An Object-Oriented Architecture for the Real-Time Transmission of Multimedia Data Streams

Thomas Peuker
Institut für Mathematische Maschinen
und Datenverarbeitung (Informatik) IV
Lehrstuhl für Betriebssyteme
Universität Erlangen-Nürnberg

Erlangen, March 17, 1997

Abstract

Continuous media, namely video and sound, are beginning to conquer computer networks. The possibility of transmitting such media over networks facilitates new kinds of distributed applications, such as digital television broadcasting, teleconferencing, or multimedia information systems.

All these applications, collectively called multimedia applications, require the transmission of multimedia data streams. The inherent real-time constraints of multimedia data place new demands on network systems and the employed transport protocols.

This thesis investigates into the general and time-related characteristics of multimedia data streams and multimedia applications. Furthermore, it reviews several techniques, which have been developed recently by the Internet community for the transmission of time-critical data. In particular, the Internet Integrated Services Architecture, the Resource ReSerVation Protocol, and the Real-Time Transport Protocol receive special attention.

Distributed multimedia applications require a design that emphasises flexibility, modularity, expandability, and efficiency. The object-oriented paradigm offers proven methods for the design and implementation of complex applications. The thesis summarises several object-oriented approaches to multimedia application designs, in general, and specifically demonstrates how network communication is integrated into object-oriented multimedia applications.

Based on these studies, the thesis proposes an object-oriented architecture for the transmission of multimedia data streams. The architecture is intended to serve as a communication framework for distributed multimedia applications.

This architecture has also been implemented as part of this thesis, using the Java programming language and incorporating the Real-Time Transport Protocol. The implementation successfully verifies the conceptual suitability of the architectural design.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Multimedia is the holy grail of networking.
When the word is mentioned,
both the propeller heads and the suits begin salivating as on cue.
The former see immense technical challenges
in providing (interactive) video on demand to every home.
The latter see equally immense profits in it.
— ANDREW S. TANENBAUM, COMPUTER NETWORKS, 1996

During the last years, the ever increasing power of microprocessors and the integration of sound and video equipment enabled standard personal computers and workstations to present or even generate digital media in real-time: multimedia computers were born.
Until recently, multimedia computing was limited to playing back media clips stored on bulk storage devices such as CD ROMs. However, the number of computers connected to local networks or the world-wide Internet increased by orders of magnitude, and this process will probably continue in the future. The usage of computer networks for the communication of multimedia data, promises lots of new possibilities of people-to-people and people-to-machine communication. New kinds of applications are arising, including remote representation, distributed multimedia information systems, or video-on-demand services.

From a technical point of view, multimedia networking consists mainly of the transmission of multimedia data streams. Consequently, network systems become subject to the real-time requirements inherent in multimedia data. Hence, standardized mechanisms must be deployed all over a network to ensure the timely delivery of time-based data. Additionally, new transport protocols have to be developed, which support the transmission of time-critical data such as multimedia data streams. Moreover, appropriate protocol solutions are required by distributed multimedia applications for the exchange of control and management information

Particularly exacting demands must be made on the design of the distributed multimedia applications themselves. Such a design must offer general but powerful abstractions, must be universally applicable, easily expandable, flexible, modular, and must permit efficient implementation. The object-oriented paradigm, incorporating object-oriented analysis, design, and programming methods, facilitates and supports such development goals.

The goals of this thesis are, therefore,

- to study the characteristics of multimedia data streams and the requirements for their processing and transmission,
- to evaluate the suitability of recent networking techniques and protocols for the transmission of real-time multimedia data,
- to review object-oriented design approaches for distributed multimedia applications, and finally,—based on these studies–

- to design and implement an object-oriented architecture for the transmission of multimedia data streams.

The architecture is intended to serve as a communication framework for distributed multimedia applications. It must take into account the intrinsic real-time requirements of multimedia data. The object-oriented paradigm is to be applied to the design and implementation of this architecture.

The remainder of this thesis is organized in the following chapters:
In Chapter 2, I will investigate into the general requirements of distributed multimedia applications and examine multimedia data streams particularly with respect to their time constraints.
During the course of Chapter 3, I will present various techniques and protocols developed by the Internet Engineering Task Force in connection with the transmission of multimedia data. In particular, I will discuss the structure and suggested usage of the Internet Integrated Services Architecture and the Resource ReSerVation Protocol (RSVP) in Section 3.1, and review the Real-Time Transport Protocol (RTP) in Section 3.2. Protocols that complement RSVP and RTP for certain kinds of applications will be subject of two case studies in Section 3.3.
Multimedia data transmission and object-orientation are interconnected in Chapter 4. The chapter begins with an overview of the object-oriented paradigm, and continues with a summary of various object-oriented approaches to multimedia application design. The data-stream and the object-stream model are presented in more detail, paying special attention to the integration of network communication.
Based on the considerations of the Chapters 2–4, I propose in Chapter 5 an object-oriented design of an architecture for the transmission of multimedia data streams. The design aims to integrate the vertical, inter-application data flow into the horizontal, intra-application data flow in a efficient and homogeneous way.
The `mmstream` proof-of-concept implementation of this architecture is described in Chapter 6. The implementation is written in the Java programming language and incorporates the Real-Time Transport Protocol.
Finally, Chapter 7 summarises the ideas contained in the Chapters 2–6. In addition, it provides a performance analysis of the `mmstream` implementation as well as a discussion of topics of further work.

Chapter 2

Requirements of Multimedia Applications

2.1 Characteristics of Multimedia Systems

2.1.1 Multimedia Data Streams

The information that can be perceived by the human senses is transported through different *media*, such as text or sound. Humans communicate with computers by means of various media, or use computers as tools for communication with each other. These observations led Steinmetz to the following definition of *multimedia systems* [HS91]:

> "A multimedia system is characterised by the computer-controlled generation, manipulation, presentation, storage, and communication of independent discrete and continuous media."

The ability to handle *continuous media*, such as audio or video, distinguishes multimedia systems from most conventional computer systems. Continuous media, while being actually discrete, appear smooth to the human observer when presented regularly and periodically at sufficiently high frequencies. Therefore, continuous media are *time dependent*, and their processing is subject to *time constraints*.

A good way to describe the propagation of continuous media data through a system or over a network is the *stream* metaphor: a sequence of data units flows from sources to sinks along the data path. A set of related data substreams, each carrying one particular continuous medium, forms a *multimedia data stream*. Following Husemann [Hus96, p. 20], a multimedia data stream $\mathcal{M}$ is defined formally as a sequence of data quanta c_k^u contributed by the single-medium substreams $\mathcal{S}_u$ to the multimedia stream M:

$$\begin{aligned}
\mathcal{S}_u &= \langle \dots, c_d^u, c_e^u, c_f^u, \dots \rangle &= \langle c_k^u \rangle, \qquad \forall u \in \{r, s, t, \dots\}, \qquad k \in \mathbb{N}_0 \\
\mathcal{M} &= \langle \dots, c_h^r, c_i^s, c_j^t, \dots \rangle &= \langle c_k^u \rangle, \qquad u \in \{r, s, t, \dots\}, \qquad k \in \mathbb{N}_0
\end{aligned}$$

Most research into the modelling of multimedia data is done in the context of multimedia database systems.[1] Gibbs et al. [GBT93] also consider database issues, but focus on the time aspects of multimedia data. Their investigation into the time constraints of multimedia data streams caused the authors to develop the notion of *timed streams*. A timed stream[2] consists of a finite sequence of tuples $\langle e_i, s_i, d_i \rangle$, $i = 1, \dots, n$. All e_i are *media elements*[3] of a certain *media type*. The media type is characteristic for the stream and specifies the attributes of the medium

[1] See for example [KNS90].

[2] See [GBT93, p. 8] for the complete formal definitions of timed streams.

[3] Gibbs' media elements e_i correspond to Husemann's data quanta c_i^u belonging to one substream $\mathcal{S}_u$.

of the stream. s_i and d_i are measured in a *discrete time system* and specify the time constraints of the presentation of e_i: s_i is the start time of the presentation of e_i, and d_i is the duration of the presentation.
Consider for example a fixed–frame-rate MPEG video stream: the individual video frames could be identified as the media elements e_i, whose size can vary, depending on the MPEG type of each frame e_i; because of the constant frame rate, the d_i would be equal for all i; furthermore, the end of the duration of the presentation of element e_i would coincide with the beginning of the presentation of the next element e_{i+1}, formally

$$s_{i+1} = s_i + d_i \quad \text{and} \quad d_i = const. \quad \text{for} \quad i = 1, \ldots, n-1$$

2.1.2 Multimedia Application Categories

Multimedia applications can be subdivided into different categories, each making particular demands for support on the operating system or runtime environment. Husemann [Hus96, p. 14–16] suggests three application categories:

- *Information Systems.* The main purpose of such systems is to provide information for one or several users. The requested information is typically stored in databases or media archives. Examples are electronic publishing, online galleries or weather information systems.

- *Remote Representation.* By means of a remote representation system a user can take part in or monitor events at a remote location.
  Important examples are distance conferencing or lecturing, virtual reality, or remote robotic agents.

- *Entertainment.* This major application area of multimedia technology is strongly oriented towards audio and video data.
  Example entertainment applications are digital television, video on demand, distributed games or interactive television.

User interaction possibilities are a more technical classification criterion favoured by Heinrichs [Hei96, p. 5–6]:

- *Interactive Services.* Interactive services permit the user to select the transmitted information. These services can be be further subdivided into:
  - *Conversational Services.* Services with real-time demands and no relevant buffering, like video conferencing or video surveillance.
  - *Messaging Services.* Services with temporary storing, like multimedia mail.
  - *Retrieval Services.* Information services interactively presenting previously stored information from a database or media collection, for example tele shopping or hospital information systems.

- *Distribution Services.* Distribution services transmit information from a central source to a potentially unknown set of receivers. There are two subcategories that differ in the control possibilities granted the users:
  - *Services without User Control.* Services characterised by having one central sender that broadcasts information to all participating users, for example digital television broadcasting.
  - *Services with User Control.* Services allowing the user to choose from the distributed information.

2.1.3 Resource Demands of Multimedia Applications

Applications make use of various system *resources*, which are provided by the different system components. According to Steinmetz [Ste95], a resource is a system entity that a task requires for the manipulation or presentation of data. Typical resources offered by a multimedia-capable system include for example the CPU, the memory management, the file systems, network access, and special hardware like cameras or sound subsystems.
A resource is categorised as either:

- *Active* or *passive*. Active resources provide a service, normally by making use of passive resources.
- *Exclusive* or *shared*. Exclusive resources can be used by one task at a time only, whereas shared resources may be used by various tasks concurrently.
- *Single* or *multiple*, depending on the number of instances of that particular resource existing in a given system.

The *capacity* of a resource is measured by a task's ability to perform its work in a given time span using that resource. To support continuous media, even systems comprising high-capacity networks and fast processors require *real-time mechanisms* to provide the necessary resources within the time constraints specified by the desired quality and by the type of the processed media.

For distributed multimedia applications, the required communication structure is of great importance. Traditional networking applications like file transfer generate one-to-one traffic, whereas typical multimedia applications such as video conferences or distributed virtual reality games communicate via one-to-many or even many-to-many message paths.

Table 2.1 on page 6, taken from [Hus96, p. 17], compares the demands of some typical application types.

2.1.4 Quality Of Service

The term *quality of service (QoS)* originated in the field of communications and was used to describe the technical characteristics of data transmission. The notion can be generalised in order to describe any service that a system offers, as done for example in [VKBG95]:

> "Quality of service represents the set of those quantitative and qualitative characteristics of a distributed multimedia system necessary to achieve the required functionality of an application."

All system entities—especially the operating system components and the network subsystem—must cooperate to provide services that meet the demands of the applications.

The following criteria, adapted from [Ste95], are used to classify the quality of a service offered by an active resource:

Throughput. Throughput is defined as the product of the size or number of the data units and of the rate at which the units are processed.

Delay. The *local* delay is the time a resource needs to complete a task; the *global* delay is the total delay a data unit needs for travelling from its source to its sink, thereby making use of several resources.

Jitter. Jitter is the maximum allowed variance in inter-arrival the time interval of the data.

Reliability Reliability issues comprise error frequency and possible error types as well as error-detection and error-correction mechanisms.

Category	Application	Bandwidth	Real-time requirements	Communication structure	Media
Information systems	hospital information systems	medium–very high	high	1-to-1, 1-to-n	high-resolution images, text, video, sound
	electronic publishing	low	none	1-to-1, 1-to-n	text, images
	museums	low–medium	none	1-to-1	(high-resolution) images, text
Remote representation	distance learning, conferencing	high	strong	1-to-n, n-to-m	text, video, sound
	virtual reality	medium–very high	high–very high	1-to-1, n-to-m	text, video, sound
	robotic agents	low–high	high	1-to-1	text, video, sound
	remote task agents	low	none	1-to-1, 1-to-n	text, video, sound, images
Entertainment	digital TV	very high	very high	1-to-n	high-definition video, text, stereo audio
	interactive TV	low	low	m-to-1	text
	video-on-demand	medium–very high	low–very high	1-to-1, 1-to-n	text, video, sound
	widely distributed interactive games	low	low–medium	1-to-n, n-to-m	text

Table 2.1: Requirements of different multimedia application classes.

Using the notation of Husemann [Hus96, p. 16–21], a QoS requirement q necessary for processing a single medium is defined as the n-tuple of values r_i, which parametrise the required usage of resources R_i:

$$q = (r_1, \ldots, r_n) \quad \in \quad \mathbf{R} = R_1 \times \ldots \times R_n$$

The set of QoS requirements q_i of all k substreams $\mathcal{S}_i$ constituting a multimedia data stream $\mathcal{M}$ defines the *quality of service profile* Q of the data stream:

$$Q(\mathcal{M}) \quad = \quad (q_1, \ldots, q_k)$$

Similarly, the QoS profile Q of an application a comprises all QoS requirements q_i that are needed to handle the involved m media:

$$Q(a) \quad = \quad (q_1, \ldots, q_m)$$

To assure timely presentation or manipulation of multimedia data, an application requires either *guaranteed services*, for which strictly observed *quality-of-service guarantees* are given by the service-providing system entity, or fairly reliable *predictive services*, which behave as specified for the most part. In both cases, an application must consult the *resource manager* that is responsible for a particular system service to make a *resource reservation*.
An application reserves resources by performing the following steps:

1. *Estimating* the user's subjective QoS wishes.
2. *Mapping* these estimations to a technical QoS profile.
3. *Negotiating* a contract about the QoS values with the resource management.

