in a group communication send subscribe capsules toward the source to subscribe. Unlike legacy source tree algorithms that mostly use the flood and prune method to build the tree, the "active" algorithm builds the tree with subscribe capsules that travel upstream. When a subscribe capsule arrives at a node, it first looks up a "forwarding record" (corresponding to the multicast group and source) in the active node (router) cache, and creates a fresh one if none is found.

[7]While a source based tree is not ideal in a large scale multicast environment, it serves adequately for a proof-of-concept prototype.

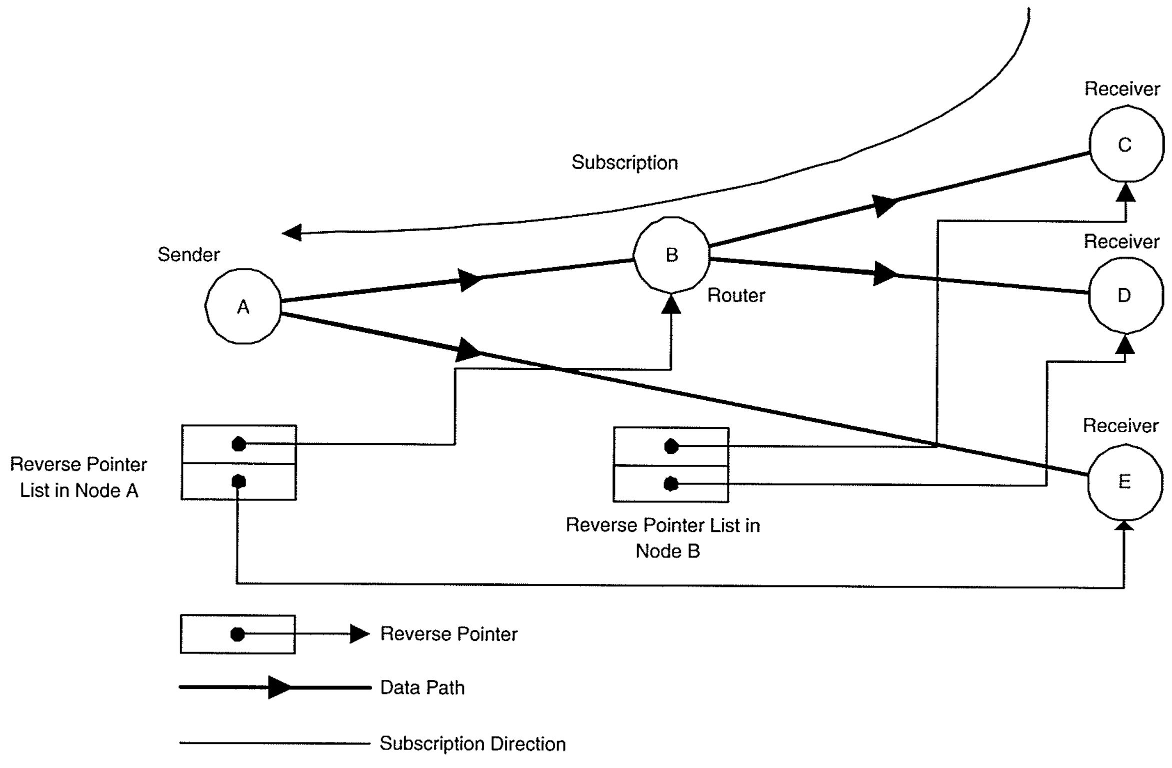


Figure 4.1: Reverse Pointers in Active Multicast

The forwarding records holding the multicast configurations are stored in the soft-store of intermediate nodes, which makes it possible to maintain information that needs to be persistent through a whole session. Such a record contains a list of reverse addresses, to which the multicast message should be forwarded. Once the forwarding record is located, a "reverse" pointer (the address of the node where the subscribe capsule comes from) is inserted into the list if the pointer does not already exist. Then, with the reverse pointer set to the address of the current node, the subscribe capsule is further forwarded upstream, until it reaches the source.

Figure 4.1 illustrates how reverse pointers work in active multicast. Node C, D, and E are receivers which send subscribe capsules upstream to the sender (i.e. source)

A. As the subscribe capsules travel through the intermediate nodes (i.e. node *B* in Figure 4.1) and eventually arrive at the sender, the reverse pointer lists on those nodes are built accordingly by them. As the figure shows, reverse pointers on all the intermediate nodes together form the multicast tree.

Active multicast propagates multicast data by sending multicast data capsules with data encapsulated within them [WGT98]. The forwarding routine for multicast data capsules uses the forwarding records it finds at active nodes. For each reverse pointer in a forwarding record, it sends a copy of itself along the path the pointer indicates. A multicast data capsule recognizes the end-node by recognizing an empty forwarding record. It then delivers itself to the receiver application. If no forwarding record is found at a network node, then there is no receiver exists downstream of the current node, so the capsule is silently discarded.

Active multicast makes forwarding decisions using the forwarding routine carried within each capsule rather than by using the pre-defined protocols residing on routers as traditional multicast schemes do. Thus by modifying the forwarding routine and introducing additional capsule types, dynamically adaptive multicast may incorporate application-defined adaptation with active multicast to provide better in-network QoS.

4.1.2 Dynamically Adaptive Multicast

In typical remote collaborative applications, media streams generated at the source contain multiple channels of data, which are called tracks. Each track belongs to one type of media (e.g. audio track or video). RTP based applications send tracks individually in different multicast sessions even when they target the same group and traverse the same multicast tree. Accordingly, transmission of a track could be adapted to accommodate the needs of individual receivers at end nodes, without affecting other tracks. The adaptation is implemented by manipulating the network packets of each single track to ensure the most important track get the best quality, while low-priority tracks can be constrained to low bit-rate transmission with correspondingly reduced quality or totally turned off. The active network based dynamically adaptive multicast service implemeneted in this thesis achieves in-network adaptability by dynamically controlling the transmission of individual media tracks in accordance with receivers' feedback and state information.

The track manipulation at each active node is initiated by the arrival of receiver feedback information reflecting a change of one or more user's interest in certain media tracks. Such interest changes may be generated by a user's interaction with the user interface of the application being run, and then be mapped to a change in the transmission priority for certain media track(s). Two specific types of user hints are currently detected and used by the adaptation scheme: One is whether the window(s) mapped to a particular video track are visible or not. When a window is not visible (i.e. the window is minimized or simply closed), the data stream corresponding to that particular video track to that receiver. The other hint is when the window(s) no longer needs to be sent mapped to a particular video track are overlapped by others or have been resized, which suggests that the video track is of less importance to the viewer. This will cause the corresponding video track to be transformed to a lower resolution. By doing this, the bandwidth saved can be reused for those video tracks that the viewer is focusing on.

Receiver feedback initiates in-network optimizations at the nodes in the multicast tree. The multicast tree used by the dynamically adaptive multicast service is a bi-directional source multicast tree rooted at the sender with receivers located at the leaf nodes. Each intermediate node has a single upstream node and one or more downstream nodes. All media tracks are transmitted along the same multicast tree, though each track is transmitted and processed separately. Media tracks are propagated downstream from the sender to the receivers by the same mechanism as active multicast, while receiver feedback is transmitted upstream from receivers to the source in capsule form. A feedback capsule corresponds to only one media track. When the feedback capsule reaches an active node, it locates the current information for the corresponding track. If the feedback information indicates a lessened requirement, the capsule code takes action to either start a JMF processor to lower the quality to a certain level or to just stop forwarding track data to the outgoing interface where the feedback originated. The track information stored at each node is also updated correspondingly. If the lessened requirement holds for all downstream recipients, then the lessened requirement is passed upstream to the next active node and the transformation is applied upstream too.

4.2 An Adaptive Multicast Protocol

As mentioned earlier, protocols in Active Networks are composed of a collection of related capsule types. The capsule types forming the dynamically adaptive multicast service include: *ARTMcastCapsule* (Adaptive Real-Time Multicast Capsule), *ARTMcastSubscribeCapsule* (Adaptive Real-Time Subscribe Multicast Capsule), *ARTCtrlCapsule* (Adaptive Real-Time Control Capsule), and *ARTProcessor* (Adaptive Real-Time Processor Capsule). *ARTMcastCapsules* are used to deliver media data from the source to all recipients, with the help of the forwarding records established by the *ARTMcastSubscribeCapsules.* A receiver that is interested in the media streams may join the multicast session by sending an *ARTMcastSubscribeCapsule* to the source. This capsule provides the functional-

```
public class CacheRecord {
    public InetAddr[] h;
    public long timestamp;
    public TrackRecord[] tr;
}
```

Figure 4.2: Structure of a Forwarding Record

ities of both multicast subscription and maintenance of track information. During a multicast session, a receiver generates feedback information and injects it into the network using an *ARTCtrlCapsule*. This capsule effectively maps changes of track importance to the observers into transformations on media tracks within the active network, and thereby implements the dynamic adaptability.

4.2.1 Forwarding Records

Active network nodes provide a soft-store mechanism for capsules to cache application-defined objects at active nodes. In the adaptive multicast service, a forwarding record is cached at every active node in the multicast tree. The *CacheRecord* class shown in Figure 4.2 implements the data structure that describes each forwarding record. It combines reverse pointers to support active multicast and track records for track-level adaptability. Instances of CacheRecord structure are stored in the soft-store of each active node.

Reverse pointers to the next downstream hops are stored in an array h (of type *InetAddr*). The field *timestamp*, introduced by the active multicast prototype, is updated by subscribe capsules to record the time of the most recent update of each forwarding record, and is used to avoid over-frequent subscription. Forwarding records are cached in the soft-store of a node. Since the soft-store can only

```
public class HostAddrs {
    float quality;
    public InetAddr[] hosts;
}

public class TrackRecord {
    public short trackID;
    public HostAddrs[] reqmnts;
}
```

Figure 4.3: Structure of the Track Record

hold application-defined objects for a short-interval, the forwarding record must be updated periodically to keep its existence in the storage. This is done by the receivers who periodically send subscribe capsules towards the sender (this will be mentioned later). However, if receivers send the subscribe capsules over-frequently, it could become a source of protocol overhead to upstream traffic from receivers to the sender. To avoid such overhead, when updating a forwarding record, the subscribe capsule fills in the field *timestamp* with the current time. When a new subscribe capsule concerning the same forwarding record comes, it checks the interval between the last update and its arrival. If the interval is too short, it will stop traversing upstream to update the forwarding record, since they still exist. Finally the field *tr* is an array holding the track records that specify what kind of transformation should be applied to the data of each track (e.g. forward untouched, kinds of modifications, or simply shut off). The structure of a track record is shown in Figure 4.3.

Media tracks sent by the source are uniquely identified by a short integer stored in the field *trackID*. For each track, there is a corresponding track record at every active node along the path it traverses which controls the transformation and forwarding of the track data. The array *reqmnts* organizes the lists of reverse pointers according to the different quality requirements on the data to be forwarded to the associated receivers. Each entry of the *reqmnts* array corresponds to a different quality requirement level and contains a list of reverse pointers to the nodes that are interested in that level of quality. For the tracks to be forwarded to the nodes pointed by the reverse pointers in that list, transformations will be made to them when necessary to ensure those nodes will get accordingly quality-adjusted ones. Nodes listed in the *reqmnts* array entry with a quality requirement of 0 (i.e. $reqmnts[DROP]$) will be cut off, while nodes listed in the *reqmnts* array entry with full quality requirement (i.e. $reqmnts[FULL]$) will be forwarded the full, unaltered, media track.

Because the active multicast service employs a source multicast tree, it uses combination of the group and source addresses as the key to locate and to access cached forwarding records. Thus, capsules with the same group and source address share the same forwarding records, and hence the same multicast tree. Figure 4.4 illustrates how the forwarding records are organized within the node cache.

4.2.2 Multicast Subscription

A receiver joins a multicast group by sending an $ARTMcastSubscribeCapsule$ to the source. This capsule type is responsible for building the multicast tree and initializing forwarding records, which will be used during the whole session. The algorithm for building the multicast tree is adopted from the active multicast prototype, and uses reverse pointers to establish a data path from the source to the receiver. Initializing the forwarding records also includes updating the track records. Because the forwarding behavior of each media track is controlled by track records, without updating the track records, the receiver will never get any

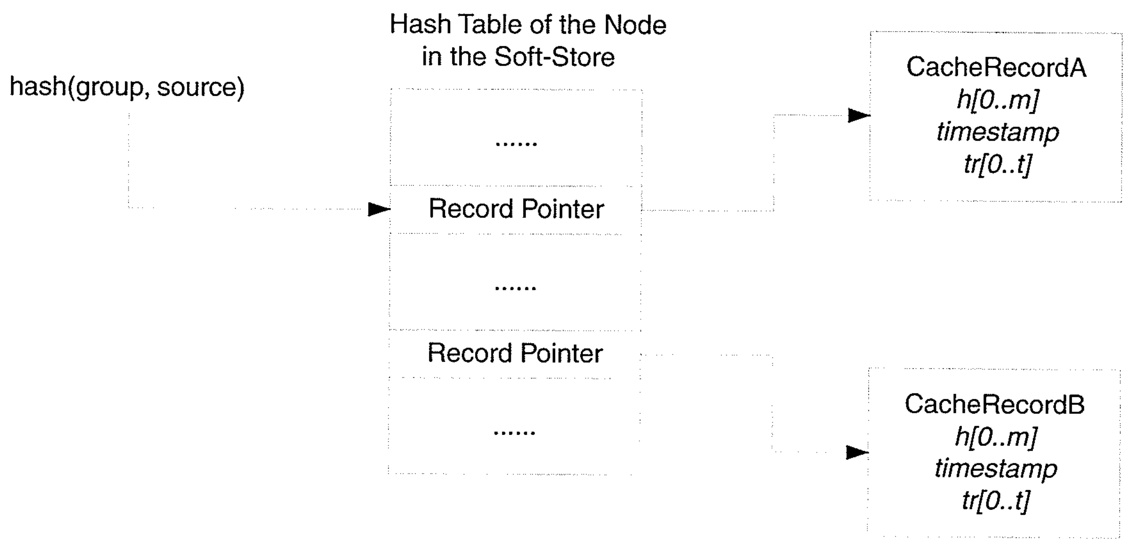


Figure 4.4: Organization of Forwarding Records in the Node Cache

data from any media track. The *ARTMcastSubscribeCapsule* contains an array *tu* (data structure abstracted in Figure 4.5) to convey its preferred quality requirement for each track. For the first time subscription to a multicast group, since the receiver has no knowledge about the media tracks the source is propagating, the *tu* array is empty and the receiver is deemed by default to be interested in all tracks at full quality[8]. If there are track records in the forwarding record, the capsule will merge its reverse pointer into the reverse pointer list which indicates that full quality is required for every track record, so that all media tracks will be forwarded through the reverse path in full.

Some active network frameworks remove cached entries from the soft-stores with

[8]While this is adequate in the prototype, assuming full quality may cause initial overloading of the network. To handle this problem, a more advanced implementation could receive guidance on initial quality levels from the active node nearest to the subscribing host and set the initial quality values accordingly.

Algorithm 1 Active Multicast Group Subscribing (*ARTMcastSubscribeCapsule*)

```
1: Locate forwarding record $m$ under the key $\langle self.group, self.target \rangle$ (Create a
   new one if none exists)
2: if $self.reverse$ not in $m.h[]$ then
3:   Append $self.reverse$ to $m.h[]$
4:   if $self.tu$ is null then
5:     {Initially, receivers have no knowledge about media tracks}
6:     for all track records $m.tr[i]$ do
7:       Append $self.reverse$ to $m.tr[i].reqmnts[FULL].hosts[]$
8:     end for
9:   else
10:    {Restore track records to avoid being pruned off because of timeout}
11:    for all track update records $self.tu[i]$ do
12:      Locate $m.tr[j]$ with $m.tr[j].trackID = self.tu[i].trackID$
13:      Append $self.reverse$ to $m.tr[j].reqmnts[self.tu[i].req]$
14:    end for
15:  end if
16:  {Build new self.tu that has the requirement for each track set to the maximum
     among those of the downstream receivers}
17:  New self.tu[m.tr.length];
18:  for all track records $m.tr[i]$ do
19:    Find $max(reqmnt)$ such that m.tr[i].reqmnts[reqmnt] is not null
20:    Append (m.tr[i].trackID, reqmnt) to self.tu[]
21:  end for
22: end if
23: {Check the time to avoid over-frequent subscription}
24: if $(localtime - m.timestamp) < 1000ms$ then
25:   return
26: else
27:   $m.timestamp \Leftarrow time$
28: end if
29: Route capsule to the source
```

```
public class TrackUpdate{
    public short trackID;
    public short req;
}

TrackUpdate[] tu;
```

Figure 4.5: Structure of *TrackUpdate* Record

coarse-grain timeouts, which will cause the related recipients to be pruned off the multicast tree. To avoid this problem, subscribe capsules must be sent periodically. When forwarding records are removed because of a timeout, it is up to the subscribe capsules from all recipients to recreate them. The recovery of reverse pointer list uses the same approach to rebuild forwarding records as was used to construct them originally. However, in addition to re-grafting branches onto the multicast tree, the subscribe capsule is also responsible for updating the track records in accordance with each recipients' current track quality requirements. Based on the local quality requirements, a recipient builds the *TrackUpdate* array with each item indicating the preferred quality of a certain track. Such information will be used to rebuild the reverse pointer lists for track records. After the forwarding record is properly updated, the subscribe capsule "sums up" the track records on the current node and thereby generates new track update information to forward upstream. The new track update information has the preferred requirement for each track set to the highest among all of the interested downstream receivers. The pseudo-code for this processing is shown in Algorithm 1.