A resource manager handles a QoS request in four phases:

1. *Schedulability Test.* The resource manager checks the (worst-case or statistical) resource utilisation and decides whether the additional request can still be met.
2. *Quality-of-Service Calculation.* The resource manager estimates the best possible performance of the requested service that can be guaranteed or sustained for most of the time.
3. *Resource Reservation.* The resource managers and all other entities involved store the new allocation with the existing ones.
4. *Resource Scheduling.* During run time, incoming events and commands are scheduled according to the contracted guarantees.

The actual QoS values in a system can vary over time. A system must therefore permanently monitor its state and take measures to meet the guarantees, or must notify the clients about insufficiencies. The system components should also be prepared to renegotiate the contract if a user decides to alter the requirements.

2.2 Process Management

The preceding considerations showed that, from an operating system point of view, *real-time scheduling* and *data transmission* are the most crucial topics.
Since this thesis deals primarily with the transmission of multimedia data, processor scheduling will be covered only within this section. The ideas presented here are mainly based on Steinmetz' analysis of multimedia operating systems [Ste95].

2.2.1 Requirements

The resource management of a real-time system schedules the access of *tasks* to system resources. A task is defined as a schedulable system entity such as a thread or a process, and is characterised by its time constraints and resource requirements. The timeliness of the media presentations or manipulations performed by a task is as crucial as the correctness of the operation.

The aim of a processor resource's management component is to calculate a schedule that allows the timely processing of as many time-critical tasks as possible. A real-time multimedia scheduler must take into account *externally* defined time constraints as well as time limits imposed by *internal* data dependencies.

The real-time requirements of multimedia applications are usually—depending on the application type—less demanding than those of traditional command and control systems, which often have a direct physical impact:

- Multimedia systems are usually more *fault tolerant* than command and control systems.
- While *missing a deadline* is not desirable, it generally does not have severe physical consequences.
- Continuous media data is most often the result of the *periodic* sampling of a signal. Hence, tasks require the processor resource periodically; such tasks are much easier to schedule than sporadic tasks.
- The *bandwidth demand* can often be adjusted dynamically to the available capacities. This is known as *media scaling*.

2.2.2 Scheduling Strategies

The following paragraphs cover only *periodic* tasks, which request the processors at constant intervals. Two algorithms dominate the scheduler implementations in multimedia-capable operating systems:

Earliest Deadline First (EDF). EDF is an *optimal dynamic* algorithm. Any beginning of a new period of a task triggers a reordering of the complete schedule, according to the deadlines of the scheduled tasks.

Rate-Monotonic Scheduling Rate-monotonic scheduling is an *optimal static*, priority-driven algorithm. During the set-up phase, each task is assigned a static priority according to its requested processing rate: the shorter a task's period, the higher is its priority. Afterwards, each task is processed with this priority, and no rearrangement of the priorities is necessary as long as no new task appears.

Both algorithms can be extended by dividing each task into a mandatory part, which secures an acceptable result, and an optional part, which refines the work. Tasks are primarily scheduled with respect to the mandatory parts, while the optional parts are only processed during underload periods. This arrangement allows scaling of continuous media.

Comparing EDF to rate-monotonic scheduling, context switches are less probable using EDF. In addition, EDF can schedule tasks successfully at processor-utilisation rates up to 100 percent, whereas rate monotonic's processor utilisation is upwardly bounded by 69 percent before a failure occurs in worst-case scenarios.[4]

On the other hand, rate monotonic incurs less overhead for continuous-media data streams: the rate-monotonic algorithm uses the periodicity of continuous media to compute a static schedule during setup while EDF must determine the task to run for every incoming event.

[4] See [Ste95] for details.

2.3 Multimedia-Data Transmission

Real-time multimedia applications, such as video on demand, use the communication subsystem in different ways than traditional applications such as file transfer or electronic mail do. Obviously, the prevailing service model of the Internet—the best-effort, point-to-point delivery service of IP—does not reflect the needs of continuous media transmission. The special needs of real-time–multimedia network traffic that also led to the proposed Integrated Services extensions for the Internet architecture (see Section 3.1), will be considered in the remainder of Section 2.3.

2.3.1 Multi-Point Communication

Multi-point communication structures, which are characterised by a group of m senders transmitting data to a group of n recipients, are often necessary for distributed multimedia applications. Typical scenarios include computer supported cooperative work (CSCW), multimedia conferencing, distributed virtual reality, distributed simulation, digital television, or distance teaching.

To avoid inefficient repeated sending of n identical messages, the communication system must offer *multicast transmission*. Multicasting requires support concerning the following issues:

Multicast Group and Address Management. To dynamically participate in multicast sessions, any participant should be able to initiate, join or leave *multicast groups* at any time. Setting up a multicast group requires the assignment and propagation of a *multicast group address*, used as the destination address for all data sent to this group.

Transport Reliability and Flow Control. New mechanisms must be developed that offer reliability without creating huge amounts of control traffic. To avoid costly retransmissions of packets dropped due to network congestion, flow control techniques must be used to adapt the sent traffic quantity to the network congestion state along all multicast data paths.

Routing. Multicasting asks for routing protocols that reduce duplicate traffic on any shared link to a minimum, preferably to one copy. In addition, out of both network-load and security concerns, data should only be sent along links that have at least one recipient.
Naturally, a routing algorithm should choose the optimal path for the transmitted data, but with multicasting, it must consider the paths to a (potentially widely spread) set of receivers.

2.3.2 Quality-of-Service Control

The general QoS criteria listed in Section 2.1.4 are very suitable to describe the QoS aspects of multimedia-data transmission. However, this generic set must be extended for transmission-specific features. The network type-independent QoS parameter list includes:

- *Connection establishment delay.*
- *Connection establishment failure probability.*
- *Sustained or peak throughput rate.*
- *Transit delay.*
- *Delay variation.*
- *Loss rate.*
- *Error rate.*
- *Security and Protection parameters.*
- *Priority.*

Some of these parameters are negotiable between network systems and applications, whereas others are fixed or statistical characteristics of a network. Each networking technique might extend the above list by specific parameters.

2.3.3 Flow Control

The main task of flow control is to protect the receiver from an unmanageable flood of packets and from consequent data loss due to reception buffer overflow. For that purpose, senders and receivers can reach traffic agreements that are independent from the underlying communication system.

There are two main approaches to flow control:

Window-Based Flow Control. Window-based systems work with respect to the maximum amount of data that the sender is allowed to sent without having received an acknowledgement for data already transmitted. The sizes of both sending and reception windows must be adapted to the participants' capacities and to the characteristics of a particular network.

Rate-Based Flow Control. In a rate-based system, a sender forwards data at a reasonable rate that is chosen in order to allow network and receivers to cope with the transmitted volume. Rate-controlled systems do not necessarily enforce an acknowledgement of the received data. However, some feedback about the current transmission quality is useful for adaptive systems that allow media scaling.

Chapter 3

Transmission of Multimedia Data

In this chapter, I discuss the transmission aspects of multimedia data streams in more detail. Section 3.1 presents the Internet Integrated Services extensions. Section 3.2 reviews higher-level transport protocols that can be used on top of the common Internet protocols and that offer additional, desirable features for multimedia applications.
Applying these considerations, I will conclude this chapter with two case studies in Section 3.3.

3.1 Internet Integrated Services Architecture

3.1.1 Introduction

The services offered by established Internet protocols are unsuitable for real-time applications because *variable queueing delays* and *packet loss because of congestion* can occur at any time.
Integrating real-time QoS mechanisms, which can control end-to-end packet delay, requires admission control and network management facilities that give network operators the ability to control the sharing of bandwidth and other QoS resources among different traffic classes. This form of network operation is known as *controlled link sharing*.

Research activities, such as the "Integrated Services Packet Network" described in [CSZ92], were undertaken to improve the situation. The result is laid down in RFC 1633 [RFC1633]: the *Integrated Services (IS) in the Internet Architecture*. Braden, Clark, and Shenker, the authors of RFC 1633, construct an Internet architecture that offers controlled link sharing, real-time services, and best-effort services for traditional applications with no real-time requirements.

Although this new architecture substantially changes the service model of the Internet in theory, the migration to the new structure should not prove very difficult because the changes are incorporated in the prevailing IP architecture as an extension of the original architecture.

The authors of [RFC1633] emphasise the separation of the new *service model*, which defines the externally visible behaviour of the network (see Section 3.1.3), from the proposed *reference implementation framework*.
The reference implementation framework consists of a *traffic control* module (see Section 3.1.4), which enforces different qualities of service, and a *reservation setup protocol* (see Section 3.1.5), which creates and maintains flow-specific state on each host and router along the paths of a flow.

3.1.2 Flow and Service Description

[RFC1633] defines a *flow* as a distinguishable stream of related datagrams that result from a single[1] user activity and require the same QoS. A flow is considered simplex, which means that it has one source and—in a multicasting-capable network—n destinations.

[1] In newer drafts such as [SW96a], the aggregated traffic, which comprises data from a number of application sessions and which is covered by one QoS control request, is also considered as a single flow.

Flows define the finest granularity of packet streams distinguishable by the IS architecture. A network element uses *filter specs* to identify the flow that a packet belongs to. These filter specifications contain the necessary information for performing the identification; for example, the source and destination host address and port number together with the protocol ID, or flow IDs as proposed for IPv6.

The *allowed traffic pattern* of a flow is described by means of a *traffic specification* or *TSpec*. Note that the TSpec describes the worst-case behaviour of the sender, not the average traffic generation. As long as a source conforms to its TSpec, it can expect that contracted QoS guarantees are met by the network. A TSpec might, for example, take the form of a token-bucket specification, or an upper bound on the peak sending rate.

The QoS requests of a flow are expressed by a *service request specification* or *RSpec*. Such specifications might contain bandwidth requirements, acceptable maximum delays or packet loss rates.

The combination of the information contained in an RSpec and a TSpec is sometimes called a *flow specification* or *flowspec*.[2] A flowspec and a filter spec together constitute a *flow descriptor*.

A *service* or *QoS control service* describes a named, coordinated set of QoS control capabilities that are provided by a single network element. A service definition must comprise a specification of the functions to be performed by the element, the information necessary to perform these functions, the information that the element exports to other entities in the system, as well as relevant information.

The IS working group has defined a framework [SW96a] for specifying services provided by network elements within an internetwork. This draft lays down the service specification document format as well as naming conventions for services and parameters.
Each service parameter appears in two forms: *local* and *composed*. Local values describe the conditions at a particular network element, whereas composed values reflect the composition of local values along a path through the network. Hence, the definition of a parameter must include a corresponding composition rule which defines how to combine an incoming composed value with the local value; the result of this composition may be forwarded to the next element along the path.
For example, to characterise the maximum transfer unit that all elements along a data path up to the local system can handle, the composition rule would be the minimum function applied to the incoming composed value and the local value.

A set of general control and characterisation parameter definitions common and meaningful to *all* services is collected in [SW96b].

3.1.3 Integrated Services Model

The core service model of [RFC1633, Section 3] considers only quantitative service commitments concerning the bounds on minimum and maximum per-packet transmission delays within a flow.[3]

Applications can be divided into two categories according to the way their performance depends on the transmission delay behaviour.

Real-Time Applications. For this class of applications the transmitted data is of any value only if it arrives within a certain time.
Often, real-time applications belong to the class of *playback applications*, which consist of a source that transforms a signal into data packets and transmits these over the network. At the receiver's site, the packets arrive potentially disordered and with variable delays. The receiver then tries to reconstruct the source data from the packets and attempts to play back the signal as faithfully as possible with some fixed offset delay from the departure time.
An application must find a suitable a priori estimate of this offset delay. It can either be

[2] See [RFC1363] for an alternative flow specification.

[3] Nevertheless, the IS architecture is not restricted only to delay-bounded services. See [SW96a] for a service specification template, and [SW96b] for general service parameters.

provided by a network service commitment containing a delay bound or by observation of the previously received traffic.

The performance of a playback application is determined by the presentation *latency* and *fidelity*, both of which are affected by the delay behaviour of the network.
First, the latency of an application depends on the offset delay predictions about future packet delays. Secondly, the fidelity of the playback can be distorted by individual packet delays exceeding the predictions.

The sensitivity to loss of fidelity leads to two application classes, each requiring a particular service class:

- *Intolerant Applications.* Intolerant systems require absolutely faithful playback fidelity and can be found for example with circuit emulation.

  Intolerant applications must choose a fixed offset delay, which is larger than the maximum expected packet delay. Consequently, a reliable upper bound on the maximum packet delay is required.

  A service that *enforces* such a *reliable* bound is called a *guaranteed service*[4], and embodies the appropriate service model for intolerant playback applications.

- *Tolerant Applications.* Tolerant systems work satisfactorily even with a certain loss of fidelity. Most video and sound applications are tolerant or even rate-adaptive in their encoding scheme.

  The service model proposed in [RFC1633] for tolerant applications is called *predictive service*. Its delay bound is *fairly but not perfectly reliable*; the delay bound could be a prediction based not on worst-case but on statistical assumptions about the behaviour of other data flows in the system. The idea behind predictive service is that a potential, small, tolerable loss of application performance will facilitate a considerable gain in overall network efficiency.

  Newer IS drafts do not mention predictive service. Instead, a similar service, called *controlled-load service*[5], was developed. Controlled-load service tries to approximate the behaviour of best-effort service under unloaded network conditions at any time. Client applications may assume that a very high percentage of the transmitted packets will be successfully delivered by the network, and that the transit delay experienced by a very high percentage of the delivered packets will not greatly exceed the minimum transit delay experienced by any successfully delivered packet.
  Admission control is necessary to ensure that the clients obtain this service quality even if the network is overloaded.

Naturally, in most cases it will be difficult to rigidly distinguish between tolerant and intolerant applications, as their characteristics are continually changing. Delay-bounded services require a characterisation of the expected traffic and admission control mechanisms in order to be able to guarantee or predict the service quality (see Section 3.1.4 for details).

Elastic Applications. Elastic applications will always wait for data to arrive rather than proceed without it. Although the performance of these applications will be marred by a large (average) delay, they require no a priori characterisation of the delay in order to function properly.
Subcategories of elastic applications with different delay expectations are interactive burst (Telnet, X, NFS), interactive bulk transfer (FTP), and asynchronous bulk transfer (electronic mail, FAX).
A network should offer a *best-effort service* for use with elastic applications, also known as *as-soon-as-possible* or *ASAP* service. Best-effort service is not subject to admission control.

Other topics addressed by the Integrated Services model include:

[4] See [SPG96] for a specification of guaranteed service conforming to [SW96a].
[5] See [Wro96b] for a specification of controlled-load service conforming to [SW96a].