4.2.3 Data Delivery

The *ARTMcastCapsule* is the wrapper and carrier capsule type for the media

Algorithm 2 Active Multicasting (*ARTMcastCapsule*)

```
1: Locate forwarding record $m$ under the key $\langle self.group, self.target\rangle$ (Exit when no such record exists)
2: Locate track record $m.tr[i]$ with $m.tr[i].trackID = self.trackID$ (Create a new one if not found with $m.tr[i].reqmnts[FULL].hosts[] \Leftarrow m.h[]$)
3: for all quality requirements $m.tr[i].reqmnts[j]$ do
4:    Replicate $self$ to $rep$
5:    Apply the transformation corresponding to $m.tr[i].reqmnt[j].quality$ on $rep$
6:    $rep.quality \Leftarrow m.tr[i].reqmnt[j].quality$
7:    for all hosts in $m.tr[i].reqmnts[j].hosts[k]$ do
8:       Route $rep$ to $m.tr[i].reqmnts[j].hosts[k]$
9:    end for
10: end for
```

data packet encoded by the source. It contains fields that specify the connection information, including: group address (*group*), source address (*target*), source port number (*spt*), and destination port number (*dpt*). Its "active part" performs the multicast function with knowledge of the reverse pointers available in each node's cache. Besides the data packets encapsulated as a byte-array, the capsule also contains other application-oriented information including the *trackID* of the track to which the contained data packet belongs (since each capsule carries the payload of just one media track). Field *quality* is an indicator of the current quality of the media track. Each media track is initially sent by the source at full quality. When the data capsule reaches a node, before being forwarded to downstream nodes, it looks up the quality requirement set by each downstream node in the track record. If less than full quality is required, the capsule code activates a processor to transform the media data to the appropriate reduced quality requirement before it is propagated further downstream. The corresponding operation from each *ARTMcastCapsule* which performs these operations on an active node is abstracted in Algorithm 2.

4.2.4 Receiver Feedback

The feedback information from receivers is carried in *ARTCtrlCapsule*s. The feedback could be generated by user interface hints or explicit event sources (e.g. buttons or menu items.) and serves to indicate that a certain media track has changed in importance to the observer. Such change is mapped to an appropriate transformation to be performed at intermediate active network nodes. *ARTCtrlCapsule*s share the *TrackUpdate* structure and most of the processing routines at each active node with the *ARTMcastSubscribeCapsule*. The major difference in what they do is how they handle the situation when no forwarding record exists. The subscribe capsule is in charge of building the multicast tree, while the control capsule just updates the forwarding behavior. So, when a needed forwarding record is not found, the subscribe capsule will create one, but the control capsule will just stop executing and hence does not propagate further. As mentioned, because subscribe capsules are sent periodically, the next subscribe capsule will be responsible for recovering the corresponding forwarding record. Algorithm 3 shows the pseudo-code for the operations that each *ARTCtrlCapsules* performs on an active node.

Algorithm 3 Receiver Feedback (*ARTCtrlCapsule*)

```
1: Locate forwarding record $m$ under the key $\langle self.group, self.target\rangle$ (Exit when
   no such record exists)
2: if $self.reverse$ not in $m.h[]$ then
3:   Append $self.reverse$ to $m.h[]$
4:   if $self.tu$ is null then
5:     for all track records $m.tr[i]$ do
6:       Append $self.reverse$ to $m.tr[i].reqmnts[FULL].hosts[]$
7:     end for
8:   else
9:     for all track update records $self.tu[i]$ do
10:      Locate $m.tr[j]$ with $m.tr[j].trackID = self.tu[i].trackID$
11:      Append $self.reverse$ to $m.tr[j].reqmnts[self.tu[i].req]$
12:    end for
13:  end if
14:  New self.tu[m.tr.length];
15:  for all track record $m.tr[i]$ do
16:    Find $max(reqmnt)$ such that m.tr[i].reqmnts[reqmnt] is not null;
17:    Append (m.tr[i].trackID, reqmnt) to self.tu[];
18:  end for
19: end if
20: Route capsule to the source
```

Chapter 5

Prototype Implementation

5.1 Architecture

The prototype is written in Java so that it can easily exploit both the active network infrastructure of ANTS (which is also written in Java) and multimedia support provided by JMF. To access the active network services and resources, the implementation is based on the programming model defined by ANTS [TW96] as follows:

1. The dynamically adaptive multicast protocol is developed by subclassing the virtual class "Protocol". Each type of packet and its forwarding routine is specified by subclassing the virtual class "Capsule".

2. End user applications are developed by subclassing the virtual class "Application" so that they can use the ANTS primitives to access the local node and its services.

3. An instance of the class Node represents the local ANTS runtime environment

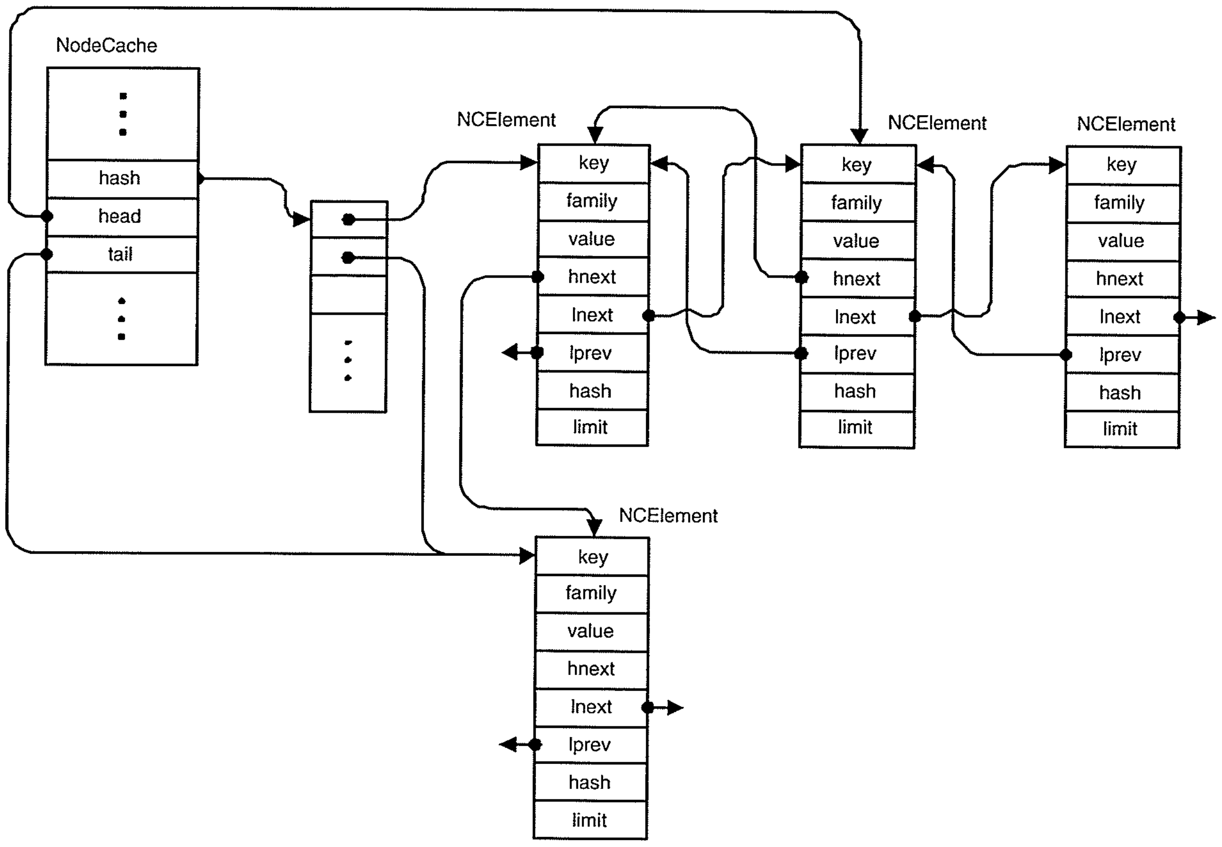


Figure 5.1: The NodeCache Hash Table and LRU Link List

on either a router or host node.

4. The protocol and applications are used by creating instances of their classes and attaching them to each node.

ANTS provides a soft-store mechanism where active network services can cache application-defined objects for an application-defined interval. Such objects (containing relevant information) can later be retrieved by another incoming capsule. The soft-store (implemented as *NodeCache* object) comprises a hash table and a doubly linked list to implement a fixed size table of application-defined objects

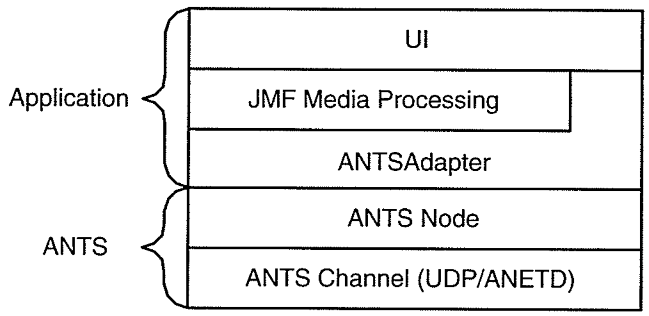


Figure 5.2: Prototype High Level Architecture

using least-recently-used replacement policy. Figure 5.1 shows how the hash table and LRU list are organized. Each application-defined object is represented by a *NCElement* object, which is stored in the hash table pointed by *hash* field in *NodeCache*. The *NCElements* with the same hash value is double linked by their *lnext* and *lprev* as a bucket. The *head* and *tail* fields of *NodeCache* mark both ends of the LRU link list. ANTS comes with two pre-defined wrapper classes (W_IJ, W_IIJ) for forming key object. W_IJ takes an $(integer, long)$ pair as the hash key, while W_IIJ takes an $(integer, integer, long)$ triple as the key. The forwarding records in the adaptive multicast service are identified by the combination of the group and source addresses which are both of type integer. Thus we use a W_IJ object to form the hash key for referencing forwarding records.

Figure 5.2 shows the high-level architecture of the prototype. Media processing is implemented using JMF. This includes media presentation, transformation, and RTP encoding and decoding. In our implementation, JMF media processing is integrated with the application, which runs on top of ANTS. JMF comes with a native RTP protocol implementation which allows java-based applications to play back and transmit RTP streams. This RTP support normally uses UDP as the underlying protocol, though in the prototype, active networks are substituted for

UDP. This is achieved by customizing the *RTPSocket* class, which, in JMF, is manipulated by *RTPSessionMgr* to send and receive RTP streams. *RTPSocket* is designed to have both a data and a control channel. Each channel has an input and an output stream to flow data in to and out of the underlying network. To enable the use of *RTPSocket* with ANTS network support, the default input and output stream, which are built using UDP support, were replaced with a new ANTS-oriented input and output stream called *ANTSHandler*. *ANTSHandler* bridges the RTP multimedia program and the ANTS active network by encapsulating RTP packets into the ANTS capsule at the sender side and de-encapsulating RTP packets out when they reach the intended receiver. For the convenience of active processing at the intermediate nodes, *ANTSHandler* also packs some important application-specific information (e.g. track number) into each capsule, so that it does not need to look into its RTP payload when the capsule is processed at an active node.

5.2 Implementation Details

To verify the adaptive scheme, an experimental environment was built where a sender and multiple receivers are inter-connected through active nodes. The sender program simulates the sources in the operating theatre by broadcasting multiple video and audio streams simultaneously. In consideration of the variety of kinds of sources in reality and to simplify the prototype, multiple audio and video files are used to simulate the live audio and video streams. The structure of the sender program is illustrated in Figure 5.3. The source media streams are represented with *DataSource* objects. JMF provides easy access to different input sources (like stored or captured live data) with its *DataSource* interface, so that data sources are not restricted to stored files, though stored files are all that is used in the prototype.

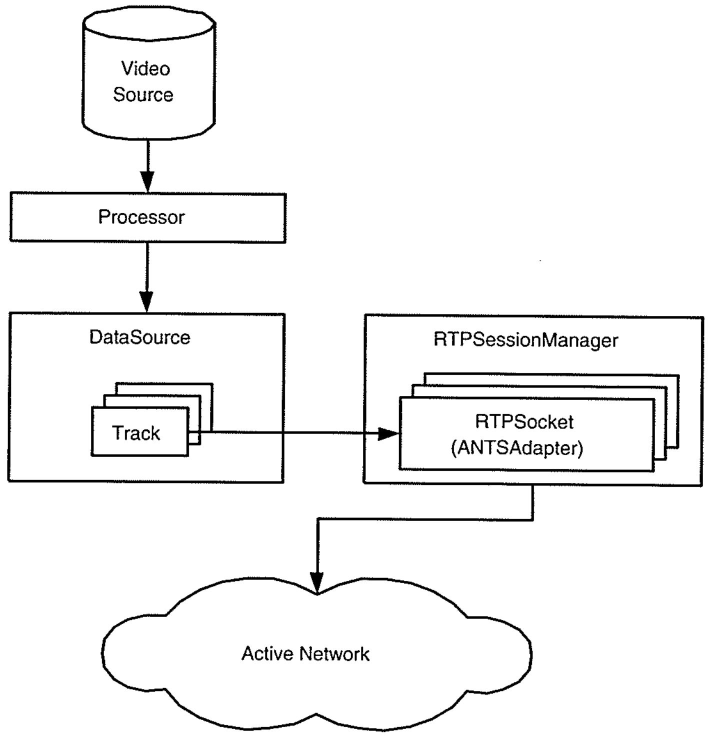


Figure 5.3: Structure of the Sender Program

For each video file to be sent, a processor is created to produce an RTP-encoded *DataSource* for transmission. Media streams often contain multiple channels of data called tracks (e.g. audio and video tracks in most video files). Those tracks in the source stream are de-multiplexed in the output data source to be transmitted individually. The transmission of each track invokes an RTP session, which is managed by an instance of *RTPSessionManager*. Normally, when a processor is asked for the output data source, it will initiate a session manager to transmit over UDP implicitly. To transmit using an ANTS network, the session manager must be created explicitly, and be supplied with the *ANTSHandler* (Figure 5.2) to disable the default UDP protocol support.

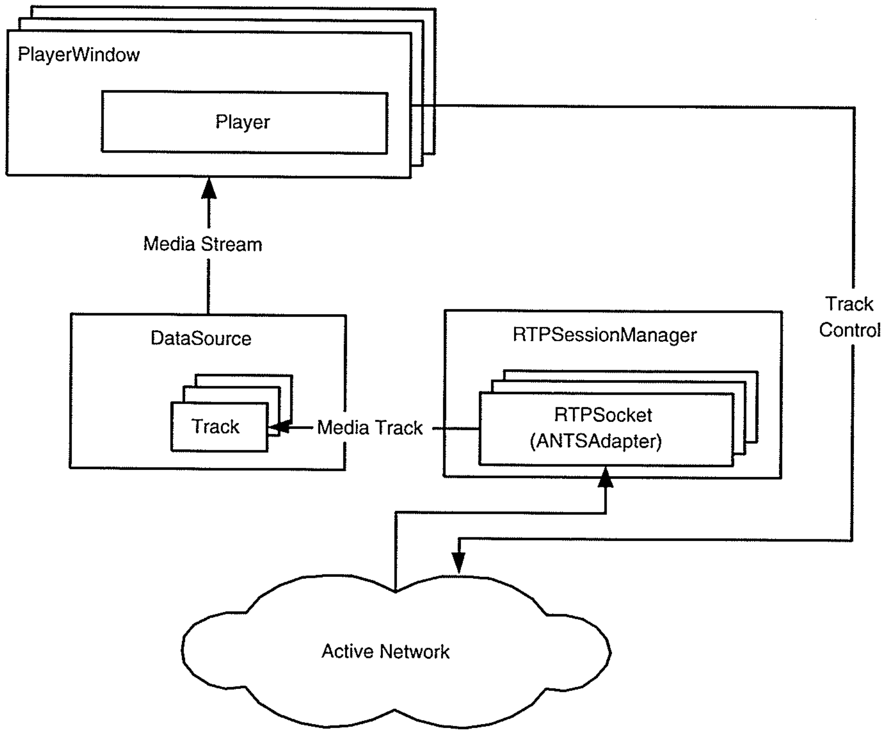


Figure 5.4: Structure of the Receiver Program

A receiver application presents media streams to the interested participants through virutal "speakers" and "monitors" (Figure 5.4). It employs JMF's Player objects to process the input streams of media data and render them in a timely fashion. The receiver application registers itself with the active network for receiving active capsules, so the multicast data capsules will be routed up to the receiver on their arrival at an end node of the multicast tree. The RTP encoded media data is then de-encapsulated out of the active capsule by the *ANTSHandler*. Managed by a *DataSource* object, the RTP encoded data can then be handled directly by a Player object. Receiver applications use *ARTCtrlCapsule*s to send feedback information into the network and initiate the previously described in-network adaptation. The sending of feedback capsules is separate from the data transmission. Since each

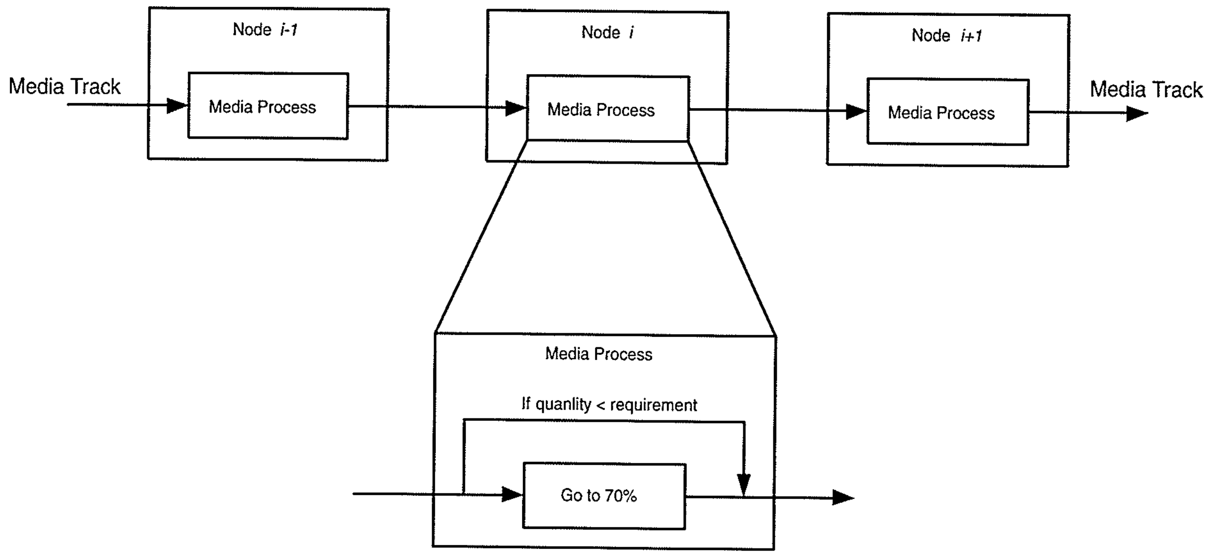


Figure 5.5: The Transformation Model on Active Nodes

ARTCtrlCapsule only contains quality requirements for an individual track, it can be integrated with various possible hint-detection schemes. In the prototype, the application detects user interaction with the windowing environment to generate requirement hints.