- *Resource Sharing.* Whereas QoS commitments lay down how the network allocates its resources, *link sharing* addresses how the total bandwidth of a link is shared among various client entities.
  Possible forms of link sharing are:
  - Multi-entity link sharing
  - Multi-protocol link sharing
  - Multi-service link sharing
- *Packet Dropping.* The Integrated Service model proposes the introduction of a *preemptible packet service*, which treats packets marked preemptible as less important than those not marked. If the system is in danger of violating its commitments in cases of network overload, it can drop packets that are marked preemptible and thereby reduce the delay of the non-preempted packets.
  As an example, when transmitting MPEG video frames, B-frames, which belong to the least important frame type, could be marked as preemptible while it would not be advisable to discard any I-frames, which are of highest importance.
- *Usage Feedback.* Controlled–link-sharing service, which can be used to impose limits on resource consumption, is one way to prevent abuse of network resources. Another possibility is to make users responsible for their usage, which is known as network *accounting*.
- *Reservation Model.* The reservation model describes how an application negotiates a certain QoS level with the network system.
  A flowspec is passed to the appropriate resource manager, which can refuse the request, accept it, or grant a lower QoS level if the network cannot promise to fulfill the original request to its full extent.
  All affected system components, especially the routers along the path of a flow, must be informed about the actually granted level to set up corresponding local reservations. For this purpose, the use of a reservation setup protocol such as RSVP[6] is recommended.

3.1.4 Traffic Control Module

The traffic control functions within a host or router are provided by the packet scheduler, the packet classifier, and the admission control. Figure 3.1 on page 15 demonstrates how these elements fit into the implementation reference model.

Packet Classifier. To support traffic control and accounting, each incoming packet is first identified by information gained from filter specs as belonging to a particular flow, and then mapped into a corresponding service class. All packets within one class are treated equally. The choice of the appropriate class is made locally at each router and reflects the importance that a router gives to an individual flow.
The packet classification can be performed by looking at the source and destination host address, the protocol number, and the port fields. To reduce the overhead, the addition of a *flow-id* field in the IP-layer packet header is proposed, which allows a short-cut classification of the packets.

Packet Scheduler. The packet scheduler mainly manages the forwarding of different packet streams by (re-)ordering the output queue according to the importance of the packets' classes. Therefore, the packet scheduler must be part of the output driver of an operating system, namely of that part, which manages the output packet queues. The packet scheduling algorithm follows for example priority-based, round-robin, or weighted-fair-queueing (WFQ) approaches.
Packet scheduling also includes the decision which packets to drop in case of overload.

[6] See Section 3.1.5 and Appendix B for a description of RSVP.

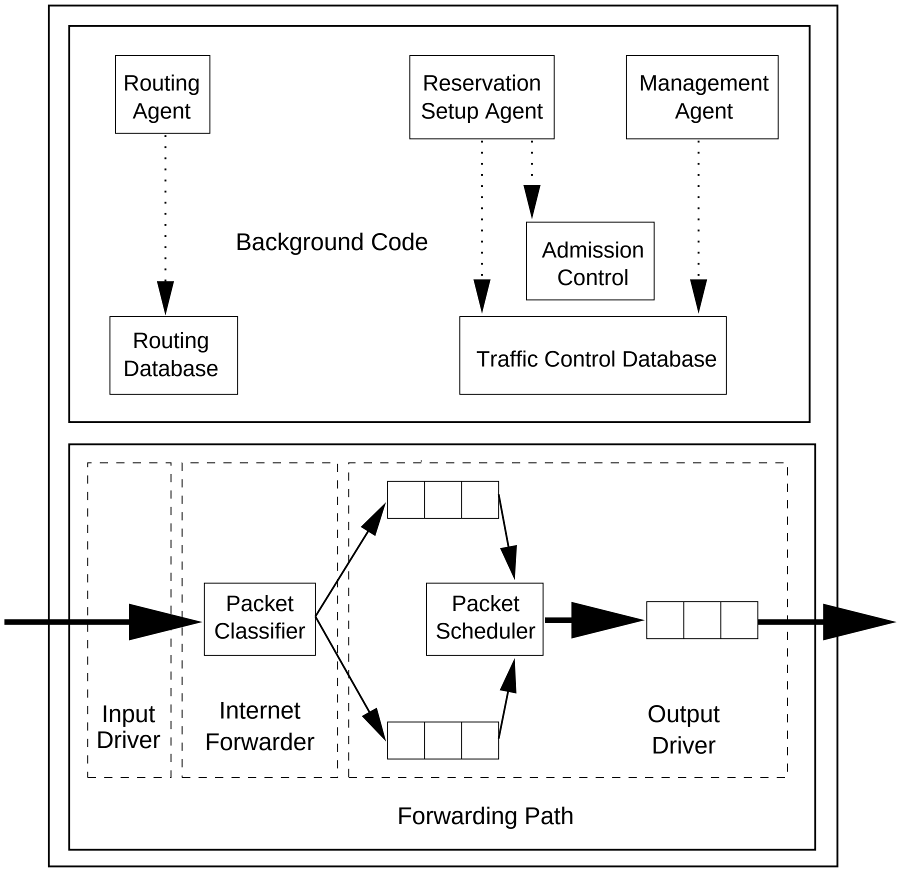

Figure 3.1: Integrated services implementation reference model for routers.

Furthermore, the packet scheduler ensures that a sending host behaves according to its promised traffic pattern. This function is called *policing* or *enforcement*.
The packet scheduler can also act as a *traffic estimator*: it measures properties of the outgoing packet streams. This information is used to generate traffic statistics which support packet scheduling and admission control decisions.

Admission Control. The admission control decides whether QoS requirements of new flows can be met without violating earlier guarantees. The admission control algorithm must be invoked at each router along the path of the new flow.
The decision of admittance can be based either on worst-case assumptions about existing and new flows, or on measures of the actual traffic. The latter approach incurs a higher risk of overload, but can utilise the available bandwidth considerably better.
Admission control can also be employed to enforce administrative policies for QoS arbitration.

3.1.5 Resource ReSerVation Protocol—RSVP

RSVP Overview

RSVP fulfills the role of a resource reservation setup protocol in an integrated services internetwork.
RSVP is designed with the following goals in mind [ZDESZ93]:

- Accommodate heterogeneous receivers.

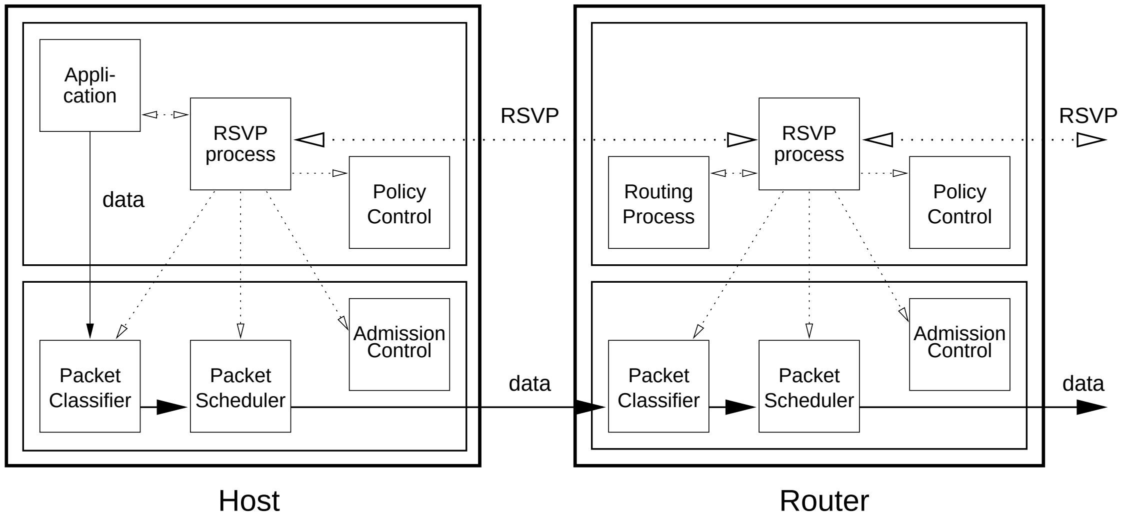

Figure 3.2: RSVP integration in host and routers.

- Adapt to changing multicast group memberships.
- Exploit the characteristics of the resource requirements of different applications in order to utilise network resources efficiently.
- Allow receivers to switch channels.
- Adapt dynamically to changes in the underlying unicast and multicast routes.
- Control the protocol overhead so that the complexity grows less than linearly with the number of participants.
- Make the design modular to accommodate heterogeneous underlying network technologies.

The introduction to the Internet draft specifying RSVP [BZBHJ96] suggests how and by whom RSVP is employed:

> "The RSVP protocol is used by a host, on behalf of an application data stream, to request a specific quality of service (QoS) from the network for particular data streams or flows. The RSVP protocol is also used by routers to deliver QoS control requests to all nodes along the path(s) of the flows and to establish and maintain state to provide the requested service. RSVP requests will generally, although not necessarily, result in resources being reserved in each node along the data path."

Figure 3.2 on page 16 visualises how RSVP is integrated into routers and hosts.

RSVP handles *receiver-initiated resource requests* for *simplex* flows.

In order to scale well for very large, dynamically changing multicast groups, RSVP incrementally builds and destroys *soft state* in each network node along the paths of a flow. That is, RSVP periodically sends refresh messages to maintain reservation state along the paths. The absence of a certain number of refresh messages automatically times out and deletes reservation state. The soft-state approach has several consequences:

- The deletion of timed-out state automatically releases resources that do not seem to be used any more.
- The QoS of a flow will automatically recover from route changes if enough capacities are available along the new paths. The state along the abandoned path will be deleted after the expiration of the time-out interval, and new reservations will be set up along the new route by the periodic refresh messages.

- Because of the periodic transmission of refresh messages, RSVP is expected to deal gracefully with occasional packet loss.[7] Hence, no costly reliability enhancements are necessary to transmit RSVP messages.

Although RSVP is located on top of IP in the protocol stack, it is not a transport protocol but operates rather similarly to control protocols such as ICMP. RSVP is also not a routing protocol but is intended to cooperate with unicast and multicast routing protocols. The cooperation between the routing module and the RSVP module of a node allows for *local repair*, which means that the routing module in a network node notifies the RSVP module of route changes to RSVP-managed flows, so that the RSVP module can quickly adjust the resource reservations to the new routes.

RSVP Messages and Objects

RSVP is designed to be usable with many QoS control services, which in turn are intended to work with more than one reservation setup mechanism. The role of RSVP consists merely of distributing QoS-control data across the network and of passing these data to the traffic control module on each host along the path.

Because of the logical separation between QoS control services and the distribution of QoS control information, the RSVP specifications define only the format[8] of *RSVP QoS control objects*, but the RSVP code treats the contents of these objects as opaque data.

RSVP objects are enclosed in RSVP messages, which are sent hop-by-hop between RSVP-capable routers as raw IP datagrams with protocol number 46. End systems can communicate with the first/last hop routers with UDP-encapsulated RSVP messages if they cannot do raw network I/O.

The direction from the source of a flow (thereafter called the sender) to the sinks of the flow (thereafter called receivers) is called *downstream*, correspondingly, the opposite direction is correspondingly called *upstream*.
Figure 3.3 demonstrates the various directions and message paths.

[Wro96a] describes how to use a certain subset of the RSVP objects with the Controlled-Load and Guaranteed QoS control services. In particular, the named draft defines the usage and contents of three RSVP object classes:[9]

- `FLOWSPEC`
- `ADSPEC`
- `SENDER_TSPEC`

These objects are encapsulated in the two fundamental RSVP message types[10], `PATH` and `RESV`, which deserve some consideration:

- `PATH` *Messages.*
  `PATH` messages travel downstream from the RSVP sender to the set of receivers, following the same unicast or multicast route as the data packets.
  The `PATH` messages store *path state* in each node along the path. The path state currently consists of the unicast IP address of the previous hop node. This information is used to route `RESV` messages in the reverse (upstream) direction.

  In addition to the address of the previous hop, the `PATH` messages contain the following objects:

[7] To tolerate n consecutive packet losses, the time-out interval must be set to $(n+1)$ times the refresh period.
[8] See Appendix B.1 for RSVP's object construction rules.
[9] See Appendix B.2 for the format definition of these classes.
[10] See Appendix B.3 for a list of all RSVP message types object classes.

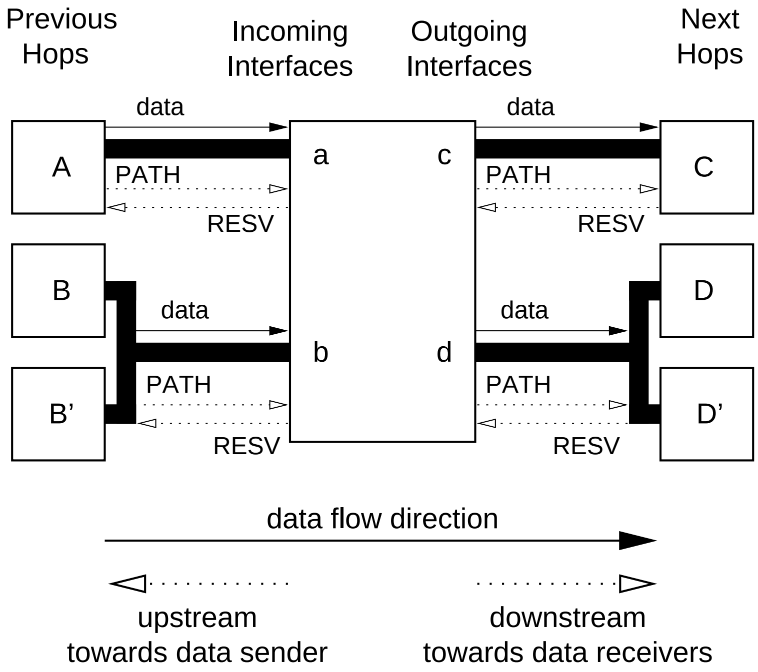

Figure 3.3: RSVP message paths and directions.