The crux of dynamic adaptability is being able to inject transformation routines into the active nodes in the multicast tree, so that the on-demand, in-network quality adjustment can be done. Since the quality requirements of different receivers may vary widely, it could be both inefficient and unnecessary (though perfectly possible) to write and deploy different transformation routines to suit all possible application needs[9]. The approach used in the prototype employs only one transformation routine to meet all levels of quality requirements. The transformation will lessen the quality to a fixed portion of that of the input, say, 70% of the original. Thus with

[9]Although where different routines make sense, it is certainly possible to do so.

several transformations working in sequence, different requirements can be approximately met (see Figure 5.5). This can also save network resources, since only one transformation routine has to be deployed. In the prototype, the *ARTProcessor* object performs the necessary transformation. To enable automatic deployment, *ARTProcessor* is implemented as a subclass of capsule. In an active node that media streams are passing through, there may be multiple instances of *ARTProcessor*, each corresponding to a different media track. The instance pointer of the appropriate processor to use is stored in the forwarding record. When a capsule for a track first arrives at a node, it looks up the corresponding *ARTProcessor*, or creates and initializes one if it is not found. This initialization will trigger on-demand code distribution if the *ARTProcessor* code is not located locally, so that as the *ARTMcastCapsule* traverses through the multicast tree, the processor will also be deployed along the transmission path. To save the processing resources of each intermediate node, each track will be processed on a node at most once.

As mentioned earlier, the media transformation is implemented with the help of JMF. In the JMF processing model, a *Processor* is the generalized object for media processing. It allows the application developer to define the specific type of processing that is to be applied to the media data. This enables the application of effects, mixing, and compositing in real-time. We exploit the *Processor* class to facilitate the development of the prototype's in track transformation ability. Each *ARTProcessor* initiates an RTP session by creating an instance of *RTPSessionMgr* corresponding to the transmission of a specific track. Each *RTPSessionMgr* uses the *RTPSocket* together with an *ANTSHandler* to send and receive RTP streams over the ANTS network. When the *RTPSessionMgr* detects that a certain track is flowing through the node, it will create an instance of a JMF *Processor* to process the incoming stream and forward it out. By calling the *getControl* method of the *Processor* object, the codec controls (such as *BitRateControl* and *QualityControl* objects) associated with this track will be returned. In the prototype, the transformation will lessen the quality to a fixed portion of that of the input (70% of the original by default). This is simply implemented by calling the *setQuality* method of the *QualityControl* object with parameter 0.7 (0.0 for minimal quality and 1.0 for maximum quality). This parameter may have different effects depending on the type of compression used by the track. For the tracks used later in the experiment (H.261 format), the quality adjustment is accomplished by changing the quantization[10] scale on a per macroblock[11] basis.

The dynamic adaptation service in this thesis is independent from user applications since the whole behaviour is controlled purely by capsules. This allows different events (e.g. loss rate reported by RTCP, UI hints, or user preference.) to be mapped to quality requirements and hence to collectively control the adaptation behaviour. To verify the design, in the proof-of-concept prototype, we use a variant of Katchbaw's UI code [KLB99] to detect user hints, and then initiate adaptation at the receiver side by injecting an *ARTCtrlCapsule* into the active network.

[10]Quantization is simply the process of decreasing the number of bits needed to store a set of values (transformed coefficients, in the context of data compression) by reducing the precision of those values. Since quantization is a many-to-one mapping, it's a lossy process and is the main source of compression in a lossy image coding scheme.

[11]Macroblock is the four 8 by 8 blocks of luminance data and the two (for 4:2:0 chroma format), four (for 4:2:2 chroma format) or eight (for 4:4:4 chroma format) corresponding 8 by 8 blocks of chrominance data coming from a 16 by 16 section of the luminance component of the picture.

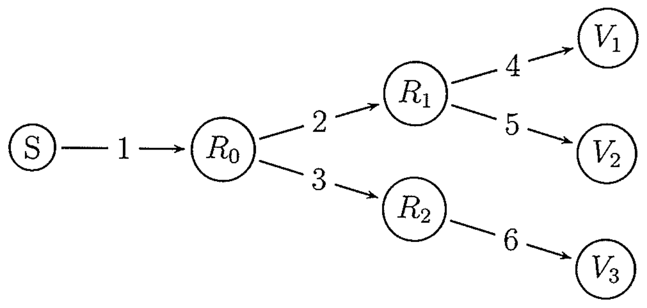


Figure 5.6: Test Configuration

5.3 Experiment

To verify the feasibility of the dynamic adaptive multicast concept, an ANTS based test environment was set up. The prototype was run on the ANTS platform in standalone mode without Anetd support. In the experimental environment, active nodes are inter-connected with communication channels established through UDP ports. This allows the prototype to operate on either a single-machine environment or on a testbed of multiple routers. The prototype was developed using JDK 1.3.1, ANTS 1.3.1, and JMF 2.1 for Linux. The test uses virtual nodes running in single-machine environment on an Intel x86 based platform running Linux kernel 2.4.17.

Figure 5.6 depicts the topology of the test environment. It consists of 6 nodes, amongst which there are 1 sender (S node), 3 receivers (V nodes), and 3 routing nodes (R nodes). Node S simulates the operation theatre, acting as the source of media streams. Three receiver nodes are used to simulate the remote viewers. And all the nodes in between behave as the active routers.

Clients tune in the multicast channel by registering themselves to the multicast group. In the test case, three clients representing the remote viewers were set up.

QoS Requirements			Transformation Routines			
V_1	V_2	V_3	S	R_0	R_1	R_2
100%	100%	100%	*None*	*None*	*None*	*None*
70%	100%	100%	*None*	*None*	*None*	*None*
70%	70%	100%	*None*	*None*	70%	*None*
100%	100%	70%	*None*	*None*	*None*	70%
70%	70%	70%	70%	*None*	*None*	*None*
49%	49%	70%	70%	*None*	70%	*None*
49%	0%	0%	70%	70%	0%	0%

Figure 5.7: The Transformation Behaviors on Routing Nodes

The client software is responsible for presenting the media stream and monitoring the user activity through UI interaction, in order to generate feedback capsules.

We used two clips from movie trailers[12] to simulate the real-time video signals from an operating theater. Each clip contains a video track and an audio track. Hence there were four media tracks in the experiment that could be controlled individually. For each media track, an RTP session over the dynamic adaptive multicast service was opened to convey the media streams to all registered clients. (All media streams are multicasted to a single multicast group).

In the experiment, the receiver has the option to receive the full quality media stream, to degrade the quality to a certain portion (70% or 50%), or to completely shutdown the receiving. All the combinations of all possible adaption schemes were tested, and the results show that:

[12]The chips were in H.261 Quarter Common Interchange Format (QCIF), $176 \times 144 pixels$, $30 frames/s$, $11 kHz$

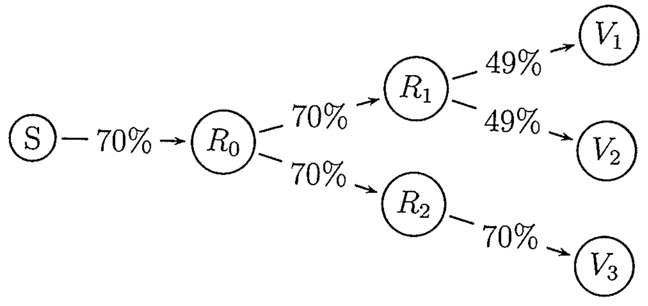


Figure 5.8: Bandwidth Usage Example

1. The active multicast based dynamic adaptive scheme has led to higher flexibility as to the degree of adaptation. Unlike the best-effort data multicast services, such as that implemented by IP which only allow a user to subscribe/unsubscribe to a certain multicast group, the clients using the dynamic adaptive service may have more control over their QoS requirements.

2. Adaption always happens on the nodes as close to the source as possible. Figure 5.7 lists some of the representative adaptation schemes on the routing nodes according to the combination of different QoS requirements initiated from different clients. Those data show that with the feedback capsules traversing upstream for the end-user, the transforming behavior on the routing nodes along the path is adapted automatically to achieve optimal link bandwidth usage. Figure 5.8 shows a example bandwidth usage on each link when the QoS requirements on V_1, V_2 and V_3 are respectively 49%, 49% and 70%. It is worth mentioning that for the QoS requirements shown on the last row of Figure 5.7, although both V_2 and V_3 have turned off the receiving, the media track still has to go through two routing nodes to get its quality degraded to 49%, since we only have the 70% transformation routine deployed in the routing nodes.

Source Stream		Processing Latency (*milliseconds*)	
Description	Compression Ratio	70%	50%
Original	1 : 1	877	875
High Quality	25 : 1	875	870
Low Quality	104 : 1	862	873
Low Quality *withMotionVectors*	113 : 1	871	865

Figure 5.9: Re-quantization Processing Latency by JMF Processor

3. Multiple media tracks can be conveyed in a single multicast session. In the experiment, two audio tracks and two video tracks sent to a single active multicast group, but the transmission of those tracks could still be adapted individually. In the traditional application adaption approaches, multicast groups are heavily employed to either transmit each individual media track, or to just deliver only part of a single track as necessitated by layered multicast approaches.

Additional experiments have been done on the processing time of the JMF processor based re-quantization, which is the major contributor of the transmission latency of the active multicast based dynamic adaptive scheme. As Figure 5.9 shows, different compression ratios have been applied to the sample video sequences during the experiments. Both 70% and 50% transformations are tested for comparison. The results show that in the current JMF based implementation, the transformation latency imposed on each active node for a single stream is a significant amount of time ($900ms$). This latency, being almost constant despite of the different compression ratios of the source, is also irrelevant to the transformation ratio. This has become a significant performance overhead that needs to be solved in the future improvement.

Chapter 6

Conclusions and Future Work

Multicasting and adaptive transmission are the most-commonly adopted multimedia networking technologies used to ensure the efficient delivery of large volumes of data. Generally, multicasting tries to minimize network load at the network layer, while most of the adaptive transmission approaches try to avoid unnecessary transmission from the application layer. In this thesis we have presented an approach to integrate dynamic adaptivity into media group transmission over a heterogeneous inter-networks by exploiting the emerging Active Networks technology. In comparison to traditional multimedia transmission solutions, the characteristics of the developed approach can be summarized as:

1. The prototype system provides a network layer based solution for multimedia distribution. This solution is facilitated by active networks technology. The network services like active multicasting and in-network adaptation are implemented using an Active Network framework as active protocols. Unlike in legacy network infrastructure, active protocols require no advance consensus about the definitions of the protocol. This can enormously simplify the de-

ployment of new network services, since the underlying active network does not have to “understand” the changing protocols itself.

2. Media data are conveyed from sender to receivers with active multicast. In contrast to legacy multicast mechanisms and protocols in which the packets can only be duplicated and forwarded passively, the active multicast capsules can determine their own forwarding behavior using the forwarding routines the multicast capsules carry. Moreover, the end user can modify the multicast algorithm for optimization without needing to upgrade the active nodes.

3. The dynamic adaptivity is implemented by incorporating the adaptation decision within the forwarding routine of the active multicast protocol. When the media data, encapsulated in the active multicast capsules, pass through an active node, the forwarding routine decides not only where they should it be forwarded to, but also the adaptive behavior to be applied to them. In the experimental prototype, the adaptation involves lowering the quality of the media stream to a certain scale to save the outbound bandwidth but other adaptations are, of course, also possible.

4. Transformation may be invoked based on a variety of hints. In the prototype, the end-user application generates the hints by monitoring receiver behaviors. Active networks provide the mechanism by which an active node can maintain the receivers’ state and other receiver-oriented rate control information necessary to implement such functionality. Receiver behavior is captured based on detecting user interface (UI) actions (e.g. iconifying, shrinking, or hiding a window) as suggested in [KLB99] and this is used to provide feedback information to the active network nodes.

Taken together, these characteristics have shown that active networks have proven to be promising to supporting multimedia transmission, in terms of both efficiency and better QoS support. The ANTS-based prototype has also demonstrated that Active Networks technology should not only be credited for its programmable network layer, but also ability to integrate application oriented services with network services, for it may lead to a more "intelligent" network infrastructure, and provide the following benefits:

1. The adaptation is carried out at active nodes on the transmission path in accordance with both receivers' feedback/state information and, perhaps also, with current congestion status. The capsule can decide when and on which node to perform the transformation, so that the bandwidth used for media stream delivery can be minimized, while still ensuring all recipients get their desired QoS. Even when the receivers' QoS requirements are dynamically changing,

2. Since the transformation codes are part of the capsules' forwarding routines, the adaptation service may can be easily deployed on any active network node along the transmission path, where adaptation is necessary.

3. The response to QoS requirement changes can also be more prompt compared to application level adaptation approaches, because the feedback capsules can trigger the transformation at intermediate nodes while they traverse upstream to the sender. Also, the propagation of these feedback capsules will be stopped after the appropriate adaptation has been activated. This can save the upstream bandwidth.

6.1 Possible Improvements and Future Work

Some of the design choices made to simplify the prototype implementations make the project require further work and improvements in some areas. First, there are performance issues involved to make this prototype system practical. For the implementation, we have chosen to build this proof-of-concept prototype using ANTS toolkit. This Java-based active network framework facilitated and simplified the development work by providing a dynamic code distribution system and a consistent programming environment. However, the byte-code-based technology can not operate efficiently on processing-intensive network nodes, for the reason that each instruction has to be mapped (i.e. loaded, decoded, and invoked) by the interpreter. The interpretation time of byte-code applications are thus normally more than ten times longer than the execution of native machine code [KAC98]. Compilation in advance and just-in-time (JIT) compilation are the two major ways to boost java performance. Compilation in advance will translate capsules into native codes before injecting them into the network. However, compilation in advance can lead to unportable code, bigger capsule size and makes it impractical to validate each operation to protect against intentional or accidental attach on active nodes. In contrast, JIT translates Java byte code to native code during program execution. Compared with compilation in advance, JIT compilation retains the portability and security properties of byte-code, and can lead to a significant performance improvement (combined with native code caching) in Java based Active Networks [KPS02]. It would be useful to boost the performance in the final implementation.

Further, any general-purpose language should be avoided from being eventually chosen for composing active protocols, due to their relatively low performance and security concerns. Further studies [MHN01] have shown that using a well defined packet composition language, together with a its supportive infrastructure, such as service layer, configuration and operation platform, may lead to a practical active network system for general use. Thus, towards improving the performance to a practical level, the main focus could be concentrating on adopting or developing a more suitable active packet composition language while integrating the adaptive multicast service with a more mature active network platform.

On the expanding experimental Active Networks Backbone (ABone), Anetd provides for the use of multiple active network systems. The Anetd design philosophy is to be fully backward compatible with current networking software, and at the same time to support new innovative functionality in managing, designing and controlling Active Networks. Further, packets among Anetd's are exchanged in an encapsulated form using the ANEP protocol. It would be worth migrating the current prototype to the Anetd-based infrastructure to see how it performs in such an environment.

Second, in the implementation of the prototype, the powerful JMF library was used to simplify the programming of the media processing and maintain the focus of the thesis on the active network programming model. This imposes the requirement that the JMF class library must exist on all active nodes through which the capsules pass, which is practically infeasible. In conjunction with the performance issues, one possible approach that is worth considering is to abstract a set of media processing primitives, such as quantization and discrete cosine transform. Those primitives could then be programmed as lightweight, standardized service layer components, and deployed among all active nodes. With those optimized "service-let"s, the construction of active packets could be facilitated, and consequently, we might be able to see significant improvement in performance which would result in more practical applications.

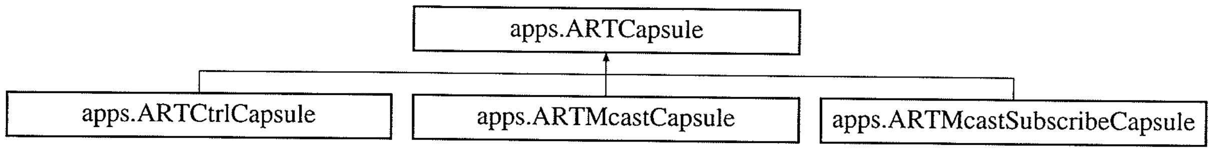
Appendix A

Source Code Documentation

This documentation does not include the sender and receiver applications, since they have a simple flow and are self-explanatory. Refer to the source and comments for implementation details.

A.1 ARTCapsule Class Reference

Inheritance diagram for ARTCapsule::



A.1.1 Detailed Description

ARTCapsule is the base class for all customized capsule types.

It defines constants and data types that are shared by all of its subclasses. The subclasses it defines include **HostAddresses**, **TrackRecord**, and **CacheRecord**.

A.1.2 Static Public Attributes

- final int **MAX_TRANSFORM** = 4
- final int **FULL_TRANSMISSION** = 3

The documentation for this class was generated from the following file:

- **ARTCapsule.java**

A.2 HostAddresses Class Reference

A.2.1 Detailed Description

HostAddress is just an array type to hold a list of host addresses.

Public Attributes

- int[] **hosts**
  The array of host addresses.

The documentation for this class was generated from the following file:

- **ARTCapsule.java**

A.3 TrackRecord Class Reference

A.3.1 Detailed Description

TrackRecord class is used to control the track based adaptation.

Public Attributes

- short **trackID**

  Each track has a unique ID.

- **ARTProcessor processor**

  Each track injects its own processor into the ANTS network with this field.

- **HostAddresses**[ ] **proc**

  The list of hosts to go through a transformation.

The documentation for this class was generated from the following file:

- **ARTCapsule.java**

A.4 CacheRecord Class Reference

A.4.1 Detailed Description

CacheRecord is the forwarding record cached in the soft-store of active nodes.

Public Attributes

- int[ ] **h**

  Array h holds the reverse pointers.

- long **timestamp**

  The timestamp to record time of the most recent update.

- **TrackRecord**[ ] **tr**

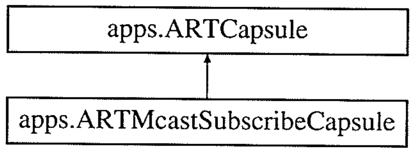
Array tr controls the adaptation of each track. See **TrackRecord** *for more details.*

The documentation for this class was generated from the following file:

- **ARTCapsule.java**

A.5 ARTMcastSubscribeCapsule Class Reference

Inheritance diagram for ARTMcastSubscribeCapsule::



A.5.1 Detailed Description

ARTMcastSubscribeCapsule does multicast group subscription for receivers.

Its forwarding routine (**evaluate()**) will set up the fowarding records (Cache-Record) at the active nodes it traverses. **ARTMcastSubscribeCapsule** is derived from **ARTCapsule**.

Public Attributes

- int **group**

  Group address.

- int **target**

  Sender address.

- int **reverse**

  The address of the downstream node from where this subscribe capsule is arrived.

- int **max_track**

  Maximum number of tracks.

- **TrackUpdate[] tu**

  Track update array.

A.5.2 Constructor & Destructor Documentation

A.5.2.1 ARTMcastSubscribeCapsule ()

Default constructor.

A.5.2.2 ARTMcastSubscribeCapsule (int *target*, int *group*)

This contructor is used to create a read-for-transmitting subscribe capsule.

Parameters:

target: target address (source address)

group: group address

A.5.3 Member Function Documentation

A.5.3.1 void BuildTrackUpdate (short *trackIDs*[], short *procs*[])

Re-build the track update array to be forwarded upstream.

Once the subscribe capsule has set up the track records by integrating the quality requirements it carries with the information in the node's current track records, the track update array has to be updated to reflect the change and then propagated upstream.

A.5.3.2 Xdr decode ()

Decoding the object from External Data Representation.

XDR has to be decoded so that the data it carries can be actually understood.

A.5.3.3 Xdr encode ()

Encode the object into External Data Representation.

A capsule has to be encoded into XDR before it can be sent out.

A.5.3.4 boolean evaluate (Node *n*)

The forwarding routine for the subscribe capsule.

Parameters:

n: The active node this capsule is currently being running on.

A.5.3.5 int length ()

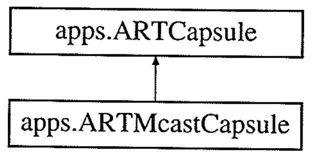
Length of the data part of the capsule.

The documentation for this class was generated from the following file:

- **ARTMcastSubscribeCapsule.java**

A.6 ARTMcastCapsule Class Reference

Inheritance diagram for ARTMcastCapsule::



A.6.1 Detailed Description

ARTMcastCapsule is the wrapper and carrier capsule for the media data packet.

It's a subclass of **ARTCapsule**.

Public Attributes

- int **group**

  Group address.

- int **target**

  Source address.

- short **spt**

  Source port.

- short **dpt**

  Destination port.

- short **trackID**

  The unique ID of the track.

- boolean **processed**

  Indicating if the data has already been processed.

- ByteArray **data**

  The actual data payload.

A.6.2 Constructor & Destructor Documentation

A.6.2.1 ARTMcastCapsule ()

Default constructor.

A.6.2.2 ARTMcastCapsule (int *ga*, int *ta*, short *sa*, short *da*, short *track*, ByteArray *d*)

This contructor is used to create a read-for-transmitting data capsule.

Parameters:

ga: group address

ta: target address (source address)

sa: source port

da: destination port

track: track ID

d: multicast data

A.6.3 Member Function Documentation

A.6.3.1 Xdr decode ()

Decoding the object from External Data Representation.

XDR has to be decoded so that the data it carries can be actually understood.

A.6.3.2 Xdr encode ()

Encode the object into External Data Representation.

A capsule has to be encoded into XDR before it can be sent out.

A.6.3.3 boolean evaluate (Node *n*)

The forwarding routine that will be executed on an active node.

Parameters:

n: The current node that the forwarding routine is being executed on.

A.6.3.4 ByteArray getData ()

Read the data payload.

A.6.3.5 short getDstPort ()

Get destination port.

A.6.3.6 short getSrcPort ()

Get source port.

A.6.3.7 int length ()

Get the length of the data part of this capsule type.

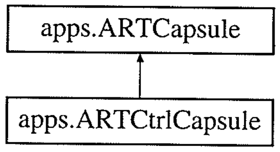
A.6.3.8 void resetSrc (int *source*)

Reset src address. The documentation for this class was generated from the following file:

- **ARTMcastCapsule.java**

A.7 ARTCtrlCapsule Class Reference

Inheritance diagram for ARTCtrlCapsule::



A.7.1 Detailed Description

ART CtrlCapsule conveys changes of quality requirement from receiver to sender, and initiates media processing when necessary.

Public Attributes

- int **group**

  Group address.

- int **target**

  Sender address.

- int **reverse**

The address of the downstream node from where this subscribe capsule is arrived.

- int **max_track**

  Maximum number of tracks.

- **TrackUpdate[ ] tu**

  Track update array.

A.7.2 Constructor & Destructor Documentation

A.7.2.1 ARTCtrlCapsule ()

Default contstructor.

A.7.2.2 ARTCtrlCapsule (int *target*, int *group*)

Constructor.

Parameters:

target: target address

group: group address

A.7.3 Member Function Documentation

A.7.3.1 void BuildTrackUpdate (short *trackIDs*[ ], short *procs*[ ])

A.7.3.2 Xdr decode ()

Encode the object into External Data Representation.

A capsule has to be encoded into XDR before it can be sent out.

A.7.3.3 Xdr encode ()

Encode the object into External Data Representation.

A capsule has to be encoded into XDR before it can be sent out.

A.7.3.4 boolean evaluate (Node *n*)

The forwarding routine for the control capsule.

Parameters:

n: The active node this capsule is currently being running on.

A.7.3.5 int length ()

Size of the capsule's data payload.

The documentation for this class was generated from the following file:

- **ARTCtrlCapsule.java**

A.8 ANTSHandler Class Reference

A.8.1 Detailed Description

ANTSHandler handles the encapsulation/de-encapsulation and the transmission of RTP packet in ANTS network.

Public Attributes

- SourceTransferHandler **outputHandler** = null
- Application **application**

The owner application.

- **ARTMcastCapsule cap**

  The active multicast capsule to be sent or received.

- int **mygroup**

  The multicast group address.

- int **mytarget**

  The source address.

- short **myport**

  Port number.

- boolean **IsSender**

  Marking the sender or receiver.

- short **trackID**

  The ID of the track whose data is to be handled.

A.8.2 Constructor & Destructor Documentation

A.8.2.1 ANTSHandler (Application *app*, int *group*, int *target*, short *hport*, short *track*)

Constructor of **ANTSHandler**.

ANTSHandler is created with both the source and destination addresses and ports, so that those information is not needed when doing actual sending and receiving.

Parameters:

app: the appcliation

group: the address of the active multicast group

target: the source address

hport: the source port

track: the track number at the sender. This parameter is -1 when **ANTSHandler** is created at receiver

A.8.3 Member Function Documentation

A.8.3.1 int getMinimumTransferSize ()

Get the minimum transfer size.

The minimum transfer size is the size of the data part of the multicast capsule

A.8.3.2 int read (byte *buffer*[ ], int *offset*, int *length*)

Read data from ANTS network.

Parameters:

buffer: The user buffer to receive data;

offset: The offset of the user buffer start from where the received data are written;

length: length of the buffer

A.8.3.3 void receiveCapsule (ARTMcastCapsule *cap*)

Receive a multicast Capsule.

Parameters:

cap: The user capsule to be filled with the received data.

A.8.3.4 int write (byte *buffer*[], int *offset*, int *length*)

Write data into ANTS network.

Parameters:

buffer: user data buffer to be sent

offset: the starting offset in the buffer

length: number of bytes to be sent

The documentation for this class was generated from the following file:

- **ANTSHandler.java**

A.9 ARTProcessor Class Reference

A.9.1 Detailed Description

ARTProcessor is the object that does the transformation.

It is injected into the active network nodes by the sender using the *processor* field of **TrackRecord**. ART uses its own **ANTSHandlerEx** class to bridge the RTP sesssion and ANTS network, because there's no way to inject **ANTSHandler** into the network along with **ARTProcessor**, since **ANTSHandler** is not a capsule type.

Public Attributes

- RTPSessionMgr **mgr** = null

Session manager for receiving from the sender.

- RTPSocket **rtpsock** = null

  RTP socket for receiving from the sender.

- **ANTSHandlerEx rtp** = null

  ANTSHandler *handles the RTP session between the processor and the sender.*

- **ANTSHandlerEx rtcp** = null

  ANTSHandler *handles the RTCP session between the processor and the sender.*

- RTPSessionMgr **smgr** = null

  Session manager for forwarding to the sender.

- RTPSocket **srtpsock** = null

  RTP socket for forwarding to the sender.

- **ANTSHandlerEx srtp** = null

  ANTSHandler *handles the RTP session between the processor and receiver.*

- **ANTSHandlerEx srtcp** = null

  ANTSHandler *handles the RTCP session between the processor and receiver.*

- Node **server** = null

  The node that the processor is running on.

- int **source**

  Source address.

- int **group**

Group address.

- int **target**

  Target address.

- short **port**

  Port number.

- short **trackID**

  The ID of the track this processor is processing.

A.9.2 Constructor & Destructor Documentation

A.9.2.1 ARTProcessor (Node *node*, int *src*, int *grp*, int *trgt*, short *pt*)

Parameters:

node: the current node

src: source address

grp: group address

trgt: target address

pt: port number

A.9.3 Member Function Documentation

A.9.3.1 void controllerUpdate (ControllerEvent *ce*)

Callback function to handle the state change of JMF processor.

Parameters:

ce: The event to be handled. See JMF documentation for more details

A.9.3.2 boolean initialize () `[protected]`

Initialize and get the JMF processor object ready for processing.

A.9.3.3 void receive (Capsule *cap*)

Receive capsule.

Parameters:

cap: The capsule to be filled with received data

A.9.3.4 void run ()

Start **ARTProcessor**.

A.9.3.5 void setQuality (Player *p*, float *val*)

Setting the encoding quality to the specified value on the encoder.

0.5 is a good default.

The documentation for this class was generated from the following file:

- **ARTProcessor.java**

Appendix B

Source Code

B.1 ARTCapsule.java

```
package apps;

import ants.*;

/**
 * ARTCapsule is the base class for all customized capsule types. It
    defines
 * constants and data types that are shared by all of its subclasses.
    The subclasses
 * it defines include HostAddresses, TrackRecord, and CacheRecord.
 */

public abstract class ARTCapsule extends Capsule {
  public static final int MAX_TRANSFORM = 4;
  public static final int FULL_TRANSMISSION = 3;

  /**
   * HostAddress is just an array type to hold a list of host
      addresses.
   */
  public class HostAddresses {
    //!The array of host addresses.
    public int[] hosts;
  }

  /**
   * TrackRecord class is used to control the track based adaptation.
   */
  public class TrackRecord {
        //!Each track has a unique ID.
    public short trackID;
        //!Each track has its own processor on an active node.
    public ARTProcessor processor;
        //!The list of hosts to go through a transformation
    public HostAddresses[] proc;
  }

  /**
   * CacheRecord is the forwarding record cached in the soft-store of
      active nodes
   */
  public class CacheRecord {
    //!Array h holds the reverse pointers.
    public int[] h;
        //!The timestamp to record time of the most recent update.
    public long timestamp;
        //!Array tr controls the adaptation of each track. See
            TrackRecord for more details.
    public TrackRecord[] tr;
  }
}
```

B.2 ARTCtrlCapsule.java

```
package apps;

import ants.*;
import ants.wrapper.*;

/**
 * ART CtrlCapsule conveys changes of quality requirement from
 * receiver to sender, and
 * initiates media processing when necessary.
 */
public class ARTCtrlCapsule extends ARTCapsule
{
  public class TrackUpdate{
    public short trackID;
    public short proc;
  }

  protected byte[] mid() { return findMID("apps.ARTCtrlCapsule"); }
  protected byte[] pid() { return findPID("apps.ARTCtrlCapsule"); }
protected byte[] gid() { return findGID("apps.ARTCtrlCapsule"); }

//!Group address
int group;
//!Sender address
int target;
//!The address of the downstream node from where this subscribe
   capsule is arrived.
int reverse;

//!Maximum number of tracks.
int max_track;

//!Track update array.
TrackUpdate[] tu;

//! Size of the capsule's data payload.
public int length() {
  int alen;
  if(tu == null)
    alen = 0;
  else
    alen = tu.length;
  return super.length() + Xdr.INT + Xdr.INT +
    Xdr.INT + Xdr.INT + alen * 2 * Xdr.SHORT;
}

//! Encode the object into External Data Representation.
/*!
 * A capsule has to be encoded into XDR before it can be sent out.
 */
public Xdr encode() {
  Xdr xdr = super.encode();

  xdr.PUT(target);
  xdr.PUT(group);
  xdr.PUT(reverse);
  if(tu == null)
    xdr.PUT(0);
  else {
    xdr.PUT(tu.length);
    for(int i = 0; i < tu.length; i++) {
      xdr.PUT(tu[i].trackID);
      xdr.PUT(tu[i].proc);
    }
  }
```

```
    return xdr;
  }

  //! Encode the object into External Data Representation.
  /*!
   * A capsule has to be encoded into XDR before it can be sent out.
   */
  public Xdr decode() {
    Xdr xdr = super.decode();

    target = xdr.INT();
    group = xdr.INT();
    reverse = xdr.INT();

    int len = xdr.INT();
    if(len >  0) {
      tu = new TrackUpdate[len];
      for(int i = 0; i < len; i++) {
        tu[i] = new TrackUpdate();
        tu[i].trackID = xdr.SHORT();
        tu[i].proc = xdr.SHORT();
      }
    }

    return xdr;
  }

  //!The forwarding routine for the control capsule
  /*!
   *\param n: The active node this capsule is currently being running
      on.
   */
  public boolean evaluate(Node n) {
    try {
      CacheRecord m = (CacheRecord)n.getCache().get(group, target);

      if (m == null) {
        //Control capsule do nothing when there's no cached record
            exist
        //The track record could be supplied with following subscribe
            caps.
        System.out.println("Ctrl m null");
        return true;
      }

      if(reverse != 0) {
        System.out.println("Reverse not 0");
```

```
boolean found = false;
for (int i = 0; m.h != null && i < m.h.length; i++)
  if (m.h[i] == reverse) {
    found = true;
    break;
  }

if(found) {
  //Reverse addr. has been added to the group. Since subscribe
  // capsules have been received, the reverse must exist some
  -
  // where in M.tr[]. Find it out and update it position.

  //Assert m.tr must not be null
  if(m.tr != null) {
    System.out.print("Ctrl--_m.tr_not_null");
    //Assert tu must not be null
    //For each recrod in self.tu[]
    for(int i = 0; i < tu.length; i++) {
      boolean jumpout = false;
      short proc = tu[i].proc;
      for(int j = 0; j < m.tr.length; j++)
        //Locate M.tr[j] with trackID = self.tu[i].trackID
        if(m.tr[j].trackID == tu[i].trackID) {
          if(m.tr[j].proc[proc].hosts == null) {
            //Just append reverse if it's null
            m.tr[j].proc[proc].hosts = new int[1];
            m.tr[j].proc[proc].hosts[0] = reverse;
          } else {
            //Else look for reverse in proc[proc]
            int[] h_target = m.tr[i].proc[proc].hosts;
            for(int k = 0; k < h_target.length; k++)
              if(h_target[k] == reverse) {
                jumpout = true;
                break;
              }
            if(jumpout)
              break;

            //Not in proc[], append to the list
            int t_len = h_target.length;
            int[] nht = new int[t_len + 1];
            System.arraycopy(h_target, 0, nht, 0, t_len);
            nht[t_len] = reverse;
           m.tr[i].proc[proc].hosts = nht;
        }
```

```
                    //Remove reverse from other proc[]'s if exists
                    //Right now just deal with proc[FULL_TRANSMISSION]
                    int indx = (proc == 3)? 0: 3;
                    int[] h_target = m.tr[i].proc[indx].hosts;
                    if(h_target == null)
                      break;
                    //Find reverse
                    for(int k = 0; k < h_target.length; k++)
                      if(h_target[k] == reverse) {
                        int[] nht = new int[h_target.length - 1];
                        int t =0;
                        for(int s = 0; s < h_target.length; s++)
                          if(s != k)
                            nht[t++] = h_target[s];
                        m.tr[i].proc[indx].hosts = nht;
                        break;
                      }
                    break;
                  }
              }

              //Build new self.tu, so that each trackID has the maximum
                  meaningful proc
              tu = new TrackUpdate[m.tr.length];
              //For each record in M.tr[]
              for(int t = 0; t < m.tr.length; t++) {
                tu[t] = new TrackUpdate();
                tu[t].trackID = m.tr[t].trackID;
                //Find maximum proc#, of which M->tr[i].proc[proc#] is
                    not null;
                if(m.tr[t].proc[FULL_TRANSMISSION] != null)
                  tu[t].proc = FULL_TRANSMISSION;
                else
                  tu[t].proc = 0;
              }
            } else
              System.out.println("NULL_TR");
          }
        }

        reverse = n.getAddress();

        if (n.getAddress() != target) {
          System.out.println("Route_to_next" + NodeAddress.toString(
              target));
          return n.routeForNode(this, target);
        }
```

```
      return true;
    }catch(Exception e){
      e.printStackTrace();
      return false;
    }
  }

  //! Default contstructor
  public ARTCtrlCapsule() {
    tu = null;
  }

  public void BuildTrackUpdate(short[] trackIDs, short[] procs) {
    if(trackIDs == null) {
      tu = null;
      return;
    }
    int t_len = trackIDs.length;
    tu = new TrackUpdate[t_len];
    for(int i = 0; i < trackIDs.length; i++) {
      tu[i] = new TrackUpdate();
      tu[i].trackID = trackIDs[i];
      tu[i].proc = procs[i];
    }
  }

  //! Constructor
  /*!
   *\param target: target address
   *\param group: group address
   */
  public ARTCtrlCapsule(int target, int group) {
    this.target = target;
    this.group = group;
    this.reverse = 0;
  }
}
```