- `SENDER_TEMPLATE` objects. A `PATH` message must carry a `SENDER_TEMPLATE` object, which contains the information necessary to identify data packets of the sender of the `PATH` message. `SENDER_TEMPLATE` objects are structured in the format of filter specs, which are also used in `RESV` messages.
  `SENDER_TEMPLATE` objects are never changed on their way along the path.
- `SENDER_TSPEC` objects. `SENDER_TSPEC` objects are also compulsory, immutable constituents of every `PATH` message.
  The TSpec describes the bounds of the traffic that the sender expects to generate. Receivers can use these estimations when requesting resource reservations.
- `ADSPEC` objects. The optional `ADSPEC` objects gather both information about the properties of the data path—such as available services, delay and bandwidth estimates—as well as parameters required by specific QoS control services.
  `ADSPEC` objects are generated by data senders or by intermediate network elements, and are modified as they travel from one RSVP-capable node to another. An `ADSPEC` object *advertises* the possible service parameters composed of the properties of all previous-hop nodes upstream. The arriving `ADSPEC` object is combined with the node's own parameters and service conditions, and then forwarded to the next node.
  A receiver can use the `ADSPEC` information to predict the end-to-end QoS, to choose the most appropriate service and to scale its QoS request according to the current possibilities of the network.
  A receiver can in particular tell from an `ADSPEC` whether there are any non-RSVP-capable hosts in the path; if this is the case, the values of all parameters should be considered as unreliable, because then there are nodes in the path where the flow will experience only best-effort service.

• `RESV` *Messages.*
`RESV` messages are sent upstream by receivers who want to establish a reservation for one or more senders. The `RESV` messages follow exactly the reverse route of the data packets, using the path state created by the `PATH` messages.

Sender Selection	Reservations	
	Distinct	Shared
Explicit	Fixed-Filter (FF) style	Shared-Explicit (SE) style
Wildcard	(None defined)	Wildcard-Filter (WF) style

Table 3.1: RSVP reservation attributes and styles.

The fundamental objects contained in `RESV` messages belong to the `FLOWSPEC`, `FILTER_SPEC`, and `STYLE` classes. A receiver, wishing to make a reservation, can use the information contained in previously received `ADSPEC` and `SENDER_TSPEC` objects to choose reasonable service parameters values for the TSpec and RSpec in his `RESV` message; in the same way, the data from `SENDER_TEMPLATE` objects is used to configure potential `FILTER_SPEC` objects.

An `RESV` message coming in from a downstream node is passed to the admission and policy control modules. If they both grant the service request, *reservation state* is set up in the node by passing the `FILTER_SPEC` object to the packet classifier, and by configuring the packet scheduler with the `FLOWSPEC` data as described in Section 3.1.4.
Note that this process does not necessarily produce a new reservation. Depending on the reservation style selected with a `STYLE` object, the new request might instead be merged or shared with other requests, thereby modifying existing reservations. The effective `FLOWSPEC` of some mergeable requests is computed by finding the *least upper bound (LUB)* or *greatest lower bound (GLB)* that identifies the most demanding request.
A `RESV` message with a potentially modified `FLOWSPEC` is forwarded upstream if the reservation request is not already covered by previous requests from other receivers of the same flow.

The possible RSVP reservation attributes and styles are listed in Table 3.1. Senders, for whose traffic reservations are established, can be selected *implicitly* by a *wildcard*, which selects all senders in a session[11], or *explicitly* by a list of all selected senders. When using the explicit sender-selection style, filter specs for each selected sender are needed.
In addition, either one reservation can be *shared* among the flows within a session, or a *distinct* reservation is made for each data sender.

The basic model is a one-pass scheme: receivers send reservation requests upstream where they are either accepted or rejected by the nodes along the reverse data path. However, senders and network elements advertise their traffic characteristics and service capabilities downstream, such that receivers are provided with information about the behaviour of the upstream path elements. The receivers can use this information to parametrise their reservation requests according to the actual flow characteristics and network capacity.
This combined scheme is called *One Pass With Advertising (OPWA)*.[12]

RSVP Interfaces

RSVP communicates with the various components inside a network node: on a router there must be interfaces for the routing and traffic control modules, whereas on hosts there must exist an interface for applications (an RSVP API) and also—if existing—for traffic control.

From the point of view of an application programmer, the client application programming interface is of major importance. Its main purpose is to pass the data contained in `SENDER_TSPEC`, `ADSPEC`, and `FLOWSPEC` objects between the local RSVP implementation and client applications. The following paragraphs describe the basic functions of an RSVP application interface, using the

[11] A session is defined to be the collection of all data flows that are sent to the same destination by means of the same transport-layer protocol.

[12] See [SB95] for a conceptual description of OPWA.

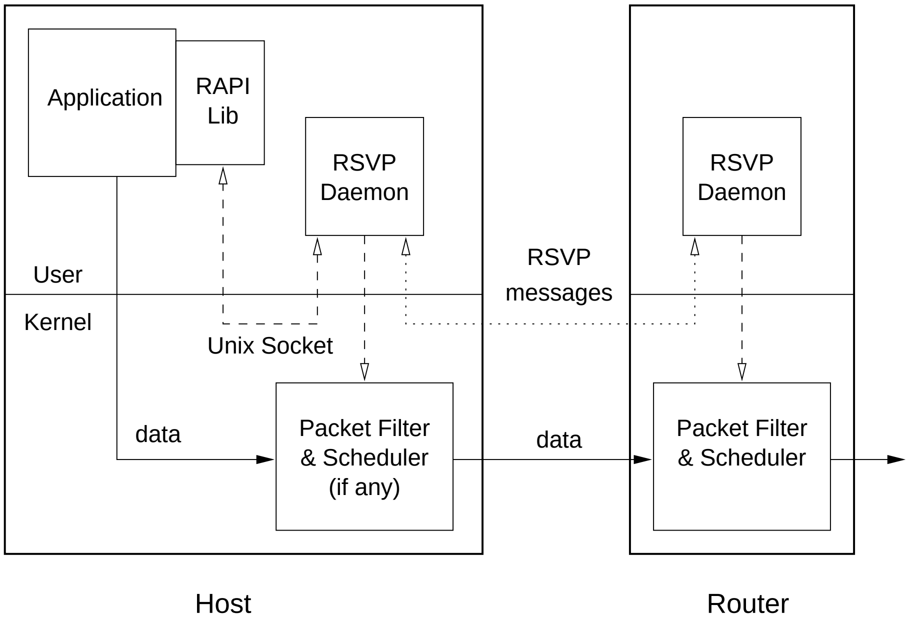

Figure 3.4: ISI RSVP implementation components.

RSVP implementation developed by the Information Science Institute (ISI) of the University of Santa Cruz as an example.[13]

The ISI implementation consists of an RSVP client library (RAPI), which is linked to application programs, and an RSVP daemon (rsvpd). Figure 3.4 shows how the different components fit together.

The RAPI library is described in [BH96]; the C function prototypes of the RAPI functions described below are summarised in Appendix B.4.

An application calls procedures from the RAPI library, which in turn interacts with the local RSVP daemon through a UNIX domain-socket. The RSVP daemon configures the host's packet filter and packet classifier, and communicates with other RSVP processes by means of normal RSVP messages.

The fundamental functions described in [BH96] are:

- *Creation of Sessions.*
  The `rapi_session()` call creates and initialises an API session and returns a session handle for further use. After a successful `rapi_session()` call, the application may receive up-calls indicating events and errors.
- *Specification of Sender Parameters.*
  An application that intends to send a flow of data must issue a `rapi_sender()` call. Thereby, it specifies the parameters of its data flow. The RSVP daemon uses this flowspec to create `PATH` messages for this flow. Repeated calls of the `rapi_sender()` function with different parameters simply change the flowspec.
- *Creation, Modification, or Deletion of a Reservation.*
  The `rapi_reserve()` function is used to make, modify, or delete a reservation for the flow described in the filter spec argument. The issuing of a `rapi_reserve()` call triggers the sending of `RESV` messages by the local RSVP daemon.
- *Removal of a Session.*
  The state registered with a session is deleted by calling the `rapi_release()` function. This

[13] The ISI RSVP implementation is available at `ftp://ftp.isi.edu/rsvp/release/`.

causes the RSVP daemon to tear down the reservations made on behalf of the released session by sending `RESVTEAR` messages, and to issue `PATHTEAR` messages in case the application acted as a data sender.

3.1.6 Internet Multicasting

As mentioned in Section 2.3.1, multicasting is essential to many multimedia applications. Therefore, in the sense of an integrated services model, QoS reservations and predictions, flow control, network management, addressing, and routing must be designed to support multicasting.

This section will examine several multicast techniques developed for IP networks. Other multicast-capable network or transport protocols, such as ST-II, XTP, or VTMP, that do not use IPv4, are not covered here.

IP Multicasting

IP multicasting is defined in [RFC1112] as an extension to the standard network-layer IP protocol.

Multicast host groups are identified by means of *Class D addresses*. Potential Class D addresses range from `224.0.0.0` to `239.255.255.255`. They can be used as destination addresses only.
A host group may either be *permanent* or *transient*. Permanent groups[14] have well-known, administratively assigned IP addresses.[15]

IP multicasting relies on external address and group management protocols to inform hosts about multicast group addresses and control membership access.
The *Internet Group Management Protocol (IGMP)*, as defined in [RFC1112], is used by IP hosts to report their group memberships to any immediately-neighbouring multicast(-capable) router. A host sends an IGMP group membership report message to a neighbouring router whenever it joins or leaves a multicast host group, or whenever it receives a query message, which is periodically sent by the router.

IP multicast routing uses modified *distance vector protocols*[16] to construct a spanning tree per group that covers all member hosts of the group.

Because TCP is only compatible with point-to-point connections and UDP offers no flow control or error correction mechanisms, other multicast-capable transport protocols must be used on top of IP multicasting.

Multicast Transport Protocol—MTP

MTP [RFC1301] is designed to be an efficient and reliable multicast transport-layer protocol that could be used on top of network-layer protocols such as IP multicasting. MTP leaves the allocation or dissemination of group addresses up to entities, which are not part of MTP itself.

A set of communicating processes, called a web, is governed by a principal member, called the master. The master controls the web's membership and operational parameters, which include rudimentary resource reservations. New members must register with the master for group membership, thereby requesting certain resources such as minimum bandwidth.

To synchronise the reception order of messages in the web, the master hands out transmit tokens to members who wish to send data.

To reduce control overhead, error recovery is implemented by a negative-acknowledgement (NAK) based selective-retransmission scheme. Nevertheless, the amount of overhead, especially the one generated by the ordering mechanism, might prove troublesome, because almost all control traffic passes by the master, which could turn out to become a bottleneck.

[14] The term is misleading: it is the address, not the membership of the group, that is permanent.
[15] Permanent group addresses are published in the "Internet Assigned Numbers", see [RFC1700].
[16] Such as the *Distance Vector Multicast Routing Protocol (DVMRP)*, see [RFC1075].

MGA, RAMP, MRP

Braudes and Zabale [RFC1458] analyse different multicast protocols, and propose a new protocol suite, which tries to eliminate the deficiencies of existing protocols.
This protocol set is designed to fit into the Integrated Services Architecture, and to take advantage of the information transmitted by reservation systems such as RSVP. The integrated protocol set comprises three cooperating components:

- *Multicast Group Authority (MGA).*
  The MGA manages the multicast address space (the Class D addresses), registers services and their QoS state, and controls group membership.
  The MGA is structured hierarchical, like the Domain Name System. Service requests are first made to a local agent, which may propagate the requests upwards in the hierarchy if necessary.
- *Reliable, Adaptive Multicast Protocol (RAMP).*
  RAMP is transport-layer protocol, which offers reliable as well as unreliable multicast service, using a network-layer protocol such as IP multicasting.
  Error recovery is done by a NAK-based selective-retransmission scheme. A retransmission of a packet set is performed by means of multicasting if a certain threshold of NAK messages for that set is exceeded; otherwise the packets to be retransmitted are sent by unicast, thereby seeking a compromise between unicast and multicast retransmission.
  Flow control is achieved by means of a rate-based approach.
- *Modified Routing Protocol (MRP).*
  MRP must be integrated into routers similar to IGMP. The tasks of MRP are path set-up, path tear-down, forwarding based on QoS values, and prioritised packet loss due to congestion.
  When making a routing decision, MRP considers not only the destination address in a packet—as conventional routing algorithms do—but also ensures that the QoS level offered by a possible path matches the level that is requested for a particular packet.

Multicast Backbone—MBone

MBone facilitates the worldwide broadcasting of digital audio and video, such as the live transmission of meetings of the Internet Engineering Task Force (IETF).
It is a virtual overlay network imposed on the Internet: MBone consists of multicast-capable islands each containing one or more *multicast routers (mrouters)*, which connect the islands logically by tunnels.
MBone packets are encapsulated in normal IP packets, which are sent as regular unicast packets from mrouter to mrouter. This encapsulation is necessary as long as not all routers on the way between two mrouters support multicasting. The tunnels are set up manually by configuring the tables of the mrouters.

The scope of multicast message is limited by the IP *time-to-live (TTL)* field, which is decremented by the weight that is assigned to every passed tunnel.

MBone multicasting utilises Class D IP addresses to identify multicast host groups, and the IGMP protocol to report group membership inside an island. Initially, the DVMRP protocol was used between mrouters. Newer research work proposes the *Multicasting OSPF (MOSPF)* [RFC1584] or *Protocol Independent Multicast (PIM)* [Hui95] protocols, which try to improve MBone routing.

3.1.7 Integrated Services in IP-ATM Networks

Experts of the IETF and of the ATM Forum are developing techniques for running IP on top of local area ATM networks. The proposed techniques are named *Classical IP*[17] and *LAN Emulation*,

[17] See [RFC1577].

respectively.

The problems of integrating real-time services in the IETF Classical IP architecture are summarised in [RFC1821]. The main issues are the service model, the reservation model, and QoS-aware routing.

As described above, the service model of Integrated Services Internet architecture used service classes—such as best effort, guaranteed, predictive delay, and controlled delay—for QoS specification. ATM in turn has five service classes, which are not in direct correspondence with the IP classes:

- *CBR (constant bit rate),* which emulates a leased line, providing very tightly constrained delay and designed for applications which can use a fixed bandwidth pipe.
- *VBR (variable bit rate)–real-time,* which attempts to constrain delay for applications whose bandwidth requirements vary.
- *VBR–non–real-time,* intended for variable bandwidth applications without tight delay constraints.
- *UBR (unspecified bit rate),* which most closely approximates the best effort service of traditional IP.
- *ABR (available bit rate),* which uses a complex feedback mechanism to control data loss.

The differences between the reservation models of RSVP and ATM (UNI 3.0) signalling are listed in Table 3.2.

Category	RSVP	ATM UNI 3.0
Orientation	Receiver-based	Sender-based
State	Soft state	Hard state
QoS Setup Time	Separate from route establishment	Concurrent with route establishment
QoS Changes	Dynamic QoS	Static QoS

Table 3.2: Differences between RSVP and ATM UNI 3.0.