B.3 ARTMcastProtocol.java

```
package apps;

import ants.*;

public class ARTMcastProtocol extends Protocol {
  public ARTMcastProtocol() throws Exception {
    startProtocolDefn();

    startGroupDefn();
    addCapsule("apps.ARTMcastCapsule");
    addCapsule("apps.ARTMcastSubscribeCapsule");
        addCapsule("apps.ARTCtrlCapsule");
        addCapsule("apps.ARTProcessor");
        endGroupDefn();

    endProtocolDefn();
  }
}
```

B.4 ARTMcastCapsule.java

```
package apps;

import ants.*;
import ants.wrapper.*;

/**
 * ARTMcastCapsule is the wrapper and carrier capsule for the media
    data packet.
 * The forwarding routine will set up forwarding records (CacheRecord)
     at the
 * active nodes it traverses.
 * It's a subclass of ARTCapsule.
 */
public class ARTMcastCapsule extends ARTCapsule
{
  protected byte[] mid() { return findMID("apps.ARTMcastCapsule"); }
  protected byte[] pid() { return findPID("apps.ARTMcastCapsule"); }
  protected byte[] gid() { return findGID("apps.ARTMcastCapsule"); }

  //!Group address.
  int group;
  //!Source address.
  int target;
  //!Source port.
  short spt;
  //!Destination port.
  short dpt;

  //!The unique ID of the track.
  public short trackID;
```

```
//!Indicating if the data has already been processed.
public boolean processed;
//!The actual data payload.
ByteArray data;

//!Get source port
public short getSrcPort() { return spt; }
//!Get destination port
public short getDstPort() { return dpt; }

//! Read the data payload
public ByteArray getData() {
  return data;
}

//!Get the length of the data part of this capsule type
public int length() {
  int s = Xdr.INT + Xdr.INT + Xdr.SHORT + Xdr.SHORT + Xdr.SHORT +
      Xdr.BYTEARRAY(data);
  return super.length() + s;
}

//! Encode the object into External Data Representation.
/*!
 * A capsule has to be encoded into XDR before it can be sent out.
 */
public Xdr encode() {
  Xdr xdr = super.encode();
  xdr.PUT(group);
  xdr.PUT(target);
  xdr.PUT(spt);
  xdr.PUT(dpt);
  xdr.PUT(trackID);
  xdr.PUT(data);

  return xdr;
}

//! Decoding the object from External Data Representation.
/*!
 * XDR has to be decoded so that the data it carries can be actually
     understood.
 */
public Xdr decode() {
  Xdr xdr = super.decode();
  group = xdr.INT();
  target = xdr.INT();
```

```
    spt = xdr.SHORT();
    dpt = xdr.SHORT();
    trackID = xdr.SHORT();
    data = xdr.BYTEARRAY();

    return xdr;
  }

  //! The forwarding routine that will be executed on an active node
  /*!
   *\param n: The current node that the forwarding routine is being
       executed on.
   */
  public boolean evaluate(Node n) {
    int h_len;
    TrackRecord new_tr = null;
    CacheRecord m = (CacheRecord)n.getCache().get(group, target);
    if(m != null) {
      if(m.h == null) {
        n.deliverToApp(this, dpt);
        return true;
      } else
        System.err.println("H_is_not_NULL");
      boolean found = false;
      for(int i = 0; m.tr != null && i < m.tr.length; i++)
        if(m.tr[i].trackID == trackID) {
          new_tr = m.tr[i];
          found = true;
          break;
        }

      if(!found) {
        new_tr = new TrackRecord();
        new_tr.trackID = trackID;
        new_tr.processor = new ARTProcessor(n, src, group, target, dpt
            );
        new_tr.processor.trackID = trackID;
        new Thread(new_tr.processor).start();
        new_tr.proc = new HostAddresses[MAX_TRANSFORM];
        for(int i = 0; i < MAX_TRANSFORM; i++)
          new_tr.proc[i] = new HostAddresses();
        if(m.h != null) {
          h_len = m.h.length;
          int[] temp = new int[h_len];
          System.arraycopy(m.h, 0, temp, 0, h_len);
          new_tr.proc[FULL_TRANSMISSION].hosts = temp;
          System.out.println("Set_to_Full_Transmission");
          }

          //Append new_tr to M.tr
          if(m.tr == null) {
            m.tr = new TrackRecord[1];
            m.tr[0] = new_tr;
          } else {
            int tr_len = m.tr.length;

            TrackRecord[] ntr = new TrackRecord[tr_len + 1];
            System.arraycopy(m.tr, 0, ntr, 0, tr_len);
            ntr[tr_len] = new_tr;
            m.tr = ntr;
          }
        }

        //Route self to all hosts in M.tr[i].HostAddresses[]
        if(!processed) {
          HostAddresses target = new_tr.proc[FULL_TRANSMISSION];
          for(int i = 0; i < target.hosts.length; i++)
            n.routeForNode(this, target.hosts[i]);
        } else
          new_tr.processor.receive(this);
      }
      return true;
    }

    //! Default constructor
    public ARTMcastCapsule() {
      processed = false;
    }

    //! This contructor is used to create a read-for-transmitting data
        capsule.
    /*!
     * \param ga: group address
     * \param ta: target address (source address)
     * \param sa: source port
     * \param da: destination port
     * \param track: track ID
     * \param d: multicast data
     */
    public ARTMcastCapsule(int ga, int ta, short sa, short da,
        short track, ByteArray d) {
      group = ga;
      target = ta;
      spt = sa;
    dpt = da;
    data = d;
    trackID = track;
    processed = false;
  }

  //!Reset src address
  public void resetSrc(int source) {
    src = source;
    resources = 64;
  }
}
```

B.5 ARTMcastSubscribeCapsule.java

```
package apps;

import ants.*;
import ants.wrapper.*;

/**
 * ARTMcastSubscribeCapsule does multicast group subscription for
    receivers.
 * Its forwarding routine (evaluate()) will set up the fowarding
    records
 * (CacheRecord) at the active nodes it traverses.
 * ARTMcastSubscribeCapsule is derived from ARTCapsule.
 */
public class ARTMcastSubscribeCapsule extends ARTCapsule
{
  public class TrackUpdate{
    public short trackID;
    public short proc;
  }

  protected byte[] mid() { return findMID("apps.
     ARTMcastSubscribeCapsule"); }
  protected byte[] pid() { return findPID("apps.
     ARTMcastSubscribeCapsule"); }
  protected byte[] gid() { return findGID("apps.
     ARTMcasSubscribeCapsule"); }

  //!Group address.
  int group;
  //!Source address.
  int target;
```

```
//!The address of the downstream node from where this subscribe
   capsule is arrived.
int reverse;

//!Maximum number of track
int max_track;

//!Track update array
TrackUpdate[] tu;

//! Length of the data part of the capsule
public int length() {
  int alen;
  if(tu == null)
    alen = 0;
  else
    alen = tu.length;
  return super.length() + Xdr.INT + Xdr.INT +
    Xdr.INT + Xdr.INT + alen * 2 * Xdr.SHORT;
}

//! Encode the object into External Data Representation.
/*!
 * A capsule has to be encoded into XDR before it can be sent out.
 */
public Xdr encode() {
  Xdr xdr = super.encode();

  xdr.PUT(target);
  xdr.PUT(group);
  xdr.PUT(reverse);
  if(tu == null)
    xdr.PUT(0);
  else {
    xdr.PUT(tu.length);
    for(int i = 0; i < tu.length; i++) {
      xdr.PUT(tu[i].trackID);
      xdr.PUT(tu[i].proc);
    }
  }

  return xdr;
}

//! Decoding the object from External Data Representation.
/*!
 * XDR has to be decoded so that the data it carries can be actually
```

```
      understood.
  */
 public Xdr decode() {
   Xdr xdr = super.decode();

   target = xdr.INT();
   group = xdr.INT();
   reverse = xdr.INT();

   int len = xdr.INT();
   if(len >  0) {
     tu = new TrackUpdate[len];
     for(int i = 0; i < len; i++) {
       tu[i] = new TrackUpdate();
       tu[i].trackID = xdr.SHORT();
       tu[i].proc = xdr.SHORT();
     }
   }

   return xdr;
 }


 //!The forwarding routine for the subscribe capsule
 /*!
  *\param n: The active node this capsule is currently being running
      on.
  */
 public boolean evaluate(Node n) {
   try {
   CacheRecord m = (CacheRecord)n.getCache().get(group, target);

   if (m== null) {
     // n.log("no info at node "+ NodeAddress.toString(n.getAddress()
        ));
     System.out.println("M_NULL");
     m = new CacheRecord();
     m.h = null;
     n.getCache().put(group, target , m, 20);
   } else {
     // n.log("found something at node "+ NodeAddress.toString(n.
        getAddress()));
   }

   if(reverse != 0) {
     System.err.println("REVERSE_NOT_NULL");
     boolean found = false;
```

```
for (int i = 0; m.h != null && i < m.h.length; i++)
  if (m.h[i] == reverse) {
    found = true;
    break;
  }

if(!found) {
  int h_len;
  if(m.h == null)
    h_len = 0;
  else
    h_len = m.h.length;

  int[] nh = new int[h_len + 1];
  if(m.h != null)
    System.arraycopy(m.h, 0, nh, 0, h_len);
  nh[h_len] = reverse;
  m.h = nh;

  if(m.tr != null) {
    if(tu == null) {
      //Append self.reverse to M.tr[i].proc[FULL_TRANSMISSION].
          hosts
      System.out.println("NULL_TU");
      for(int i = 0; i < m.tr.length; i++) {
        int[] target = m.tr[i].proc[FULL_TRANSMISSION].hosts;
        if(target == null) {
          target = new int[1];
          target[0] = reverse;
        } else {
          int t_len = target.length;
          int[] nht = new int[t_len + 1];
          System.arraycopy(target, 0, nht, 0, t_len);
          nht[t_len] = reverse;
        }
        m.tr[i].proc[FULL_TRANSMISSION].hosts = target;
      }
    } else {
      //For each recrod in self.tu[]
      for(int i = 0; i < tu.length; i++) {
        for(int j = 0; j < m.tr.length; j++)
          //Locate M.tr[j] with trackID = self.tu[i].trackID
          if(m.tr[j].trackID == tu[i].trackID) {
            //Append self.reverse to M.tr[j].proc[self.tu[i].
                proc]
            int[] target = m.tr[j].proc[tu[i].proc].hosts;
            System.out.println(tu[i].trackID + "-" + tu[i].proc)
```

```
                    ;
                if(target == null) {
                  target = new int[1];
                  target[0] = reverse;
                } else {
                  int t_len = target.length;
                  int[] nht = new int[t_len + 1];
                  System.arraycopy(target, 0, nht, 0, t_len);
                  nht[t_len] = reverse;
                }
                m.tr[i].proc[FULL_TRANSMISSION].hosts = target;
              }
            }
          }

          //Build new self.tu, so that each trackID has the maximum
              meaningful proc
          tu = new TrackUpdate[m.tr.length];
          //For each record in M.tr[]
          for(int t = 0; t < m.tr.length; t++) {
            tu[t] = new TrackUpdate();
            tu[t].trackID = m.tr[t].trackID;
            //Find maximum proc#, of which M->tr[i].proc[proc#] is not
                null;
            if(m.tr[t].proc[FULL_TRANSMISSION] != null)
              tu[t].proc = FULL_TRANSMISSION;
            else
              tu[t].proc = 0;
          }
        } else
          System.out.println("NULL_TR");
      }
    } else
      System.err.println("REVERSE_NULL");

    reverse = n.getAddress();
    System.err.println("REVERSE_SET");
    if (n.time() - m.timestamp < 1000)
      return true;
    else
      m.timestamp = n.time();

    if (n.getAddress() != target) {
      // n.log("onward to "+ NodeAddress.toString(target));
      return n.routeForNode(this, target);
    }
    return true;
```

```
      }catch(Exception e){
          e.printStackTrace();
          return false;
        }
    }

    //! Default constructor
    public ARTMcastSubscribeCapsule() {
      tu = null;
    }

    //!Re-build the track update array to be forwarded upstream.
    /*!
     * Once the subscribe capsule has set up the track records by
        integrating
     * the quality requirements it carries with the information in the
        node's
     * current track records, the track update array has to be updated
        to reflect
     * the change and then propagated upstream.
     */
    public void BuildTrackUpdate(short[] trackIDs, short[] procs) {
      if(trackIDs == null) {
        tu = null;
        return;
      }
      int t_len = trackIDs.length;
      tu = new TrackUpdate[t_len];
      for(int i = 0; i < trackIDs.length; i++) {
        tu[i] = new TrackUpdate();
        tu[i].trackID = trackIDs[i];
        tu[i].proc = procs[i];
      }
    }

    //! This contructor is used to create a read-for-transmitting
       subscribe capsule.
    /*!
     * \param target: target address (source address)
     * \param group: group address
     */
    public ARTMcastSubscribeCapsule(int target, int group) {
      this.target = target;
      this.group = group;
      this.reverse = 0;
    }
}
```

B.6 ARTProcessor.java

```
package apps;

import java.net.*;

import javax.media.*;
import javax.media.rtp.*;
import javax.media.rtp.event.*;
import javax.media.rtp.rtcp.*;
import javax.media.protocol.*;
import javax.media.control.*;
import com.sun.media.rtp.*;

import ants.*;

/**
 * ARTProcessor is the object that does the transformation.
 * It is injected into the active network nodes by the sender
 * using the processor field of TrackRecord. ART uses
 * its own ANTSHandlerEx class to bridge the RTP sesssion and ANTS
 * network, because there's no way to inject ANTSHandler into the
   network
 * along with ARTProcessor, since ANTSHandler is not a capsule type.
 */
public class ARTProcessor
  implements ReceiveStreamListener, SessionListener,
  ControllerListener, Runnable
{
  //!Session manager for receiving from the sender.
  RTPSessionMgr mgr = null;
  //!RTP socket for receiving from the sender.
  RTPSocket rtpsock = null;
  //!ANTSHandler handles the RTP session between the processor and the
      sender.
  ANTSHandlerEx rtp = null;
  //!ANTSHandler handles the RTCP session between the processor and
     the sender.
  ANTSHandlerEx rtcp = null;

  //!Session manager for forwarding to the sender.
  RTPSessionMgr smgr = null;
  //!RTP socket for forwarding to the sender.
  RTPSocket srtpsock = null;
  //!ANTSHandler handles the RTP session between the processor and
     receiver.
  ANTSHandlerEx srtp = null;
```

```
//!ANTSHandler handles the RTCP session between the processor and
    receiver.
ANTSHandlerEx srtcp = null;

//!The node that the processor is running on.
Node server = null;

private DataSource dataOutput = null;

//!Source address.
int source;
//!Group address.
int group;
//!Target address.
int target;
//!Port number.
short port;
//!The ID of the track this processor is processing.
public short trackID;

boolean sessionStarted = false;
boolean dataReceived = false;
Object dataSync = new Object();

protected byte[] mid() { return findMID("apps.ARTProcessor"); }
protected byte[] pid() { return findPID("apps.ARTProcessor"); }
protected byte[] gid() { return findGID("apps.ARTProcessor"); }

/**
 * The inline verision of ANTSHandler. The processor capsule relies
    on the
 * functionality of ANTSHandler to bridge RTP and active network.
    Since the
 * orignial ANTSHandler is not a capsule type, it can not be
    injected into
 * the network along with the processor capsule. This inline
    ANTSHandlerEx does
 * exactly what the ANTSHandler does
 */
public class ANTSHandlerEx implements PushSourceStream,
   OutputDataStream{
  SourceTransferHandler outputHandler = null;
  Node node;

  ARTMcastCapsule cap;
  ARTMcastCapsule tcap;
  int mysrc;
```

```
    int mygroup;
    int mytarget;
    short myport;
    boolean readonly;

    boolean IsSender;

    //! Constructor
        /*!
         *\param n: the current active node
         *\param src: the source address
         *\param group: the group address
         *\param target: the target address
         *\param hport: the port number
         *\param ro: readonly flag
         */
    public ANTSHandlerEx(Node n, int src, int group, int target,
      short hport, boolean ro){
      node = n;
      mysrc = src;
      mygroup = group;
      mytarget = target;
      myport = hport;
      readonly = ro;
      if(!readonly)
        tcap = new ARTMcastCapsule();
    }

    public void receiveCapsule(ARTMcastCapsule cap) {
      this.cap = cap;
      tcap.prime(cap);
      tcap.trackID = cap.trackID;
      if(outputHandler != null)
        outputHandler.transferData(this);
    }

    public Object[] getControls() {
      return new Object[0];
    }

    public Object getControl(String controlName) {
      return null;
    }

    public ContentDescriptor getContentDescriptor(){
      return null;
    }
    public long getContentLength(){
      return SourceStream.LENGTH_UNKNOWN;
    }

    public boolean endOfStream(){
      return false;
    }

    //Read from ANTS
    public int read(byte buffer[],
      int offset,
      int length){
      if(cap == null)
        System.err.println("READ_NULL_CAP");
      cap.getData().getBytes(0, buffer, offset, length);
      //cap = null;
      return length;
    }

    public int getMinimumTransferSize(){
      return cap.getData().length();
    }

    synchronized public void setTransferHandler(SourceTransferHandler
        transferHandler){
      this.outputHandler = transferHandler;
      System.err.println("outputHandler set!");
    }

    public int write(byte[] buffer,
      int offset, int length){
      if(!readonly) {
        ARTMcastCapsule rc = new ARTMcastCapsule(mygroup, mytarget,
          myport, myport, trackID, new ByteArray(buffer, offset,
              length));
        rc.resetSrc(source);
        rc.processed = true;
        node.send(rc);
      }
      return length;
    }
  }

//! Constructor
/**
 *\param node: the current node
```

```
   *\param src: source address
   *\param grp: group address
   *\param trgt: target address
   *\param pt: port number
   */
  public ARTProcessor(Node node, int src, int grp, int trgt, short pt)
      {
    server = node;
    source = src;
    group = grp;
    target = trgt;
    port = pt;
  }

  //! Initialize and get the JMF processor object ready for processing
     .
  protected boolean initialize() {
    try {
      InetAddress ipAddr;
      SendStream sendStream;
      SessionAddress localAddr = new SessionAddress();
      SessionAddress destAddr;

      rtpsock = new RTPSocket();
      //ANTSHandlerEx does not use the group and target address. It
          routes capsules
      //using the information provided by them.
      rtp = new ANTSHandlerEx(server, source, group, target, port,
          false);
      //rtcp is just a dumb stub without any use. RTCP capsules in not
           covered by
      //the process
      rtcp = new ANTSHandlerEx(server, source, group, target, port,
          true);

      rtpsock.setInputStream(rtp);
      rtpsock.setOutputStream(rtp);
      RTPPushDataSource rtcpsource;
      rtcpsource = rtpsock.getControlChannel();
      rtcpsource.setOutputStream(rtcp);
      rtcpsource.setInputStream(rtcp);

      rtpsock.connect();
      mgr = new RTPSessionMgr(rtpsock);
      mgr.addSessionListener(this);
      mgr.addReceiveStreamListener(this);
      ipAddr = InetAddress.getByName("127.0.0.1");
      destAddr = new SessionAddress(ipAddr, port,
            ipAddr, port + 1);
      mgr.initSession(localAddr, getSDES(mgr), .05, .25);

      // You can try out some other buffer size to see
      // if you can get better smoothness.
      BufferControl bc = (BufferControl)mgr.getControl("javax.media.
         control.BufferControl");
      if (bc != null)
        bc.setBufferLength(350);

      mgr.startSession(destAddr, 1, null);
      sessionStarted = true;
      System.err.println("Session_Started");
    } catch (Exception e) { System.err.println("EXCEPTION_HERE");}

    return true;
  }

  private String createTransmitter() {
    try {
      InetAddress ipAddr;
      SendStream sendStream;
      SessionAddress localAddr = new SessionAddress();
      SessionAddress destAddr;

      System.err.println("CREATING_TRANSMITTER!*****");
      // Cheated.  Should have checked the type.
      PushBufferDataSource pbds = (PushBufferDataSource)dataOutput;
      PushBufferStream pbss[] = pbds.getStreams();

      smgr = new com.sun.media.rtp.RTPSessionMgr();
      srtpsock = new RTPSocket();
      srtp = new ANTSHandlerEx(server, source, group, target, port,
         false);
      //rtcp is just a dumb stub without any use. RTCP capsules in not
           covered by
      //the process
      srtcp = new ANTSHandlerEx(server, source, group, target, port,
         true);

      srtpsock.setInputStream(srtp);
      srtpsock.setOutputStream(srtp);
      RTPPushDataSource rtcpsource;
      rtcpsource = srtpsock.getControlChannel();
      rtcpsource.setOutputStream(srtcp);
      rtcpsource.setInputStream(srtcp);

      System.out.println("Creating RTP session " + pbss.length);

      srtpsock.connect();
      smgr = new RTPSessionMgr(srtpsock);

      System.out.println("RTP Session Initialized");
      ipAddr = InetAddress.getByName("127.0.0.1");
      destAddr = new SessionAddress(ipAddr, port,
          ipAddr, port + 1);
      smgr.initSession(localAddr, getSDES(mgr), .05, .25);

      smgr.startSession(destAddr, 1, null);
      sendStream = smgr.createSendStream(dataOutput, 0);
      sendStream.start();
      System.err.println("Send stream started");
    } catch (Exception e) {}

    return null;
  }

  String cname = null;
  private SourceDescription[] getSDES(SessionManager mgr) {
    SourceDescription[] desclist = new SourceDescription[3];
    if (cname == null)
      cname = mgr.generateCNAME();

    desclist[0] = new SourceDescription(SourceDescription.
        SOURCE_DESC_NAME,
              System.getProperty("user.name"),
              1,
              false);
    desclist[1] = new SourceDescription(SourceDescription.
        SOURCE_DESC_CNAME,
              cname,
              1,
              false);
    desclist[2] = new SourceDescription(SourceDescription.
        SOURCE_DESC_TOOL,
              "ANTS Receivepowered by JMF",
              1,
              false);
    return desclist;
  }
```

```
public boolean isDone() {
  return false;
}


/**
 * SessionListener.
 */
public synchronized void update(SessionEvent evt) {
  if (evt instanceof NewParticipantEvent) {
    Participant p = ((NewParticipantEvent)evt).getParticipant();
    System.err.println("  - A new participant had just joined: " + p
        .getCNAME());
  }
}


/**
 * ReceiveStreamListener
 */
public synchronized void update( ReceiveStreamEvent evt) {
  System.err.println("******* Update");
  SessionManager mgr = (SessionManager)evt.getSource();
  Participant participant = evt.getParticipant(); // could be null.
  ReceiveStream stream = evt.getReceiveStream(); // could be null.

  if (evt instanceof RemotePayloadChangeEvent) {
    System.err.println("  - Received an RTP PayloadChangeEvent.");
    System.err.println("Sorry, cannot handle payload change.");
    System.exit(0);
  } else if (evt instanceof NewReceiveStreamEvent) {
    try {
      stream = ((NewReceiveStreamEvent)evt).getReceiveStream();
      DataSource ds = stream.getDataSource();

      // Find out the formats.
      RTPControl ctl = (RTPControl)ds.getControl("javax.media.rtp.
          RTPControl");
      if (ctl != null){
        System.err.println("  - Recevied new RTP stream: " + ctl.
            getFormat());
      } else
        System.err.println("  - Recevied new RTP stream");

      if (participant == null)
        System.err.println("    The sender of this stream had yet to
            be identified.");
          else {
            System.err.println("␣␣␣␣The␣stream␣comes␣from:␣" +
                participant.getCNAME());
          }

          // create a player by passing datasource to the Media Manager
          Processor p = javax.media.Manager.createProcessor(ds);
          if (p == null) {
            System.err.println("Processor␣NULL");
            return;
          } else
            System.err.println("Processor␣Created");

          p.addControllerListener(this);
          p.configure();

        } catch (Exception e) {
          System.err.println("NewReceiveStreamEvent␣exception␣" + e.
              getMessage());
          return;
        }

      } else if (evt instanceof StreamMappedEvent) {
        if (stream != null && stream.getDataSource() != null) {
          DataSource ds = stream.getDataSource();
          // Find out the formats.
          RTPControl ctl = (RTPControl)ds.getControl("javax.media.rtp.
              RTPControl");
          System.err.println("␣␣-␣The␣previously␣unidentified␣stream␣");
          if (ctl != null)
            System.err.println("␣␣␣␣" + ctl.getFormat());
          System.err.println("␣␣␣␣had␣now␣been␣identified␣as␣sent␣by:␣"
              + participant.getCNAME());
        }
      } else if (evt instanceof ByeEvent) {
        System.err.println("␣␣-␣Got␣\"bye\"␣from:␣" + participant.
            getCNAME());
        //Supposed to close the playerwindow
      }
    }


  /**
   * Setting the encoding quality to the specified value on the
       encoder.
   * 0.5 is a good default.
   */
```

```
void setQuality(Player p, float val) {
  Control cs[] = p.getControls();
  QualityControl qc = null;

  for (int i = 0; i < cs.length; i++) {
    if (cs[i] instanceof QualityControl &&
      cs[i] instanceof Owned) {
      Object owner = ((Owned)cs[i]).getOwner();

      // Check to see if the owner is a Codec.
      // Then check for the output format.
      if (owner instanceof Codec) {
        qc = (QualityControl)cs[i];
        qc.setQuality(val);
        System.err.println("-_Setting_quality_to_" +
               val + "_on_" + qc);
        break;
      }

    }
  }
}

//! Callback function to handle the state change of JMF processor
/*!
 *\param ce: The event to be handled. See JMF documentation for more
      details
 */
public void controllerUpdate(ControllerEvent ce) {
  System.err.println("Controller_UPDATE");
  System.err.println(ce.toString());

  Processor p = (Processor)ce.getSourceController();

  if (p == null) {
    System.err.println("Processor_NULL");
    return;
  }

  if (ce instanceof ConfigureCompleteEvent) {
    System.out.println("Processor_CONFIGURED");
    TrackControl[] tracks = p.getTrackControls();
    if(tracks == null || tracks.length < 1) {
      System.err.println("CAN'T_FIND_ANY_TRACK");
      return;
    }
```

```
          ContentDescriptor cd = new ContentDescriptor(ContentDescriptor.
              RAW_RTP);
          p.setContentDescriptor(cd);

          p.realize();
        } else if (ce instanceof RealizeCompleteEvent) {
          System.out.println("Processor␣REALIZED");
          setQuality(p, (float).7);
          dataOutput = p.getDataOutput();
          String result = createTransmitter();
          if (result != null) {
            p.close();
            System.err.println(result);
          }

          p.start();
        }

        if (ce instanceof ControllerErrorEvent) {
          System.err.println("Processor␣internal␣error:␣" + ce);
        }
      }

      //! Receive capsule.
      /*!
       *\param cap: The capsule to be filled with received data
       */
      public void receive(Capsule cap) {
        if(!(cap instanceof ARTMcastCapsule))
          return;

        ARTMcastCapsule mcap = (ARTMcastCapsule)cap;
        if(sessionStarted)
          rtp.receiveCapsule(mcap);
      }

      //! Start ARTProcessor
      public void run() {
        if (!initialize()) {
          System.err.println("Failed␣to␣initialize␣the␣sessions.");
          System.exit(-1);
        }

        // Check to see if AVReceive is done.
        try {
          while (!isDone())
          Thread.sleep(1000);
```

```
    } catch (Exception e) {}

    System.err.println("Exiting_ARTProcessing");
  }

}// end of ARTProcessor
```

B.7 ANTSRTCPHandler.java

```
package apps;

import javax.media.rtp.*;
import javax.media.*;
import javax.media.protocol.*;
import java.io.*;
import java.net.*;
import com.sun.media.ui.*;

import ants.*;
import utils.*;

public class ANTSRTCPHandler implements PushSourceStream,
    OutputDataStream{
  SourceTransferHandler outputHandler = null;
  Application application;
  ARTMcastCapsule cap;
  int mygroup;
  int mytarget;
  short myport;

  boolean IsSender;
  short trackID;

  // track indicates the track # at sender. For receiver track = -1,
      as it
  // should be got from the sender
  public ANTSRTCPHandler(Application app, int group, int target,
    short hport, short track){
    application = app;
    mygroup = group;
    mytarget = target;
    myport = hport;

    if(track < 0) {
      IsSender = false;
      System.out.println("Receiver_Here");
      } else {
        IsSender = true;
        trackID = track;
        System.out.println("Sender␣Here");
      }
    }

    public void receiveCapsule(ARTMcastCapsule cap) {
      this.cap = cap;
      if(outputHandler != null)
        outputHandler.transferData(this);
    }

    public Object[] getControls() {
      return new Object[0];
    }

    public Object getControl(String controlName) {
      return null;
    }

    public ContentDescriptor getContentDescriptor(){
      return null;
    }

    public long getContentLength(){
      return SourceStream.LENGTH_UNKNOWN;
    }

    public boolean endOfStream(){
      return false;
    }

    //Read from ANTS
    public int read(byte buffer[],
        int offset,
        int length){
        cap.getData().getBytes(0, buffer, offset, length);
/*      System.arraycopy(cap.getData(),
         0,
         buffer,
         offset,
         cap.getData().length());
      System.out.println("Read************");*/
      return length;
    }
```

```
  public int getMinimumTransferSize(){
    return cap.getData().length();
  }

  public void setTransferHandler(SourceTransferHandler transferHandler
     ){
    this.outputHandler = transferHandler;
  }

  //Write data to ANTS
  public int write(byte[] buffer,
      int offset, int length){
    /*ARTMcastCapsule rc = new ARTMcastCapsule(mygroup, mytarget,
      myport, myport, trackID, new ByteArray(buffer, offset, length));
    application.send(rc);*/
    return length;
  }
}
```

B.8 ANTSHandler.java

```
package apps;

import javax.media.rtp.*;
import javax.media.*;
import javax.media.protocol.*;
import java.io.*;
import java.net.*;
import com.sun.media.ui.*;

import ants.*;
import utils.*;

/**
 * ANTSHandler handles the encapsulation/de-encapsulation and the
    transmission
 * of RTP packet in ANTS network.
 **/
public class ANTSHandler implements PushSourceStream, OutputDataStream
   {
  SourceTransferHandler outputHandler = null;
  //!The owner application
  Application application;
  //!The active multicast capsule to be sent or received
  ARTMcastCapsule cap;
  //!The multicast group address
  int mygroup;
  //!The source address
  int mytarget;
  //!Port number
  short myport;

  //!Marking the sender or receiver.
  boolean IsSender;
  //!The ID of the track whose data is to be handled
  short trackID;

  //! Constructor of ANTSHandler
  /*!
    ANTSHandler is created with both the source and destination
        addresses and
        ports, so that those information is not needed when doing
            actual sending
        and receiving.
    \param app: the appcliation
        \param group: the address of the active multicast group
        \param target: the source address
        \param hport: the source port
        \param track: the track number at the sender. This parameter
            is -1 when ANTSHandler is created at receiver
  */
  public ANTSHandler(Application app, int group, int target,
    short hport, short track){
    application = app;
    mygroup = group;
    mytarget = target;
    myport = hport;

    if(track < 0) {
      IsSender = false;
      System.out.println("Receiver_Here");
    } else {
      IsSender = true;
      trackID = track;
      System.out.println("Sender_Here");
    }
  }

//!Receive a multicast Capsule
/*!
 *\param cap: The user capsule to be filled with the received data.
 */
public void receiveCapsule(ARTMcastCapsule cap) {
    this.cap = cap;
    if(outputHandler != null)
      outputHandler.transferData(this);
  }

  // Not applicable
  public Object[] getControls() {
    return new Object[0];
  }

  //! Not applicable
  public Object getControl(String controlName) {
    return null;
  }

  //! Not applicable
  public ContentDescriptor getContentDescriptor(){
    return null;
  }

  //! Not applicable
  public long getContentLength(){
    return SourceStream.LENGTH_UNKNOWN;
  }

  //! Not applicable
  public boolean endOfStream(){
    return false;
  }

  //!Read data from ANTS network
  /*!
        \param buffer: The user buffer to receive data;
        \param offset: The offset of the user buffer start from where
            the received data are written;
        \param length: length of the buffer
  */
  public int read(byte buffer[],
      int offset,
      int length){
    cap.getData().getBytes(0, buffer, offset, length);
    return length;
  }

  //!Get the minimum transfer size
  /*!
   * The minimum transfer size is the size of the data part of the
```

```
        multicast capsule
      */
    public int getMinimumTransferSize(){
      return cap.getData().length();
    }

    public void setTransferHandler(SourceTransferHandler transferHandler
        ){
      this.outputHandler = transferHandler;
    }

    //! Write data into ANTS network
    /*!
      \param buffer: user data buffer to be sent
          \param offset: the starting offset in the buffer
          \param length: number of bytes to be sent
    */
    public int write(byte[] buffer,
        int offset, int length){
      ARTMcastCapsule rc = new ARTMcastCapsule(mygroup, mytarget,
        myport, myport, trackID, new ByteArray(buffer, offset, length));
      application.send(rc);
      return length;
    }
}
```