3.2 Multimedia Transport Protocols

3.2.1 Requirements and Benefits

Existing general purpose protocols, such as TCP or UDP, have proved very successful at meeting the demands of classical applications like electronic mail or file transfer. However, these protocols turned out to be unsuitable for distributed multimedia applications with real-time requirements.

TCP for example is not capable of multicasting; in addition, the retransmission of one lost packet stops an application from processing other packets until the lost one finally arrives. But by the time a late or retransmitted packet arrives, it might be useless for a real-time sensitive application.

UDP, on the other hand, incurs considerable less overhead than TCP and does not delay an application in cases of packet loss, but offers no support for the detection of packet loss or miss-ordered packets. Both UDP and TCP do not guarantee minimum transmission rates or upper bounds on the end-to-end delay.

A list of features desirable for multimedia real-time transmission protocols includes:

- Application control over error detection and recovery measures.
- Support for multiplexing of data streams.
- Encryption capability.

- Timestamps for real-time media.

In the remainder of Section 3.2, I will first present the Real-Time Transport Protocol (Section 3.2.2), which fulfills the listed requirements and offers some additional features.

3.2.2 RTP

RTP Overview

RTP is designed as an end-to-end transport protocol that is suitable for unicast or multicast transmission of real-time data, such as interactive audio or video.

RTP follows the two principles outlined by Clark and Tennenhouse in [CT90]:

Application Level Framing. The key *architectural* principle is Application Level Framing. The idea is that the application should break down the data into aggregates, which are meaningful to the application, and which do not dependent on a specific network technology. These data aggregates are called *Application Data Units (ADUs)*. The frame boundaries of ADUs should be preserved by lower levels, that is, by the network system.

The rule, by which the size of an ADU is chosen, states that an application must be able to process each ADU separately and potentially out of order with respect to other ADUs. So, the loss of some ADUs, even if a retransmission is triggered, does not stop the receiving application from processing other ADUs. Therefore and to express data loss in terms meaningful to the application, each ADU must contain a name that allows the receiver to understand the place of an ADU in the sequence of ADUs produced by the sender. Hence, RTP data units carry sequence numbers and timestamps, so that a receiver can determine the time and sequence relation between ADUs.

The ADU is also the main unit of error recovery. Because each ADU is a meaningful data entity to the receiving application, the application itself can decide about how to cope with a lost data unit: real-time applications, such as digital video broadcasting, might prefer to simply ignore a few lost frames instead of delaying the presentation until the lost frames are retransmitted, whereas file transfer applications cannot accept the loss of a single data unit. In addition, if the application has the choice of how to deal with a lost unit, the sender can decide whether to buffer the data units for potential retransmission, to recompute the lost units if requested again, or to send new data that which diminishes[18] the harm done by the loss of the original ADU.

Integrated Layer Processing Because application level framing breaks down the data in pieces that an application can handle separately from other data units, all processing of a single, complete ADU can be done in one integrated processing step for reasons of efficiency. This *engineering* principle is called Integrated Layer Processing.[19]

While the authors of [CT90] agree that layered isolation, as employed in conventional protocol stacks, is suitable for the network layer and below, they argue that many of the functions of the transport and above layers could be structured in a way that would permit the use of the more efficient Integrated Layer Processing.

Especially for RISC systems, whose performance is substantially limited by the costs of memory cycles, an integrated processing loop is more efficient than several, isolated steps that each read the data from memory, possibly process it in someway, and write it back to memory.

RTP is used in combination with other network or transport protocols—typically on top of UDP—as visualised in Figure 3.5. The RTP specification recommends the use of two different

[18] For example in a video conference, if an MPEG I-frame got lost, the following B- and P-frames would make no sense to the receiver; but by the time the lost I-frame got retransmitted, it would arrive far too late for displaying. Instead, a new I-frame of the current scene, which would be too early compared to the normal MPEG frame rhythm, would be sent.

[19] Integrated Layer Processing is a principle for the *implementation* of a protocol. Layering is nevertheless considered useful for protocol *design*.

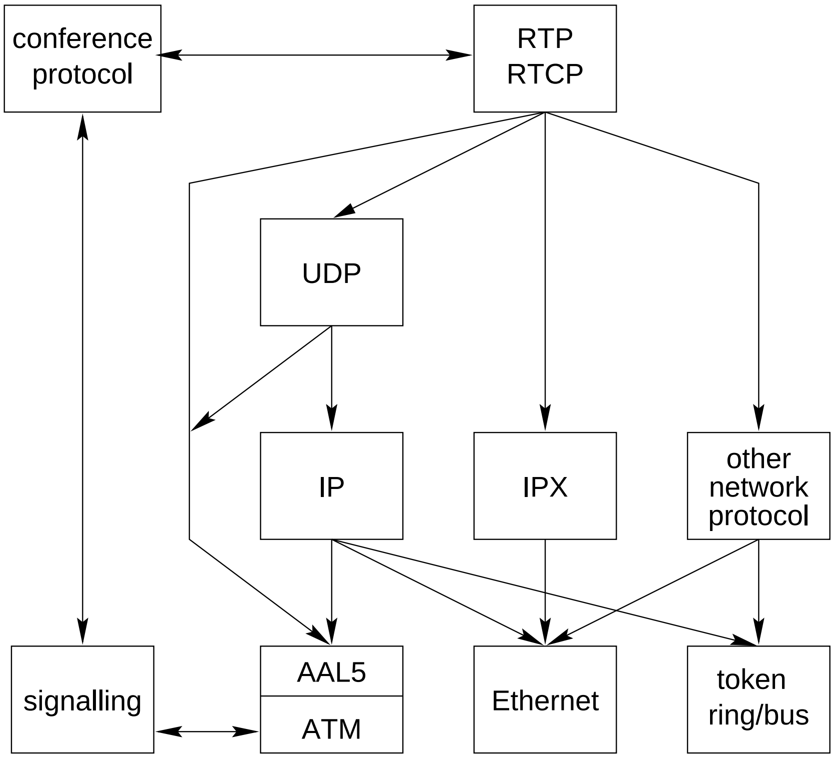

Figure 3.5: RTP in relation to network and transport protocols.

generalised destination ports within one RTP session, one port for the reception of RTP payload packets and one for RTCP control packets. If RTP is used in conjunction with RSVP, one RTP session corresponds to two RSVP sessions, which are identified by the addresses of the RTP data and control destination ports, respectively.

RTP supplements the data delivery functions of the underlying protocols with the following services:

- Payload type identification.
- Sequence numbering.
- Timestamping.

It should be noted that RTP itself does not make any QoS commitments, does not guarantee reliable, timely, or in-order delivery, and does not enforce any error treatment measures. However, extensions, such as those described by Parnes in [Par96], add reliability to RTP for applications that cannot tolerate packet loss, for instance white-board applications.

The accompanying *RTP Control Protocol (RTCP)* facilitates monitoring of the delivery QoS, and conveys rudimentary information about the participants of an RTP session.

RTP and RTCP are defined in [RFC1889]. However, this specification serves only as a protocol framework; RTP is intended to be tailored to the needs of particular applications by additional specifications:

- *Payload format specification documents*, which define how a particular payload is to be carried in RTP. Currently, payload format specification RFCs exists for H.261 video streams [RFC2032], for CellB video encoding [RFC2029], for JPEG-compressed video [RFC2035], and for MPEG video [RFC2038].

- A *profile specification document*, which defines a set of payload type codes and their mapping to payload formats, as well as any extension or modification of the RTP message headers that are necessary for a specific class of applications. A profile for audio and video conferences is given in [RFC1890].

RTP Payload Transmission

[RFC1889] introduces the term of an *RTP session* to describe a set of participants that communicate via a particular pair of destination transport address, for example a multicast IP address together with two UDP ports for control and payload data. Within one session, only a single payload type and the associated control information are transmitted; this rule has several consequences:

- Multiple RTP sessions must be used for *multi*media applications.
- Demultiplexing of the received data streams by the payload type is avoided, following the principle of integrated layer processing.
- Different QoS reservations can be established for each session in accordance with the different needs of each medium and each receiver's possibilities.

An RTP packet consists of a fixed header, a variable-length list of sources that contributed to the payload in the packet, a potential header extension as defined in a profile specification, and the actual payload, which is encoded according to a payload format specification.

The fields of highest importance in the RTP packet header are:[20]

- *Synchronisation source (SSRC) identifier.* A receiver uses the synchronisation source identifier to demultiplex the packets received via one destination port. Therefore, each SSRC identifier must be unique within one RTP session.
  All packets from one synchronisation source fit into the same timing and sequence numbering space.
- *Sequence number.* The synchronisation source increments by one the sequence number of each RTP data packet sent in one session. This number can be used by a receiver to restore the sequence of packets disordered during the transmission, to detect packet loss, and to compute the number of expected and lost packets.[21]
- *Timestamp.* The timestamp describes the sampling time of the first octet of the payload data carried by the packet. The sampling instant must be measured by means of a clock, which produces timestamps increase monotonically and linearly in time. The clock frequency that is used for the mapping of timestamps to wall clock time should be layed down in a protocol or payload specification, or could be recomputed with the help of NTP timestamps sent within RTCP packets.
  A receiver can use the timestamps for intra- and inter-stream playout synchronisation as well as for inter-arrival jitter estimation.[22]

An RTP data packet does not contain any length indication; an application must either rely on the protocol used beneath RTP to provide framing mechanisms, or, in case the underlying protocol provides a continuous octet stream abstraction, define a method of encapsulating RTP packets in the octet stream.

[20] See Appendix A.1 for the complete structure of the RTP packet header.
[21] See [RFC1889, A.3] for an algorithm that determines the number of RTP packets expected and lost.
[22] See [RFC1889, A.8] for an algorithm that calculates the inter-arrival jitter.

Control and Feedback Information

RTCP control packets[23] are sent periodically (with small random time variations to avoid traffic bursts) to the same (multicast) host address as the RTP data packets. Their content facilitates:

- *QoS monitoring and congestion control.* The primary function of RTCP messages is to provide feedback on the quality of the data distribution. The conveyed information can be used for flow and congestion control, to control adaptive encodings, and to detect network faults. Applications that produce payload data generate *RTCP sender reports.*[24] These reports contain counters of sent packets and octets that allow other session participants to estimate a sender's data rate.
  Applications that have recently received packets issue *RTCP receiver reports*, which contain the highest sequence number received, loss estimations, jitter measures, and timing information needed to calculate the round-trip delay between the sender and the receiver.[25]

- *Media synchronisation.* Sender reports contain an NTP timestamp and an RTP timestamp. The RTP timestamp describes the same instant as the NTP timestamp but is measured in the same units as the timestamps issued in the data packets of the sender. These two timestamps allow to synchronise a receiver's playout clock rate with the sampling clock rate of the sender. In addition, a receiver can use this RTP/NTP timestamp correspondence information to synchronise the playout offset delay of related streams.

- *Member identification.* RTP data packets and most RTCP packets carry only an SSRC identifier but convey no other context data about a session participant. Therefore, *RTCP source description packets* are sent by both data senders and receivers.[26] They contain additional information about the session members, especially the obligatory *canonical name (cname)*, which identifies a participant throughout all related sessions independently from the SSRC identifiers. The representation of the canonical name must be understandable by both humans as well as machines. Other conveyed information includes email addresses, telephone numbers, the application name as well as optional application-specific information. An application that decides to leave a session must transmit an *RTCP bye packet* to indicate that it will not participate in the session anymore.[27]

- *Session size estimation and control traffic scaling.* The total control traffic generated by all participants in a session should not exceed a small percentage of the data traffic, typically 5 percent. The control messages from other session members enable a participant to estimate the session size and to adapt its control packet rate, so as to remain within the limit of the control traffic share.[28] So, RTP scales up to large numbers of participants.

Different schemes have been developed to exploit delivery feedback information in order to estimate the network congestion state and to adjust the application bandwidth accordingly. These schemes work for applications that can scale their media quality and their data rates to a given bandwidth value.
Busse, Deffner, and Schulzrinne propose a simple though interesting and workable mechanism [BDS96]. The fraction of lost packets, conveyed by normal RTCP receiver reports, is used as the primary indicator of the congestion state of the network:[29] a data sender classifies the network path to each receiver as `UNLOADED`, `LOADED`, or `CONGESTED`. Depending either on the average classification of all paths or on the worst path, the sender makes an adjustment decision about its sending behaviour: it may increase, hold, or decrease its application bandwidth.

[23] See Appendix A.2 for the basic RTCP packet types and their structure.
[24] See A.2.1 for the structure of RTCP sender reports.
[25] See A.2.2 for the structure of RTCP receiver reports.
[26] See A.2.4 for the structure of RTCP source description packets.
[27] See A.2.3 for the structure of RTCP bye packets.
[28] See [RFC1889, A.7] for an algorithm computing the RTCP transmission interval in dependence on the session size and on the RTCP target bandwidth share.
[29] The usage of the jitter information is subject to further work.

Bolot, Turletti, and Wakeman take a similar but more complex approach [BTW94]. Instead of using the information from standard RTCP messages, a sender solicits congestion state information from a subset of the receivers, and then decides about appropriate bandwidth adjustments.

3.3 Case Studies

Two examples of multimedia applications, namely multimedia conferencing and media on demand, will be examined in this section. I will apply the considerations from Section 3.1 and Section 3.2, and will present further mechanisms, which are specific to these application classes.

3.3.1 Multimedia Conferencing

The typical media ingredients of a multimedia conference, as part of computer supported cooperative work, include audio, video, and perhaps a shared whiteboard or similar. These media each need specific treatment:

- The intelligibility of digitised audio is very sensitive to missing samples and playout jitter. A bandwidth share of 64 kbps is necessary for the uncompressed transmission of one audio source at telephone quality. Compression mechanism can further reduce the data rate. In any case, the bandwidth demand is small compared to the capacities of current LANs, whereas packet loss due to congestion occurs often over the Internet.

- Video generated by a typical computer-mounted camera is less sensitive to loss and jitter than audio, but the data rate even of compressed video is much higher than the audio data rate.
  Because audio and video conferences are real-time interactive applications, a low transmission latency is required for both media; the low-latency requirement further implies that buffering cannot be used in larger scale to compensate for transmission jitter, as could be done on the receiving end of video-on-demand systems.
- Compared to the other two media, whiteboard applications generate very little traffic. Their real-time demands, concerning delay, latency or jitter, are also comparably low. However, the reliable transmission of every single event is absolutely necessary to give each participant an identical, consistent view of the virtual board.

The Real-Time Transport Protocol described above in Section 3.2.2 is well suited to be used as the transport protocol for conferencing applications. For multicasting-capable IP networks, encapsulation of audio and video RTP packets in multicasted UDP packets is a common and working practice. Additional measures should be taken to ensure reliable transmission of whiteboard application data with RTP.[30]

The transport of audio and video data with RTP in conferencing applications is described in a profile for audio and video conferences [RFC1890]. It provides interpretation of generic fields within the RTP and RTCP headers suitable for audio and video conferences that were left open by the RTP specification [RFC1889]. It also states guidelines for a variety of different audio and video encodings, as well as a set of default mappings from payload type numbers to these encodings.

If available, resource reservation mechanisms can be used, especially for widely distributed sessions. However, QoS-control–capable network equipment is not very common yet and is technically incompatible with the popular bus-style Ethernet LANs. So, monitoring of the delivery by means of RTCP receiver reports and corresponding adjustment of the encoding to the congestion state is advisable, as proposed for example in [BDS96] and [BTW94].

However, RTP and other transport-level protocols provide no or only minimal support for higher level session management functions, such as session announcement, session description, and conference control.

[30] As mentioned in Section 3.2.2, see for instance [Par96].

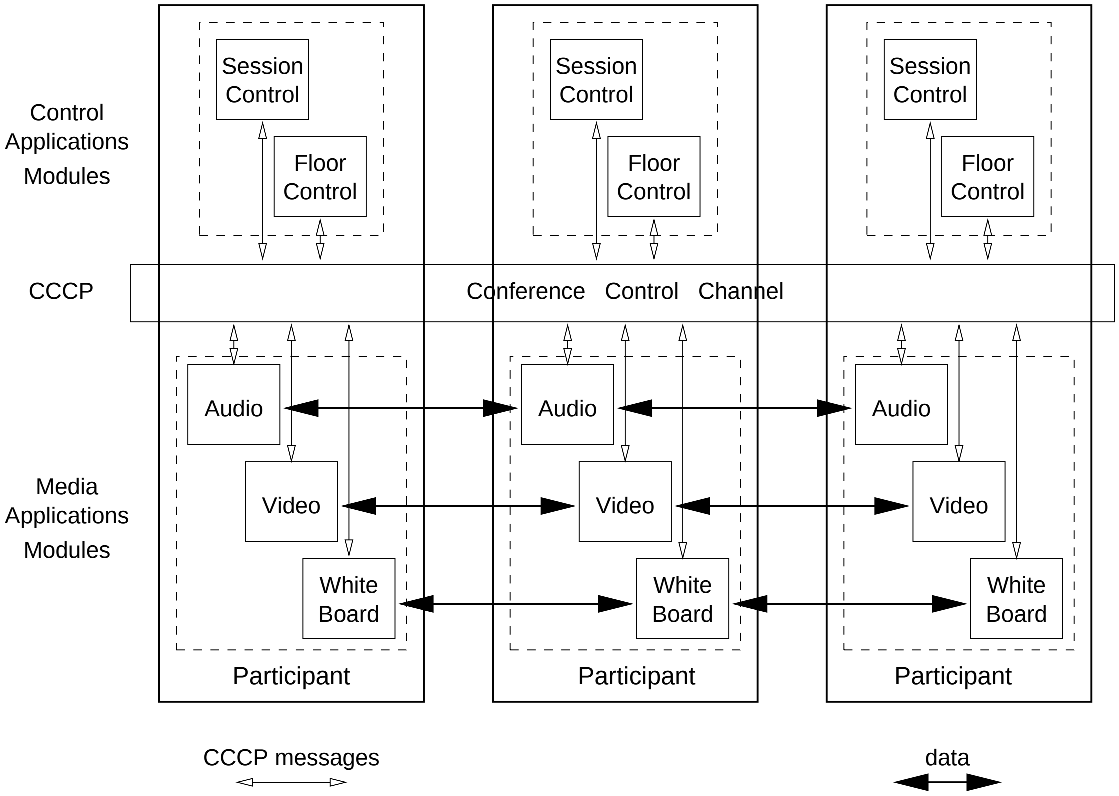

Figure 3.6: Conference Control Channel Protocol.

Handley, Wakeman, and Crowcroft summarise the main tasks of conference control [HWC95]:

- *Application control.* The parametrising of applications with a correct initial state.
- *Membership control.* Information about the current conference members and their tools and applications.
- *Floor management.* Control over the input to particular applications.
- *Network management.* Requests for end-to-end media connection establishment and tear down, and requests from the network to change bandwidth usage because of congestion.
- *Meta-conference management.* Initiation, announcement, invitation and termination of conferences.

The authors propose their *Conference Control Channel Protocol (CCCP)* for performing the mentioned conference control tasks except for meta-conference management. The operation of CCCP is depicted in Figure 3.6.

The Multiparty Multimedia Session Control (MMUSIC) working group of the IETF is developing a related set of protocols that addresses the issues related to meta-conference management:

Session Description Protocol (SDP). SDP [HJ96] is intended to describe the properties of multimedia sessions for purposes of session announcement or session initiation. The distributed information can comprise:

- Session name and purpose.
- Time during which the session is active.
- Media conveyed in the session.
- Information necessary to receive those media (such as addresses, ports, formats).

- Bandwidth estimations to facilitate resource reservations.
- Contact information for the initiator or session manager.

Session Announcement Protocol (SAP). The Session Announcement Protocol [Han96] is used to advertise a multimedia conference or session. To announce a session, a client periodically multicasts SAP packets to a well-known multicast address and port, using the same TTL as the session itself. An SAP packet consists of an SAP header and a payload block, which is in turn an SDP message describing the session.

Session Initiation Protocol (SIP). SIP [HSS96] is used to invite individual users to participate in multimedia sessions. It communicates via location servers and user agents, thereby allowing mobility by relaying and redirecting invitations to a user's current location. It can also be used to invite media recording or playback servers into a session. SIP also makes use of SDP.

3.3.2 Media On Demand

No standards for home video-on-demand distribution networks have been accepted yet. While the backbone of a video-on-demand network would probably be based on ATM and/or SONET, more and differing techniques have been proposed by telephone, cable TV, and other companies for the local distribution. Tanenbaum [Tan96, p. 750–753] identifies the four most important approaches as:

- Asymmetric Digital Subscriber Line (ADSL)
- Fiber To The Curb (FTTC)
- Fiber To The Home (FTTH)
- Hybrid Fiber Coax (HFC)

The outcome of this competition is still open, as well as the decision about a protocol stack. ATM in general seems preferable, but the choice of the appropriate adaption layer is more difficult; a modified version of AAL 5, not discarding blocks that contain bit errors, could turn out suitable. Figure 3.7 shows a potential stack, comprising ATM and its sublayers, potentially RTP, MPEG and its layers, and applications on top.[31]

The Internet community is also developing mechanisms and protocols that allow the integration of media servers in multimedia sessions. The IETF MMUSIC working group for example is working on the *Real-Time Streaming Protocol (RTSP')*. This is an application-level text-based protocol intended to control the delivery and recording of real-time data. Its tasks comprise the choice of delivery channels—such as (multicast) UDP and TCP—and delivery mechanisms based on RTP, the exchange of session descriptions, and the control of data streams by commands such as `PLAY`, `RECORD`, or `STOP`. RTSP' is capable of controlling multiple delivery sessions at once, which can include both live-fed and stored media clips.

Session descriptions, encoded for example in the SDP format, are retrieved from a web server or the media server itself, typically using HTTP. Each media stream listed in a session description is identified by an RTSP URL, which points to a media server that provides the particular stream.

[31] Figure 3.7 is adapted from [Tan96, p. 754] with the addition of the RTP layer.

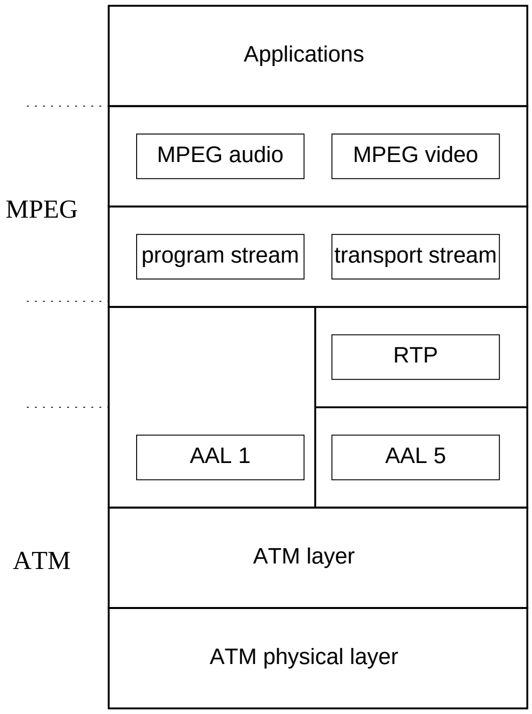

Figure 3.7: A video-on-demand protocol stack.

Chapter 4

Multimedia Data Transmission and Object Orientation

This chapter provides a conceptual link between object-oriented multimedia-application design and multimedia-data transmission. I will first review the object-oriented paradigm briefly in Section 4.1, and will then give an overview of object-oriented design approaches for multimedia applications in Section 4.2. Thereafter in Sections 4.3 and 4.4, I will will describe in some detail data streams and object streams, respectively.

4.1 The Object-Oriented Paradigm

Object-oriented analysis, design, and programming are proven powerful means to master the complexity inherent in the development process of large, distributed applications.

The ingredients and benefits of object orientation are well described by Booch in [Boo94, S. 2]: the object-oriented conceptual framework, called the *object model*, is based on the combination of the principles of *abstraction, encapsulation, modularity,* and *hierarchy*, often supplemented by the principles of *typing, concurrency, and persistence.*

According to Booch, the usage of the object model for the design process of an application offers several advantages over traditional techniques:

- The application of the object model helps substantially to develop complex programs.
- Only object-oriented analysis and design permit to fully exploit the expressive power of modern object-oriented programming languages.
- The object model facilitates the reuse of both code and designs.
- Systems designed according to object-oriented principles can evolve over time and can be adapted easily to accommodate future demands, without necessitating the abandon or complete restructuring of the original design.
- Object orientation reduces the risks of software development.
- The object model resembles human cognition more closely than traditional design paradigms.

The object-oriented paradigm—especially the concepts of *object-oriented decomposition, inheritance, specialisation, and polymorphism*—are particularly well suited for multimedia application programming because:

- object-oriented decomposition leads to meaningful abstractions that allow to construct and handle composed media objects conveniently,
- polymorphism and inheritance hide specific details of particular hardware devices from the rest of the program,

- specialisation offers the flexibility necessary for adding support for new media types and hardware devices easily.

4.2 Overview of Object-Oriented Design Approaches for Multimedia Applications

The object-oriented design process produces a hierarchy of classes, from which a collection of objects is instantiated to built a particular application. However, different points of view and different contexts lead to very dissimilar class hierarchies. Steinmetz et al. summarise some fundamentally differing approaches in [SF92] and [Ste94, p. 491–514]:

Application-related class hierarchies. The class hierarchy resembles the abstractions created for only one application or a small set of related programs; the chance of reusing these classes is very low.

Generic application class hierarchy. The class hierarchy is created by concentrating basic functionality that is part of a wide range of applications in common (base) classes. Applications are built by developing and instantiating specialised subclasses.
The abstractions that are encapsulated in the class hierarchy must be generally applicable on the one hand but meaningful and powerful on the other hand. Steinmetz [SF92, p. 399] proposes to regard media as perception media, storage media, transmission media, or presentation media in order to find common functionality.

Device-associated–objects. The objects reflect the physical hardware devices of a system and represent the behaviour and the interfaces of a device. Naturally, it is advisable that the interfaces to objects that provide similar semantics by using different devices, should be defined in hardware-independent base classes. Code based on this approach can be executed in parallel rather easily. However, finding a universally usable interface to audio or video I/O devices is no trivial task. The integration of additional functionality such as synchronisation often leads to complex designs that must make full use of multiple inheritance.

Data flow principle. The class hierarchy derived from the data flow principle consists of source classes, sink classes, intermediate processing classes, and connecting objects. Within this "Lego"-like model, applications are built by forming a data flow path of connected objects, which are instantiated from appropriate classes. The individual elements of the stream can be either unstructured chunks of memory or first class objects, which not only carry state information, but also offer presentation, transport, or storage functionality. Simple data streams are discussed in Section 4.3, whereas object streams are further described in Section 4.4.

Communication-oriented class hierarchy. Distributed multimedia applications require the consideration of communication issues. Blakowski [Bla91] describes a set of classes, which puts major emphasis on communication aspects: he distinguishes information classes, presentation classes, and transport classes. The data contained in information objects can be used to form presentation objects, which are used for presenting the information. Alternatively, the information objects can be converted into transport objects for transmission. Steinmetz [SF92, p. 399] proposes the addition of storage classes.

Naturally, combinations and hybrids of these approaches are possible and useful.

4.3 Data Streams

The data flow principle appears to be very useful for designing applications that process stream-like multimedia data. I will first discuss the data flow principle by means of an exemplary object-oriented framework developed by Gibbs et al., and will then demonstrate how network support can be integrated into a data-stream–oriented class hierarchy.

4.3.1 Components

The basic building blocks of Gibbs' framework, as described in [Gib91] and [GDT92], are *multimedia objects*, which have the following properties:

- A multimedia object is an *active object*, which means, that it has a state (its instance variables) and a behaviour (its methods) like any ordinary *passive object*, but may in addition spontaneously perform actions without the reception of messages from other objects. Hence, a private *thread of control* must be assigned to each multimedia object. Depending on the programming environment, the thread of control can be implemented for example as a separate process or a user or kernel level thread.

- A multimedia object produces and/or consumes multimedia data at certain rates, thereby becoming subject to real-time and synchronisation constraints.

- Each multimedia object contains one or more *ports* through which it exchanges data with other objects. The number of ports, the data type that is communicated through a port, and the direction of the port are determined by the class of the multimedia object.

- Multimedia objects can be divided into three categories according to their possible activities, which can be characterised by the direction of the ports:

  - *Sources* produce multimedia data, which they sent to other objects through *output* ports.
  - *Sinks* consume data, which they receive through their *input* ports.
  - *Filters* or *processing objects* receive data through *input* ports, process the data, and transmit the results through their *output* ports.

  Connectors are used to link the output ports of the data producing objects with the input ports of the data consuming objects.

- A multimedia object can either be a software-based object, which manipulates a process associated with the object, or it can be a hardware-based object, which issues commands directly to the hardware.