B.9 ANTSTransmit.java

```
package apps;

import java.awt.*;
import java.io.*;
import java.net.*;
import java.net.InetAddress;
import javax.media.*;
import javax.media.protocol.*;
import javax.media.format.*;
import javax.media.control.TrackControl;
import javax.media.control.QualityControl;
import javax.media.rtp.*;
import javax.media.rtp.rtcp.*;
import com.sun.media.rtp.*;

import ants.*;
import utils.*;
```

```
/**
 * ANTSTransmit is the sender application.
 */
public class ANTSTransmit extends Application {

  // Input MediaLocator
  // Can be a file or http or capture source
  private String MediaSource;
  private MediaLocator locator;
  private String ipAddress;
  private int group;
  private int target;
  private short portBase;

  private Processor processor = null;
  private SessionManager rtpMgrs[];

  // My codes here
  private RTPSocket rtpsocks[];
  private ANTSHandler rtp[];
  private ANTSHandler rtcp[];

  private DataSource dataOutput = null;
  private  int maxsize = 2000;


  /**
   * Starts the transmission. Returns null if transmission started ok.
   * Otherwise it returns a string with the reason why the setup
      failed.
   */
  public synchronized String go() {
    String result;

  // Create a processor for the specified media locator
  // and program it to output JPEG/RTP
    result = createProcessor();
    if (result != null)
      return result;

  // Create an RTP session to transmit the output of the
  // processor to the specified IP address and port no.
    result = createTransmitter();
    if (result != null) {
      processor.close();
      processor = null;
      return result;
    }

    // Start the transmission
    processor.start();

    return null;
  }

  /**
   * Stops the transmission if already started
   */
  public void stop() {
    synchronized (this) {
      if (processor != null) {
      processor.stop();
      processor.close();
      processor = null;
      for (int i = 0; i < rtpMgrs.length; i++)
        rtpMgrs[i].closeSession("Session_ended");
      }
    }
  }

  private String createProcessor() {
    if (locator == null)
      return "Locator_is_null";

    DataSource ds;
    DataSource clone;

    try {
      ds = javax.media.Manager.createDataSource(locator);
    } catch (Exception e) {
      return "Couldn't_create_DataSource";
    }

    // Try to create a processor to handle the input media locator
    try {
      processor = javax.media.Manager.createProcessor(ds);
    } catch (NoProcessorException npe) {
      return "Couldn't_create_processor";
    } catch (IOException ioe) {
      return "IOException_creating_processor";
    }

    // Wait for it to configure
    boolean result = waitForState(processor, Processor.Configured);
```

```
if (result == false)
  return "Couldn't␣configure␣processor";

// Get the tracks from the processor
TrackControl [] tracks = processor.getTrackControls();

// Do we have atleast one track?
if (tracks == null || tracks.length < 1)
  return "Couldn't␣find␣tracks␣in␣processor";

// Set the output content descriptor to RAW_RTP
// This will limit the supported formats reported from
// Track.getSupportedFormats to only valid RTP formats.
ContentDescriptor cd = new ContentDescriptor(ContentDescriptor.
   RAW_RTP);
processor.setContentDescriptor(cd);

Format supported[];
Format chosen;
boolean atLeastOneTrack = false;

// Program the tracks.
for (int i = 0; i < tracks.length; i++) {
  Format format = tracks[i].getFormat();
  if (tracks[i].isEnabled()) {
    supported = tracks[i].getSupportedFormats();

  // We've set the output content to the RAW_RTP.
  // So all the supported formats should work with RTP.
  // We'll just pick the first one.

    if (supported.length > 0) {
      if (supported[0] instanceof VideoFormat) {
    // For video formats, we should double check the
    // sizes since not all formats work in all sizes.
        chosen = checkForVideoSizes(tracks[i].getFormat(),
          supported[0]);
      } else
        chosen = supported[0];
      tracks[i].setFormat(chosen);
      System.err.println("Track␣" + i + "␣is␣set␣to␣transmit␣as:")
        ;
      System.err.println("␣␣" + chosen);
      atLeastOneTrack = true;
    } else
    tracks[i].setEnabled(false);
 } else
```

```
        tracks[i].setEnabled(false);
    }

    if (!atLeastOneTrack)
      return "Couldn't set any of the tracks to a valid RTP format";

    // Realize the processor. This will internally create a flow
    // graph and attempt to create an output datasource for JPEG/RTP
    // audio frames.
    result = waitForState(processor, Controller.Realized);
    if (result == false)
      return "Couldn't realize processor";

    // Set the JPEG quality to .5.
    setJPEGQuality(processor, (float).5);

    // Get the output data source of the processor
    dataOutput = processor.getDataOutput();

    return null;
  }


  /**
   * Use the SessionManager API to create sessions for each media
   * track of the processor.
   */
  private String createTransmitter() {
  // Cheated.  Should have checked the type.
    PushBufferDataSource pbds = (PushBufferDataSource)dataOutput;
    PushBufferStream pbss[] = pbds.getStreams();

    rtpMgrs = new SessionManager[pbss.length];
    rtpsocks = new RTPSocket[pbss.length];
    rtp = new ANTSHandler[pbss.length];
    rtcp = new ANTSHandler[pbss.length];

    SessionAddress localAddr, destAddr;
    InetAddress ipAddr;
    SendStream sendStream;
    short port;
    SourceDescription srcDesList[];

    System.out.println("Creating RTP session " + pbss.length);
    for (short i = 0; i < pbss.length; i++) {
      try {
        rtpsocks[i] = new RTPSocket();
    // rtpsocks[i].setContentType("rtpraw");

    port = (short)(portBase + 2 * i);
    rtp[i] = new ANTSHandler(this, group, target, port, i);
    rtcp[i] = new ANTSHandler(this, group, target,
      (short)(port + 1), i);
    rtpsocks[i].setInputStream(rtp[i]);
    rtpsocks[i].setOutputStream(rtp[i]);
    RTPPushDataSource rtcpsource;
    rtcpsource = rtpsocks[i].getControlChannel();
    rtcpsource.setOutputStream(rtcp[i]);
    rtcpsource.setInputStream(rtcp[i]);

    rtpsocks[i].connect();

    rtpMgrs[i] = new RTPSessionMgr(rtpsocks[i]);
    System.out.println("RTP Session Initialized");
    srcDesList = new SourceDescription[] {
      new SourceDescription(
        SourceDescription.SOURCE_DESC_EMAIL,
        "zhoux@cs.umanitoba.ca",
        1,
        false),
      new SourceDescription(SourceDescription.SOURCE_DESC_CNAME,
        rtpMgrs[i].generateCNAME(),
        1,
        false),
      new SourceDescription(SourceDescription.SOURCE_DESC_TOOL,
        "JMF_RTP_Player_v2.0",
        1,
        false),
    };

    localAddr = new SessionAddress();

    ipAddr = InetAddress.getByName(ipAddress);
    // destAddr here doesn't matter, 'cuz we've used the
       ANTSHandler instead.
    destAddr = new SessionAddress(ipAddr, port,
        ipAddr, port + 1);


    rtpMgrs[i].initSession(localAddr, srcDesList, 0.05, 0.25);

    rtpMgrs[i].startSession(destAddr, 1, null);
    sendStream = rtpMgrs[i].createSendStream(dataOutput, i);
    sendStream.start();
```

```
        } catch (Exception e) {
          return e.getMessage();
        }
      }
      return null;
    }


    /**
     * For JPEG and H263, we know that they only work for particular
     * sizes.  So we'll perform extra checking here to make sure they
     * are of the right sizes.
     */
    Format checkForVideoSizes(Format original, Format supported) {
      int width, height;
      Dimension size = ((VideoFormat)original).getSize();
      Format jpegFmt = new Format(VideoFormat.JPEG_RTP);
      Format h263Fmt = new Format(VideoFormat.H263_RTP);

      if (supported.matches(jpegFmt)) {
        // For JPEG, make sure width and height are divisible by 8.
        width = (size.width % 8 == 0 ? size.width :
          (int)(size.width / 8) * 8);
        height = (size.height % 8 == 0 ? size.height :
          (int)(size.height / 8) * 8);
      } else if (supported.matches(h263Fmt)) {
      // For H.263, we only support some specific sizes.
        if (size.width < 128) {
          width = 128;
          height = 96;
        } else if (size.width < 176) {
          width = 176;
          height = 144;
        } else {
          width = 352;
          height = 288;
        }
      } else {
      // We don't know this particular format.  We'll just
      // leave it alone then.
        return supported;
      }

      return (new VideoFormat(null,
          new Dimension(width, height),
          Format.NOT_SPECIFIED,
          null,
        Format.NOT_SPECIFIED)).intersects(supported);
  }


  /**
   * Setting the encoding quality to the specified value on the JPEG
   *   encoder.
   * 0.5 is a good default.
   */
  void setJPEGQuality(Player p, float val) {
    Control cs[] = p.getControls();
    QualityControl qc = null;
    VideoFormat jpegFmt = new VideoFormat(VideoFormat.H261);

    // Loop through the controls to find the Quality control for
    // the JPEG encoder.
    for (int i = 0; i < cs.length; i++) {
      if (cs[i] instanceof QualityControl &&
        cs[i] instanceof Owned) {
        Object owner = ((Owned)cs[i]).getOwner();

        // Check to see if the owner is a Codec.
        // Then check for the output format.
        if (owner instanceof Codec) {
          Format fmts[] = ((Codec)owner).getSupportedOutputFormats(
              null);
          for (int j = 0; j < fmts.length; j++) {
            if (fmts[j].matches(jpegFmt)) {
              qc = (QualityControl)cs[i];
              qc.setQuality(val);
              System.err.println("-␣***********Setting␣quality␣to␣" +
                val + "␣on␣" + qc);
              break;
            }
          }
        }
        if (qc != null)
          break;
      }
    }
  }


  /****************************************************************
   * Convenience methods to handle processor's state changes.
   ****************************************************************/
```

```
private Integer stateLock = new Integer(0);
private boolean failed = false;

Integer getStateLock() {
  return stateLock;
}

void setFailed() {
  failed = true;
}

private synchronized boolean waitForState(Processor p, int state) {
  p.addControllerListener(new StateListener());
  failed = false;

  // Call the required method on the processor
  if (state == Processor.Configured) {
    p.configure();
  } else if (state == Processor.Realized) {
    p.realize();
  }

  // Wait until we get an event that confirms the
  // success of the method, or a failure event.
  // See StateListener inner class
  while (p.getState() < state && !failed) {
    synchronized (getStateLock()) {
      try {
        getStateLock().wait();
      } catch (InterruptedException ie) {
        return false;
      }
    }
  }

  if (failed)
    return false;
  else
    return true;
}

/**************************************************************
 * Inner Classes
 **************************************************************/

class StateListener implements ControllerListener {
  public void controllerUpdate(ControllerEvent ce) {
      // If there was an error during configure or
      // realize, the processor will be closed
      if (ce instanceof ControllerClosedEvent)
        setFailed();

      // All controller events, send a notification
      // to the waiting thread in waitForState method.
      if (ce instanceof ControllerEvent) {
        synchronized (getStateLock()) {
          getStateLock().notifyAll();
        }
      }
    }
  }


  /*****************************************************************
   * ANTS Codes
   *****************************************************************/
  public void setArgs(KeyArgs k) throws Exception {
    k.merge(defaults);

    for (int i = 0; i < k.length(); i++) {
      if(k.key(i).equals("-source")) {
        MediaSource = k.arg(i);
        k.strike(i);
      } else
      if (k.key(i).equals("-target")){
        target = NodeAddress.fromString(k.arg(i));
        k.strike(i);
      } else
      if(k.key(i).equals("-group")) {
          group = NodeAddress.fromString(k.arg(i));
          k.strike(i);
      } else
      if(k.key(i).equals("-port")) {
        portBase = Short.parseShort(k.arg(i));
        k.strike(i);
      }

    }
    super.setArgs(k);
  }

  synchronized public void receive(Capsule cap) {
    super.receive(cap);
```

```
        if(!(cap instanceof ARTMcastCapsule))
            return;

        ARTMcastCapsule mcap = (ARTMcastCapsule)cap;
        int DstPort = mcap.getDstPort();
        int hIndex = (DstPort - portBase) / 2;

        ANTSHandler targetHandler;
        if(((DstPort - portBase) % 2) == 0) {
          rtp[hIndex].receiveCapsule(mcap);
        } else {
          rtcp[hIndex].receiveCapsule(mcap);
        }
    }

    public void start() throws Exception {
        getNode().register(new ARTMcastProtocol());

        locator = new MediaLocator("file:/darkcity.mov");
        String result = go();
        if (result != null) {
          System.err.println("Error : " + result);
          System.exit(0);
        }

        System.err.println("Start transmission for 180 seconds...");

        // Transmit for 60 seconds and then close the processor
        // This is a safeguard when using a capture data source
        // so that the capture device will be properly released
        // before quitting.
        // The right thing to do would be to have a GUI with a
        // "Stop" button that would call stop on AVTransmit
        try {
          Thread.currentThread().sleep(180000);
        } catch (InterruptedException ie) {
        }

        // Stop the transmission
        stop();

        System.err.println("...transmission ended.");

        System.exit(0);
    }

}
```