Figure 4.1 depicts the notation for multimedia objects that is layed down by Gibbs in[Gib91]. Composite objects can be built by combining basic multimedia objects and specifying the connections between the various components. Of course, composites are active multimedia objects themselves. As a consequence, an invocation of the control and synchronisation methods of a composite must in turn invoke the corresponding methods of the components with appropriately coordinated parameters.

4.3.2 Integration of Network Communication

A framework based on the data-flow principle can be easily extended by network communication classes. However, the code and the responsibility for networking can be placed in different classes:

- Gibbs [Gib91, p. 11–12] suggests *specialised sink and source classes*. On the sender side, a sink object, connected to a source object, packetizes and transmits data over the network. On the receiver side, the object that accepts data packets from the network, acts as a source object and forwards the received data through its output ports to filters and sinks. The example of an audio talk application presented in Figure 4.2 visualizes this approach.

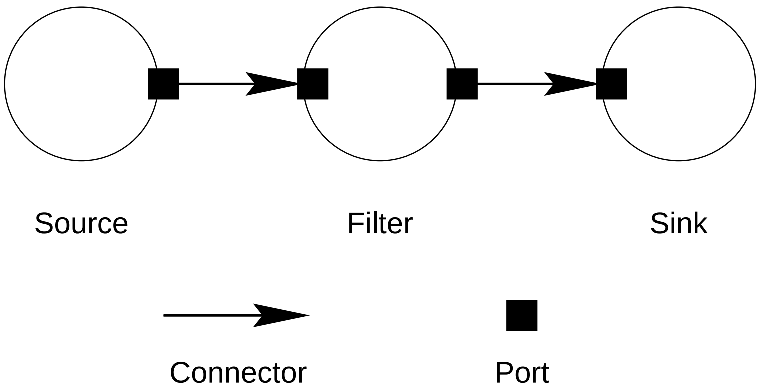

Figure 4.1: The data flow principle: sources, filters, and sinks, communicating by means of ports and connectors.

- Another possible approach, which is depicted in Figure 4.3, employs specialised connector objects that can connect a source object and sink objects across a network.

If the presentation start time s_i and duration d_i of the individual media elements e_i is not contained directly in the encoding of the e_i or known implicitly, the network communication objects must transmit the timing information together with the transport representation of the e_i by means of appropriate protocols such as RTP described in Section 3.2.2.

In addition, the class collection must be enriched by classes that can cooperate with the QoS control interface of the network system. Besides, the collection must incorporate classes that can communicate session descriptions to remote sites.

4.4 Object Streams

4.4.1 Differences from the Data Stream Model

Object streams are related to simple data streams because both models use the idea of media elements travelling through a set of sources, intermediate processing objects, and sinks. However, in the object-stream model, the elements passed on are first-class objects, whereas in the data-stream model mere pieces of memory are communicated. Such a media element object hides the memory chunk, which contains the media data and which is the basis of the internal state of the object, by offering higher-level abstractions to the processing objects.
It seems advisable to divide the abstractions and functionality between the media element classes on the one hand and the sink and source classes on the other hand according to the following rules:

- The media element classes should at least restrict access to an object's media data by allowing access only through accessor methods.

- If the encoding of the media element contains any information about the element that is useful to processing objects and that is independent of the element's media type, the element object should extract this information from the memory chunk and should make it available to processing objects through appropriate accessor methods. Such information could comprise sequence numbers timestamps, or the size of the memory chunk.

- The media element objects might also contain part of the display, persistence, transport, or synchronisation functionality. The task of presentation objects would then consist of the

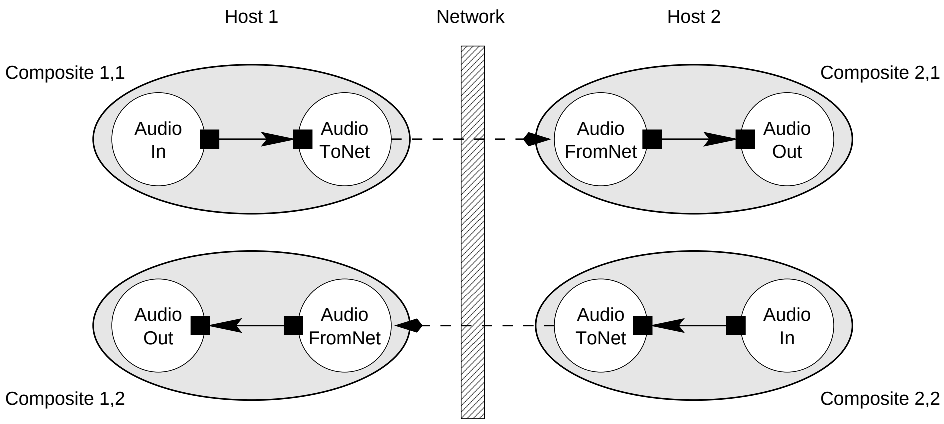

Figure 4.2: An audio talk application using specialised sink and source objects for network communication.

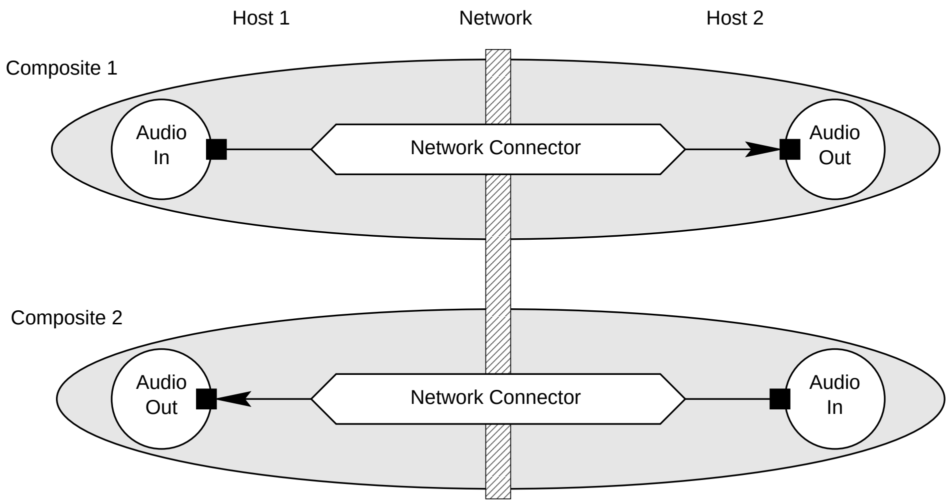

Figure 4.3: An audio talk application using specialised connector objects for network communication.

coordinated invocation of the synchronisation or presentation methods of the media element objects.[1]

4.4.2 Object Reification

The use of objects as application data units causes the dynamic creation and destruction of a very large number of media element objects. The basis of the internal state of these objects is a piece of memory that is the result of the activity of a sampling device or the network subsystem. But in a pure "objects-only" system a piece of memory is an unknown concept. This leads to the question: how can a piece of memory be made accessible as an object?

Husemann [Hus96, S. 4] describes how both individual media elements, which he calls *chunks*, as well as whole data streams can be turned into chunk objects and multimedia object streams, respectively. He names this technique *object reification*.
The aim of *reification* in general is to make a foreign concept of a higher or lower level visible on the current level of perception. This mechanism is classically used to achieve *reflection*, that is the modification of an object's attributes, which are normally not visible and accessible at the level of the object itself. These concepts or attributes comprise for example inheritance, class membership, storage, or existence.

The work of Husemann is embedded in the PM project[2]. In the PM object model, objects that exist on a meta level implement abstractions for the objects of the corresponding base level. Reification is achieved by the *projection* of meta-level objects into the base level. Because of PM's homogeneous object model, the projected meta objects are accessible through the same method invocation interface as normal base-level objects. The interaction of real base-level objects with projected meta-level objects is called *meta interaction*.

Projecting pieces of memory, which exist on the operating system meta level,[3] into the base level yields reified chunk objects, hence the term *object reification*. The ingredients [Hus96, p. 82] of a reified object are a chunk and an incomplete base-level object, which contains the necessary pieces—for example class and method information—to complement the chunk and to form a complete object.

Object reification is very similar to the instantiation of new objects: both operations take place on a meta level, both need a description of the class[4] of the object to be created, and both introduce a new object on the base level. However, object reification works differently from object instantiation during the following steps of the respective operations:

- *Memory allocation.* Instead of allocating new memory for the state variables of the reified object, the memory of the chunk is used.
- *Variable binding.* The binding of the internal variables of a reified object to memory pieces of the chunk can prove relatively complicated. Hence, object reification produces mainly objects with a rather simple set of variables.
- *Start-up initialisation.* The incomplete object is already initialised as far as possible prior to the reifying projection. Additional start-up code must be executed if it exists.

To keep the variable binding simple, Husemann suggests that only PM chunk objects can be reified from chunks. The reified chunk object can, if necessary, behave like an object of another type by means of *algorithmic type adaption.* Through algorithmic type adaption, a chunk object can behave like a video frame object, for example. *Glue code* must be specified, which describes how the methods and variables of one type are mapped to the respective ones of the other.

[1] The chunk objects of Husemann's video-on-demand case study [Hus96, Section 5.4] offer a display method, which synchronizes a chunk object with related chunk objects before it displays the object's content on the hardware device associated with the calling player object.

[2] See [HEF+91] for the initial description of the PM project in general and the PM object model in particular.

[3] Namely, the chunks might have been generated in the network subsystem or some hardware device driver.

[4] The PM model uses the more general notion of *shapes* instead of classes, see [Hau95].

In contrast to PM's object-model language ALCAID[5], common hybrid object-oriented programming languages—such as C++ or Java—do not support mechanisms such as algorithmic type adaption or reifying projection. In compensation of these shortcomings, other features of these languages can be exploited:

C++. C++ is a hybrid language. As a consequence, it includes primitive data types in addition to first level objects. In particular, C++ facilitates direct access to memory regions by treating pieces of virtual memory as arrays of 8-bit characters. Pointers to memory regions and arrays of elements of any type are also primitive data types.
The "equivalent" of object reification consists of initialising a pointer in a chunk object to the address of the piece of memory that contains the chunk's media data. Further start-up operations can include the initialisation of some state variables by interpreting the contents of the chunk, for example by extracting a sequence number or media type information.

Java. Java includes the primitive data type `byte`. However, arrays of elements of any type or class are first-class objects. Pointers or memory addresses of arrays are not accessible. Programmers cannot influence the object reification mechanism: a reifying projection is done by the Java runtime system only at the instantiation of a byte array, but the result is always a reference to an array object consisting of the specified number of bytes, which are all initialised to the value `0`.
In an application based on the object-stream model, byte arrays are filled with media data by source objects, for example by socket objects or objects representing hardware devices. Chunk objects offering more functionality or additional state variables are reified, that is initialised, by storing a reference to a byte array object in an internal variable and executing some start-up code.

[5] See [Hau95] for a description of ALCAID.

Chapter 5

Design of a Multimedia Transmission Architecture

In this chapter, I propose an architecture, which I designed to fulfill the real-time transmission needs of distributed multimedia applications. A partial implementation of this design, using the Java Development Kit, is described in the next chapter, starting on page 49.

5.1 Design Goals

The fundamental purpose of the set of classes and types constituting the architecture is to offer real-time transmission functionality and interfaces to network resource reservation mechanisms. The architecture is intended to serve as a universally suitable communication framework for distributed interactive multimedia applications, such as tele conferencing or computer supported cooperative work applications, or small-scale media-on-demand clients and servers.
In addition, the architecture comprises an object-stream–based structure for the multimedia data flow inside an application itself. This structure resembles the ideas of Husemann[1] to some extent. However, the main issue of my work is the integration of network communication.

The two main design goals that substantially shaped the architecture, are:

- *Uniform model of the intra-application data flow.* To achieve a uniform model of the horizontal, intra-application data flow, the local representation of a remote medium source should fit in the local data flow in the same way as a local source. However, this abstraction must not entail any limits on efficiency.

- *Access to network information and control over QoS mechanisms.* The objects that represent the network access points must be equipped with interfaces, which grant access to the QoS monitoring and control facilities offered by the employed network techniques or transport protocols. Other information available from or through the network, such as data about the remote session participants, must be made accessible, too.

To put it in other words: the objects that offer transmission services to an application have two faces. On the one hand, they comply to the appropriate types or interfaces of the elements constituting the local data flow path. On the other hand, all information that can be gathered from the network—for example about session participants or the network state—must be accessible and, if possible, controllable through another set of well-defined interfaces to these objects.

Naturally, general goals—such as modularity, easy expandability, or efficiency—must be kept in mind, too.

[1] See Section 3.5 and Section 4.5 of [Hus96].

5.2 Architectural Overview

Figure 5.1 illustrates the overall structure of the architecture.

The main objects, through which the media elements travel inside an application, are *Producer*, *StreamExporter*, *Stream*, and *Consumer objects*. Producers generate multimedia data streams, which the respective StreamExporters distribute by inserting the media elements into Stream objects. Consumers extract the media elements from their associated Stream objects and process the data in their specific way, for example by displaying or storing it. The purpose of the Stream objects is to decouple the Producers and Consumers by storing the media elements until a Consumer requests them. Therefore, the Stream objects might apply flow control mechanisms to avoid unnecessary memory consumption.

Network transmission of multimedia data streams is effected by means of *Connection*, *ProtocolCoDec*, *LocalSource*, and *RemoteSource objects*. A *Session object* consists of two Connection, one ProtocolCoDec, some configuration objects, and all LocalSources and RemoteSources that communicate through the ProtocolCoDec. An application can participate in multiple sessions simultaneously.

A LocalSource object coordinates the network transmission of one multimedia data stream. LocalSources conform to the StreamExporter type and export their streams to local Consumers like any other StreamExporter; however, they also packetise the media elements, use a ProtocolCoDec to encapsulate the data chunks in protocol data units of the ProtocolCoDec's particular multimedia transport protocol and to forward the resulting packet stream by means of a Connection object.

Multimedia data streams that are received from remote sources take the opposite way: packets that arrived through a Connection are processed by an appropriate ProtocolCoDec, which evaluates the protocol header that was attached to the media elements by the sender's ProtocolCoDec. The ProtocolCoDec reifies the packets' contents to the application's object representation of media elements and feeds the reified media element objects into the intra-application data flow. For that purpose, the ProtocolCoDec uses a RemoteSource object, which conforms to the StreamExporter type and represents the remote sender.

The management of an application is effected by an *AppManager*, a *GuiManager*, a *StreamManager*, a *SessionManager*, and a *SessionAgent object*.

The GuiManager handles the interaction with the user. Therefore, it provides GUI objects for configuring and monitoring architecture components, and triggers the notification of major application events to the user.