B.10 ANTSReceive.java

```
package apps;

import java.io.*;
import java.awt.*;
import java.net.*;
import java.awt.event.*;
import java.util.Vector;

import javax.media.*;
import javax.media.rtp.*;
import javax.media.rtp.event.*;
import javax.media.rtp.rtcp.*;
import javax.media.protocol.*;
import javax.media.format.AudioFormat;
import javax.media.format.VideoFormat;
import javax.media.Format;
import javax.media.format.FormatChangeEvent;
import javax.media.control.BufferControl;
import com.sun.media.rtp.*;

import ants.*;
import utils.*;

class MCASubscribe implements Runnable {
  ANTSReceive mapp;

  public void run() {
    while (true) {
      ARTMcastSubscribeCapsule subs = new
        ARTMcastSubscribeCapsule(mapp.target, mapp.group);
      synchronized(mapp.SyncObj) {
        subs.BuildTrackUpdate(mapp.trackIDs, mapp.procs);
        mapp.send(subs);
      }
      try {
        Thread.sleep(mapp.sdelay+10);
      } catch (InterruptedException e) {}
    }
  }

  MCASubscribe(ANTSReceive mapp) {
    this.mapp = mapp;
  }
}
```

```
/**
 * ANTSReceive is the receiver application.
 */
public class ANTSReceive extends Application
  implements ReceiveStreamListener, SessionListener,
    ActionListener, ControllerListener
{
  short trackIDs[] = null;
  short procs[] = null;

  SessionManager mgrs[] = null;
  RTPSocket rtpsocks[] = null;
  ANTSHandler rtp[] = null;
  ANTSHandler rtcp[] = null;

  Vector playerWindows = null;

  int group;
  int target;
  short portBase;
  int nStreams;
  int sdelay = 1000;

  boolean dataReceived = false;
  Object dataSync = new Object();

  public Object SyncObj = new Object();  // Object used to synchronize
       with Subscribing Thread

  Button shutoff;

  public ANTSReceive() {
  }

  protected boolean initialize() {
    try {
      InetAddress ipAddr;
      short port;
      SessionAddress localAddr = new SessionAddress();
      SessionAddress destAddr;

      mgrs = new com.sun.media.rtp.RTPSessionMgr[nStreams];
      rtpsocks = new RTPSocket[nStreams];
      rtp = new ANTSHandler[nStreams];
      rtcp = new ANTSHandler[nStreams];
      playerWindows = new Vector();
      SessionLabel session;

      // Open the RTP sessions.
      for (int i = 0; i < nStreams; i++) {
        // Parse the session addresses.

        rtpsocks[i] = new RTPSocket();
        port = (short)(portBase + 2 * i);
        System.err.println("  - Open RTP session for: addr: " + group
            + "/" + target + " port: " + port);

        rtp[i] = new ANTSHandler(this, group, target, port, (short)-1)
            ;
        rtcp[i] = new ANTSHandler(this, group, target,
          (short)(port + 1), (short)-1);
        rtpsocks[i].setInputStream(rtp[i]);
        rtpsocks[i].setOutputStream(rtp[i]);
        RTPPushDataSource rtcpsource;
        rtcpsource = rtpsocks[i].getControlChannel();
        rtcpsource.setOutputStream(rtcp[i]);
        rtcpsource.setInputStream(rtcp[i]);

        rtpsocks[i].connect();
        mgrs[i] = new RTPSessionMgr(rtpsocks[i]);
        mgrs[i].addSessionListener(this);
        mgrs[i].addReceiveStreamListener(this);

        ipAddr = InetAddress.getByName("127.0.0.1");
        destAddr = new SessionAddress(ipAddr, port,
            ipAddr, port + 1);
        mgrs[i].initSession(localAddr, getSDES(mgrs[i]), .05, .25);

        // You can try out some other buffer size to see
        // if you can get better smoothness.
        BufferControl bc = (BufferControl)mgrs[i].getControl("javax.
           media.control.BufferControl");
        if (bc != null)
          bc.setBufferLength(350);

        mgrs[i].startSession(destAddr, 1, null);
      }

    } catch (Exception e){
      System.err.println(e.toString());
      e.printStackTrace();
      System.err.println("Cannot create the RTP Session: " + e.
          getMessage());
        return false;
      }

      // Wait for data to arrive before moving on.

      long then = System.currentTimeMillis();
      long waitingPeriod = 60000;  // wait for a maximum of 30 secs.

      try{
        synchronized (dataSync) {
          while (!dataReceived &&
                System.currentTimeMillis() - then < waitingPeriod) {
              if (!dataReceived)
                System.err.println("  - Waiting for RTP data to arrive..."
                    );
              dataSync.wait(1000);
          }
        }
      } catch (Exception e) {}

      if (!dataReceived) {
        System.err.println("No RTP data was received.");
        close();
        return false;
      }

      return true;
    }


    /**
     * Find out the host info.
     */
    String cname = null;
    private SourceDescription[] getSDES(SessionManager mgr) {
      SourceDescription[] desclist = new SourceDescription[3];
      if (cname == null)
        cname = mgr.generateCNAME();

      desclist[0] = new SourceDescription(SourceDescription.
          SOURCE_DESC_NAME,
                System.getProperty("user.name"),
                1,
                false);
      desclist[1] = new SourceDescription(SourceDescription.
          SOURCE_DESC_CNAME,
                cname,
                1,
                false);
        desclist[2] = new SourceDescription(SourceDescription.
            SOURCE_DESC_TOOL,
                "AVReceive powered by JMF",
                1,
                false);
        return desclist;
    }


    public boolean isDone() {
        return playerWindows.size() == 0;
    }


    /**
     * Close the players and the session managers.
     */
    protected void close() {
        for (int i = 0; i < playerWindows.size(); i++) {
            try {
                ((PlayerWindow)playerWindows.elementAt(i)).close();
            } catch (Exception e) {}
        }

        playerWindows.removeAllElements();

        // close the RTP session.
        for (int i = 0; i < mgrs.length; i++) {
            if (mgrs[i] != null) {
                mgrs[i].closeSession("Closing session from AVReceive");
                mgrs[i] = null;
            }
        }
    }


    PlayerWindow find(Player p) {
        for (int i = 0; i < playerWindows.size(); i++) {
            PlayerWindow pw = (PlayerWindow)playerWindows.elementAt(i);
            if (pw.player == p)
                return pw;
        }
        return null;
    }
```

```
PlayerWindow find(ReceiveStream strm) {
  for (int i = 0; i < playerWindows.size(); i++) {
    PlayerWindow pw = (PlayerWindow)playerWindows.elementAt(i);
    if (pw.stream == strm)
      return pw;
  }
  return null;
}


/**
 * SessionListener.
 */
public synchronized void update(SessionEvent evt) {
  if (evt instanceof NewParticipantEvent) {
    Participant p = ((NewParticipantEvent)evt).getParticipant();
    System.err.println("␣␣-␣A␣new␣participant␣had␣just␣joined:␣" + p
        .getCNAME());
  }
}


/**
 * ReceiveStreamListener
 */
public synchronized void update( ReceiveStreamEvent evt) {
  SessionManager mgr = (SessionManager)evt.getSource();
  Participant participant = evt.getParticipant(); // could be null.
  ReceiveStream stream = evt.getReceiveStream();  // could be null.

  if (evt instanceof RemotePayloadChangeEvent) {
    System.err.println("␣␣-␣Received␣an␣RTP␣PayloadChangeEvent.");
    System.err.println("Sorry,␣cannot␣handle␣payload␣change.");
    System.exit(0);
  } else if (evt instanceof NewReceiveStreamEvent) {
    try {
      stream = ((NewReceiveStreamEvent)evt).getReceiveStream();
      DataSource ds = stream.getDataSource();

      // Find out the formats.
      RTPControl ctl = (RTPControl)ds.getControl("javax.media.rtp.
          RTPControl");
      if (ctl != null){
        System.err.println("␣␣-␣Recevied␣new␣RTP␣stream:␣" + ctl.
            getFormat());
      } else
```

```
          System.err.println("␣␣-␣Recevied␣new␣RTP␣stream");

        if (participant == null)
          System.err.println("␣␣␣␣The␣sender␣of␣this␣stream␣had␣yet␣to
              ␣be␣identified.");
        else {
          System.err.println("␣␣␣␣The␣stream␣comes␣from:␣" +
              participant.getCNAME());
        }

        // create a player by passing datasource to the Media Manager
        Player p = javax.media.Manager.createPlayer(ds);
        if (p == null)
          return;

        p.addControllerListener(this);
        p.realize();
        PlayerWindow pw = new PlayerWindow(this, p, stream);
        playerWindows.addElement(pw);

        // Notify intialize() that a new stream had arrived.
        synchronized (dataSync) {
          dataReceived = true;
          dataSync.notifyAll();
        }
      } catch (Exception e) {
        System.err.println("NewReceiveStreamEvent␣exception␣" + e.
            getMessage());
        return;
      }

    } else if (evt instanceof StreamMappedEvent) {
      if (stream != null && stream.getDataSource() != null) {
        DataSource ds = stream.getDataSource();
        // Find out the formats.
        RTPControl ctl = (RTPControl)ds.getControl("javax.media.rtp.
            RTPControl");
        System.err.println("␣␣-␣The␣previously␣unidentified␣stream␣");
        if (ctl != null)
          System.err.println("␣␣␣␣" + ctl.getFormat());
        System.err.println("␣␣␣␣had␣now␣been␣identified␣as␣sent␣by:␣"
            + participant.getCNAME());
      }
    } else if (evt instanceof ByeEvent) {
      System.err.println("␣␣-␣Got␣\"bye\"␣from:␣" + participant.
          getCNAME());
      PlayerWindow pw = find(stream);
      if (pw != null) {
        pw.close();
        playerWindows.removeElement(pw);
      }
    }
  }


  /**
   * ControllerListener for the Players.
   */
  public synchronized void controllerUpdate(ControllerEvent ce) {
    Player p = (Player)ce.getSourceController();

    if (p == null)
      return;

    // Get this when the internal players are realized.
    if (ce instanceof RealizeCompleteEvent) {
      PlayerWindow pw = find(p);
      if (pw == null) {
      // Some strange happened.
        System.err.println("Internal error!");
        System.exit(-1);
      }
      pw.initialize();
      pw.setVisible(true);
      p.start();
    }

    if (ce instanceof ControllerErrorEvent) {
      p.removeControllerListener(this);
      PlayerWindow pw = find(p);
      if (pw != null) {
        pw.close();
        playerWindows.removeElement(pw);
      }
      System.err.println("AVReceive internal error: " + ce);
    }
  }


  /**
   * A utility class to parse the session addresses.
   */
  class SessionLabel {
    public String addr = null;
```

```
public int group;
public int target;
public int port;
public int ttl = 1;

SessionLabel(String session) throws IllegalArgumentException {
  int off;
  String portStr = null, ttlStr = null;

  if (session != null && session.length() > 0) {
    while (session.length() > 1 && session.charAt(0) == '/')
      session = session.substring(1);

  // Now see if there's a addr specified.
    off = session.indexOf('/');
    if (off == -1) {
      if (!session.equals(""))
      addr = session;
    } else {
      addr = session.substring(0, off);
      session = session.substring(off + 1);
      // Now see if there's a port specified
      off = session.indexOf('/');
      if (off == -1) {
        if (!session.equals(""))
          portStr = session;
      } else {
        portStr = session.substring(0, off);
        session = session.substring(off + 1);
        // Now see if there's a ttl specified
        off = session.indexOf('/');
        if (off == -1) {
          if (!session.equals(""))
            ttlStr = session;
        } else {
          ttlStr = session.substring(0, off);
        }
      }
    }
  }

  if (addr == null)
    throw new IllegalArgumentException();

  if (portStr != null) {
    try {
      Integer integer = Integer.valueOf(portStr);
```

```
          if (integer != null)
            port = integer.intValue();
        } catch (Throwable t) {
          throw new IllegalArgumentException();
        }
      } else
        throw new IllegalArgumentException();

      if (ttlStr != null) {
        try {
          Integer integer = Integer.valueOf(ttlStr);
          if (integer != null)
            ttl = integer.intValue();
        } catch (Throwable t) {
          throw new IllegalArgumentException();
        }
      }
    }
  }


  public void actionPerformed(ActionEvent event) {
    System.out.println("Action_Performed");
    ARTCtrlCapsule ctl = new ARTCtrlCapsule(target, group);
    synchronized(SyncObj) {
      ctl.BuildTrackUpdate(trackIDs, procs);
      send(ctl);
      for(int i = 0; i < nStreams; i++)
        procs[i] = (procs[i] == 3)? (short)0 : (short)3;
    }
  }

  /**
   * GUI classes for the Player.
   */
  class PlayerWindow extends Frame {
    ANTSReceive application;
    Player player;
    ReceiveStream stream;

    PlayerWindow(ANTSReceive app, Player p, ReceiveStream strm) {
      application = app;
      player = p;
      stream = strm;
    }

    public void initialize() {
      Panel topPanel = new Panel();
      topPanel.setLayout(new GridLayout(1, 3, 1, 1));
      shutoff = new Button("Shut␣Off");
      shutoff.addActionListener(application);

      topPanel.add(shutoff);
      add("North", topPanel);
      add(new PlayerPanel(player));
    }

    public void close() {
      player.close();
      setVisible(false);
      dispose();
    }

    public void addNotify() {
      super.addNotify();
      pack();
    }
  }


  /**
   * GUI classes for the Player.
   */
  class PlayerPanel extends Panel {
    Component vc, cc;

    PlayerPanel(Player p) {
      setLayout(new BorderLayout());
      if ((vc = p.getVisualComponent()) != null)
        add("Center", vc);
      if ((cc = p.getControlPanelComponent()) != null)
        add("South", cc);
    }

    public Dimension getPreferredSize() {
      int w = 0, h = 0;
      if (vc != null) {
        Dimension size = vc.getPreferredSize();
        w = size.width;
        h = size.height;
      }
      if (cc != null) {
        Dimension size = cc.getPreferredSize();
        if (w == 0)
```

```
          w = size.width;
        h += size.height;
      }
      if (w < 160)
        w = 160;
      return new Dimension(w, h);
    }
  }

  public void setArgs(KeyArgs k) throws Exception {
    k.merge(defaults);

    for (int i = 0; i < k.length(); i++) {
      if (k.key(i).equals("-target")){
        target = NodeAddress.fromString(k.arg(i));
        k.strike(i);
      } else
      if(k.key(i).equals("-group")) {
        group = NodeAddress.fromString(k.arg(i));
        k.strike(i);
      } else
      if(k.key(i).equals("-port")) {
        portBase = Short.parseShort(k.arg(i));
        k.strike(i);
      } else
      if(k.key(i).equals("-n")) {
        nStreams = Integer.parseInt(k.arg(i));
        k.strike(i);
      }

    }
    super.setArgs(k);
  }

  synchronized public void receive(Capsule cap) {
    super.receive(cap);
    if(!(cap instanceof ARTMcastCapsule))
      return;

    ARTMcastCapsule mcap = (ARTMcastCapsule)cap;
    int DstPort = mcap.getDstPort();

    ANTSHandler targetHandler;
    int indx = (DstPort - portBase) / 2;
    if(((DstPort - portBase) % 2) == 0) {
      rtp[indx].receiveCapsule(mcap);
    } else {
      rtcp[indx].receiveCapsule(mcap);
    }
  }

  public void start() throws Exception {
    getNode().register(new ARTMcastProtocol());
    for(int i = 0; i < nStreams; i++) {
      getNode().attachApplication(this, portBase + 2 * i);
      getNode().attachApplication(this, portBase + 2 * i + 1);
    }

      synchronized(SyncObj) {
        trackIDs = new short[nStreams];
        procs = new short[nStreams];
        for(int i = 0; i < nStreams; i++) {
          trackIDs[i] = (short)i;  //Initially, the receiver's
              interested in all tracks.
          procs[i] = 0;
        }
      }

    new Thread(new MCASubscribe(this)).start();

    if (!initialize()) {
      System.err.println("Failed_to_initialize_the_sessions.");
      System.exit(-1);
    }

    // Check to see if AVReceive is done.
    try {
      while (!isDone())
      Thread.sleep(1000);
    } catch (Exception e) {}

    System.err.println("Exiting_AVReceive");
  }

}// end of AVReceive
```

Glossary

ANEP	Active Network Encapsulation Protocol
ANTS	Active Node Toolkit System
CBT	Core Based Tree
DVMRP	Distance Vector Multicast Routing Protocol
IGMP	Internet Group Management Protocol
JMF	Java Media Framework
LAN	Local Area Network
MOSPF	Multicast Open Shortest Path First protocol
PIM-DM	Protocol Independent Multicasting-Dense Mode
PIM-SM	Protocol Independent Multicasting-Sparse Mode
QoS	Quality of Service
RP	Rendezvous Point
RTCP	Real-Time Cotnrol Protocol
RTP	Real-time Transport Protocol
SPT	Shortest Path Tree
TBT	Truncated Broadcast Tree
TCP	Transfer Control Protocol
UDP	User Datagram Protocol
WAN	Wide Area Network

Bibliography

[ABG97a] D. S. Alexander, B. Braden, and C. A. Gunter. Internet Group Management Protocol, Version 2, November 1997. RFC 2236.

[ABG+97b] D. S. Alexander, B. Braden, C. A. Gunter, A. W. Jackson, A. D. Keromytis, G. J. Minden, and D. Wetherall. Active Network Encapsulation Protocol (ANEP). Active Networks Group, July 1997. RFC Draft.

[AGH90] D. P. Anderson, R. Govindan, and G. Homsy. Design and Implementation of a Continuous Media I/O Server. In *First International Workshop on Network and Operating System Support for Digital Audio and Video*, November 1990.

[AHK+98] D. Alexander, M. Hicks, P. Kakkar, A. Keromytis, M. Shaw, J. Moore, C. Gunter, S. Nettles, and J. Smith. The SwitchWare Active Network Implementation. In *Proceedings of the ACM SIGPLAN '98 Workshop on ML*, May 1998.

[AMK98] E. Amir, S. McCanne, and R. H. Katz. An Active Service Framework and Its Application to Real-Time Multimedia Transcoding. In *Proceedings of ACM SIGCOMM '98*, pages 178–189, September 1998.

[AMZ95] E. Amir, S. McCanne, and H. Zhang. An Application Level Video Gateway. In *Proceedings of ACM Multimedia '95*, November 1995.

[BDS96] I. Busse, B. Deffner, and H. Schulzrinne. Dynamic QoS Control of Multimedia Applications Based on RTP. In *Computer Communications*, January 1996.

[Bel58] R. Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[BFC93] T. Ballardi, P. Francis, and J. Crowcroft. Core Based Trees (CBT) - An Architecture for Scalable Inter-Domain Multicast Routing. In *Proceedings of ACM SIGCOMM '93*, pages 85–95, October 1993.

[CBZS98] K. Calvert, S. Bhattacharjee, E. Zegura, and J. Sterbenz. Directions in Active Networks. *IEEE Communications Magazine*, pages 72–78, October 1998.

[CCI90] CCITT. Recommendation H.261, Video Codec for Audiovisual Services at $p \times 64 kBit/s$, January 1990.

[CT90] D. D. Clark and D. L. Tennenhouse. Architectural Considerations for a New Generation of Protocols. In *Proceedings of ACM SIGCOMM '90*, pages 200–208, September 1990.

[DC90] S. E. Deering and D. R. Cherition. Multicast Routing in Datagram Internetworks and Extended LANs. *ACM Transactions on Computer Systems*, 8(2):85–110, May 1990.

[DEF$^+$95] S. Deering, D. Estrin, D. Farinacci, V. Jacobson, C. Liu, and L. Wei. Protocol Independent Multicast (PIM): Motivation and architecture. Internet Draft draft-ietf-idmr-pim-arch-01.ps, January 1995.

[DMRS00] S. Dawson, M. Molteni, L. Ricciulli, and S. Sui. User Guide to Anetd 1.6.3. SRI International, September 2000.

[Eri94] H. Eriksson. MBONE: The Multicast Backbone. *Communications of the ACM*, 37(8):54–60, August 1994.

[HMA$^+$99] M. Hicks, J. Moore, D. Alexander, C. Gunter, and S. Nettles. PLANet: An Active Internetwork. In *Proceedings of the 18th IEEE Computer and Communication Society (INFOCOM '99)*, March 1999.

[HSJ98] R. Frederick H. Schulzrinne, S. Casner and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. *Internet-Draft ietf-avt-rtp-new-01.txt (work in progress)*, 1998.

[JMF99] JMF 2.0 FCS. *Java Media Framework API Guide.* Sun Microsystems, Inc., November 1999.

[KAC98] Bobby Krupczak, Mostafa H. Ammar, and Kenneth L. Calvert. Implementing Protocols in Java: The Price of Portability. In *Proceedings of INFOCOM '98*, pages 765–773, March 1998.

[KLB99] M. J. Katchbaw, H. L. Lutfiyya, and M. A. Bauer. Using User Hints to Guide Resource Management for Quality of Service. In *Proceedings Intl. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA '99)*, pages 1069–1075, June 1999.

[KPS02] A. Kind, R. Pletka, and B. Stiller. The Potential of Just-in-Time Compilation in Active Networks based on Network Processors. In *Proceedings of IEEE OPENARCH'02*, June 2002.

[MHN01] J. Moore, M. Hicks, and S. Nettles. Practical Programmable Packets. In *Proceedings of the Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'01)*, pages 41–50, April 2001.

[MJ95a] R. McCanne and V. Jacobson. vat: The LBNL Audio Conferencing Tool. *Online available at http://www-nrg.ee.lbl.gov/vat*, 1995.

[MJ95b] S. McCanne and V. Jacobson. vic : A Flexible Framework for Packet Video. In *ACM Multimedia*, pages 511–522, October 1995.

[MJV96] S. McCanne, V. Jacobson, and M. Vetterli. Receiver-Driven Layered Multicast. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, volume 26,4, pages 117–130, August 1996.

[Moy93] J. Moy. MOSPF: Analysis and Experience, July 1993. Internet Draft.

[RT98] R. S. Ramanujan and K. J. Thurber. An Active Network Based Design of a QoS Adaptive Video Multicast Service. In *Proceedings of the 1998 World Conference on Systems, Cybernetics and Informatics*, pages 643–650, July 1998.

[Sch95] H. Schulzrinne. Internet Services: from Electronic Mail to Real-Time Multimedia. In *Proceedings of Kommunikation in Verteilten Systemen*, pages 21–34, February 1995.

[Ste98] W. R. Stevens. *UNIX Network Programming Volume I 2nd ed. Networking APIs: Sockets and XTI.* Prentice Hall, Inc., 1998.

[TW96] D. L. Tennenhouse and D. J. Wetherall. Towards an Active Network Architecture. *ACM Computer Communication Review*, 26(2), April 1996.

[Wet99] D. J. Wetherall. Active Network Vision and Reality: Lessons From a Capsule-Based System, December 1999.

[WGT98] D. Wetherall, J. Guttag, and D. Tennenhouse. ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols. In *Proceedings of IEEE OPENARCH*, April 1998.

[WPD88] D. Waitzman, C. Partridge, and S. Deering. Distance Vector Multicast Routing Protocol, November 1988. RFC 1075.

[Yd96] Y. Yemini and S. da Silva. Towards Programmable Networks. In *IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, October 1996.

[ZDE93] L. Zhang, S. Deering, and D. Estrin. RSVP: A New Resource ReSerVation Protocol. *IEEE Network*, 7(5):8–18, September 1993.