The StreamManager object administrates the objects that are involved in the local intra-application data flow, namely Stream and StreamExporter objects. All new StreamExporters are reported to the StreamManager. The StreamManager also sets up new Stream objects by registering them with a StreamExporter.

The SessionManager object is responsible for managing the objects occupied with network communication: in particular, it manages the operation of Sources, Sessions and session description exchange.

On behalf of the SessionManager, a SessionAgent imports information about sessions initiated by remote users, and exports information about locally started sessions to potential remote participants. The SessionAgent uses session description and announcement protocols for these purposes.

The AppManager serves as the central event dispatcher for the whole application: all major events, such as the reception of data of a new RemoteSource or user commands, are reported to the AppManager by appropriate method calls; the AppManager in turn invokes the corresponding methods of the StreamManager, the SessionManager, or the GuiManager to inform them of the event.

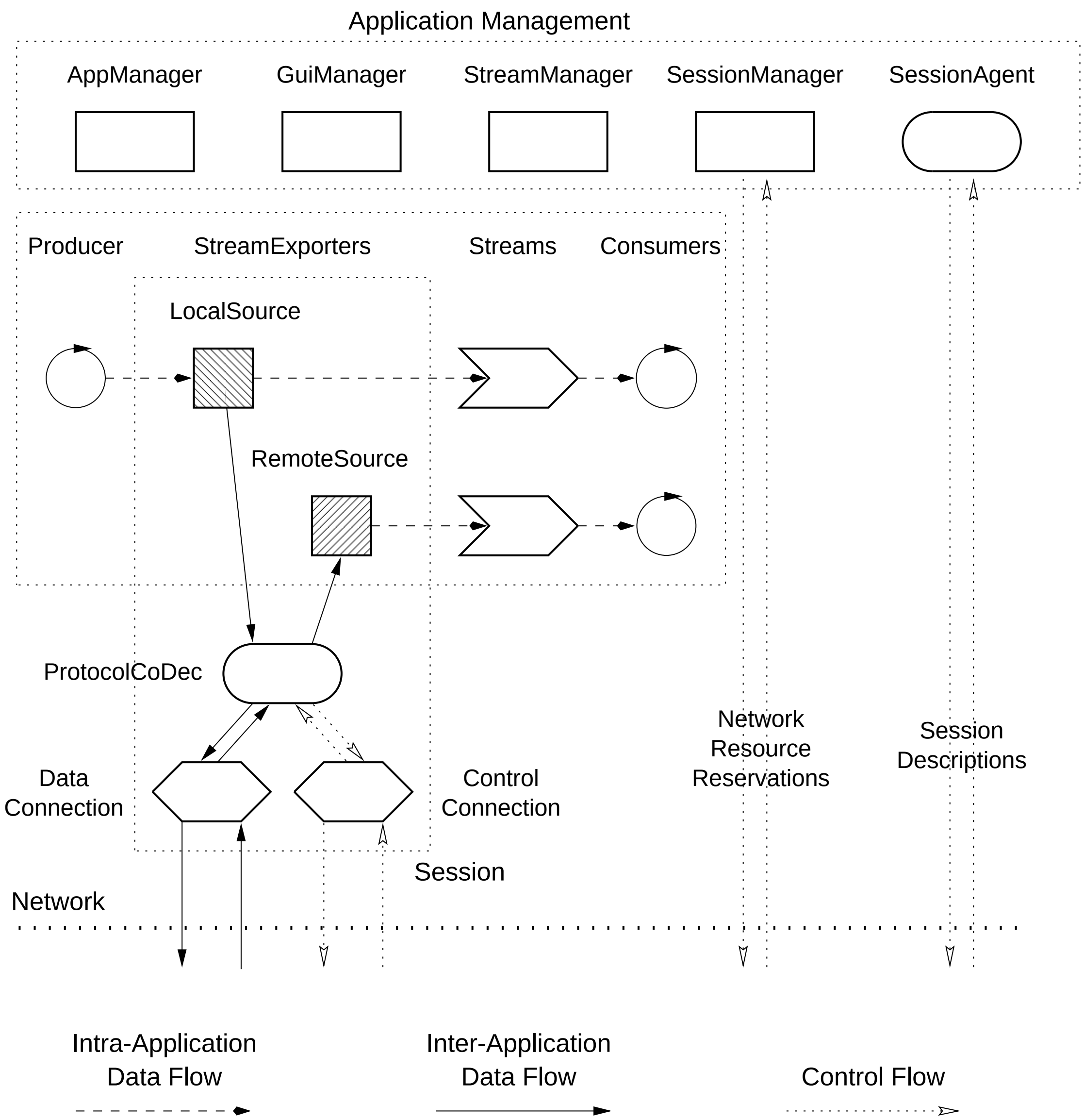

Figure 5.1: Architectural overview of the inter- and intra-application data flow.

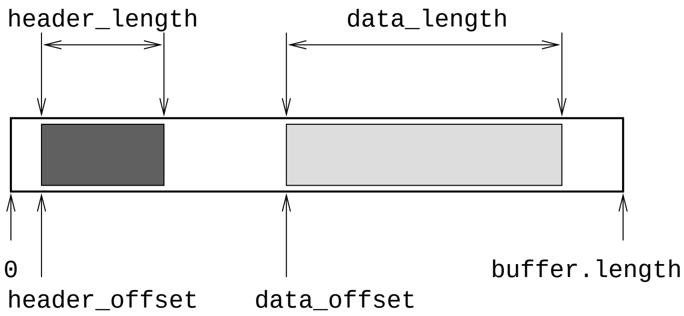

Figure 5.2: The structure of the byte array buffer contained in a chunk object.

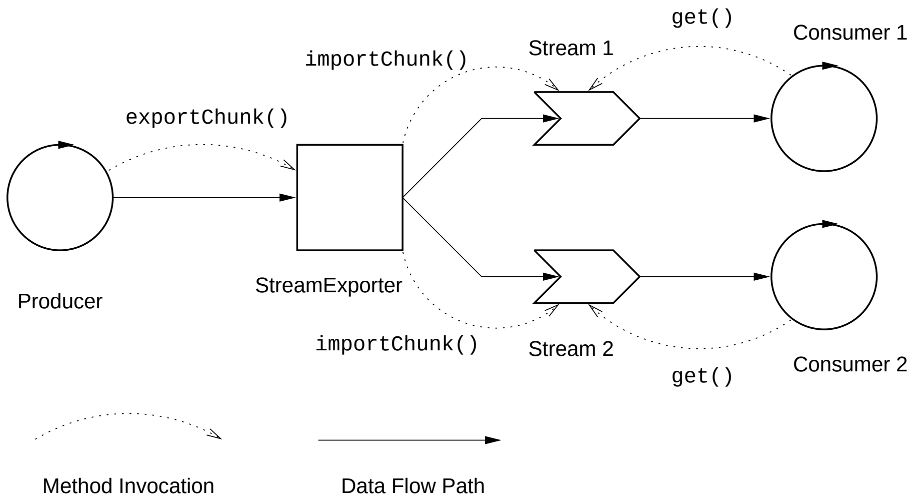

Figure 5.3: The local intra-application data flow and involved method invocations.

5.3 Local Data Flow

5.3.1 Media Elements

Media element objects, which are communicated among Producer, Consumer, StreamExporter, Stream, and the various networking objects, normally belong to the *Chunk* class. Chunk objects consist of a *byte array buffer*, which stores the media data, as well as offset and length member variables, which indicate the position and length of the header area and the payload data area of the buffer. In addition, a Chunk carries a timestamp field, which denotes the instant of the creation of the Chunk. The structure of a Chunk's buffer is shown in Figure 5.2.

For the purpose of intra- and inter-stream synchronisation, *TimedObjects* are communicated by Stream objects to Consumer objects. TimedObjects carry a reference to a payload object of any type, for example a Chunk object or any other representation of a media element, but additionally offer a `sync()` method. This method blocks its caller (normally a Consumer presenting the media element) until a certain "wakeup" point of time is reached. The wakeup instant must be previously stored in the TimedObject and is computed by mapping the timestamp of the corresponding Chunk to a local playout time.

5.3.2 Intra-Application Data Flow

Figure 5.3 illustrates the method invocations that shape the local, intra-application data flow.

Producer and *Consumer objects* are active multimedia objects in the sense of Gibbs[2], that is, to each Producer or Consumer is assigned an exclusive thread of control. Besides, Producers and Consumers are required to have a `Start()`, `Stop()` and `Finish()` method.

A Producer's only task is to generate multimedia data. The distribution of this data is handled by an associated StreamExporter object: a Producer calls the `exportChunk()` method of its StreamExporter for each generated media element. In turn, the StreamExporter distributes the media elements among the various Consumers by calling the `importChunk()` method of every registered *Stream object*. A Stream object queues the imported media elements until the Consumer extracts the elements by calling a `get()` method. Thereby, the activities of the Producer and Consumers become asynchronous. To avoid memory overflow, the Stream object can implement flow control algorithms, which can, if necessary, drop Chunks or refuse new Chunks. The decision, which Chunk to drop or refuse, depends highly only the media type. For example, the deletion of MPEG I-frames renders following B- and P-frames useless, hence no I-frames should get lost.

A Consumer object that desires to import a multimedia data stream from a particular StreamExporter invokes the `importStream()` method of the StreamManager, passing the desired StreamExporter and Stream class as arguments. The StreamManager instantiates a Stream object of the requested class and registers it with the StreamExporter. The registration causes the StreamExporter to export any new media elements from that time on to the new Stream object, too. The consumer obtains the newly instantiated Stream object as a result of the `importStream()` method invocation, and can now receive new media elements by repeatedly invoking one of the `get()` methods of its Stream object.

The `get()` methods of a Stream object differ in their semantics. All the `get()` methods extract the first Chunk element of the Stream's internal media element queue if a Chunk is available, or block the caller until a new Chunks arrives. The differences lie in the type of the returned media elements and the time-related behaviour: the returned object can either be simple `Chunk` object or a media element object of a more complex class; furthermore, a `get()` method may or may not delay the caller for the purpose of playout synchronisation.

As mentioned above, TimedObjects are used to facilitate inter- and intra-stream playout synchronisation. The necessary timing information is extracted from the timestamps of the Chunks, from which the TimedObjects are constructed. To map the timestamps to instants measurable in local wall-clock time, each Stream contains a *Clock object*, whose timestamp–to–wall-clock-time ratio is controlled by the StreamExporter's *Master_Clock object*. Each Clock may have an individual playout offset delay, however. Inter-stream synchronisation is achieved by adjusting the playout delay of one Stream's Clock to the playout delay of another related Stream. This adjustment requires a mapping between the Chunks' timestamps and the points of time of the generation of the Chunks. For example, RTCP sender reports provide this correspondence by means of an RTP timestamp and an NTP timestamp, which both denominate the same point of time, as described in Section 3.2.2.

5.4 Network Communication

5.4.1 Transmission of Multimedia Data Streams

Figure 5.4 visualises the data flow inside a session and the relevant method invocations.[3]

Session objects are designed in resemblance to RTP sessions; however, they can work with other transmission protocols as well. A Session comprises two *Connection objects* (one for payload data transmission, one for control information transmission), a *ProtocolCoDec object*, which encapsulates media elements in protocol data units of a particular transmission protocol, and a *Profile object*, which is used to configure the ProtocolCoDec and the Session object. Similar to RTP, only one payload type is conveyed in one session that is stored in the Profile.

[2] See Section 4.3.1 above.

[3] For reasons of simplicity, only one LocalSource and one RemoteSource are shown in Figure 5.4. However, the number of LocalSources or RemoteSources that participate in a session is not limited conceptually.

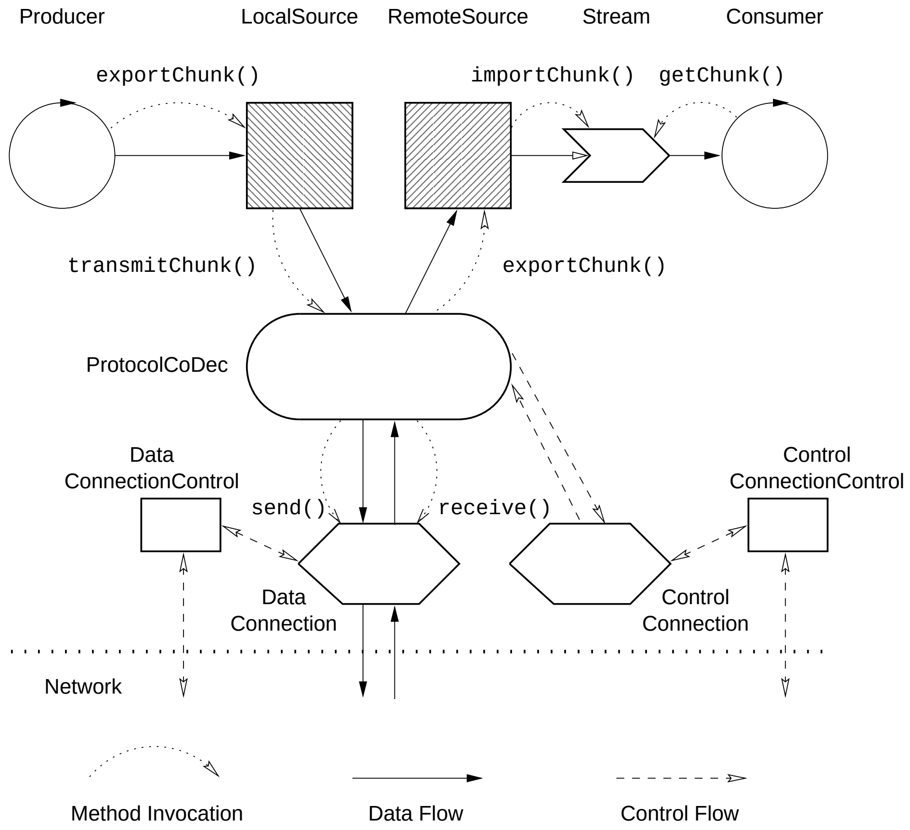

Figure 5.4: The method invocations relevant to the inter-application data flow.

The two Connection objects assigned to a Session object correspond to the two network access points of an RTP session; they are each bound to a local address and send data to a remote address. The only requirement to Connection objects is that their `send()` and `receive()` methods must support *framing*. That means, a data block which was the argument of one `send()` method invocation must always be retrieved entirely by a single invocation of the corresponding `receive()` method, or not at all in case the packet got lost during the transmission. Objects that make use of Connection objects must be prepared to deal with packet loss or a disordered packet sequence. A Connection object can be constructed around a UDP multicast socket joining a multicast host group, for example.

A Session object interacts with the underlying system's network QoS control entities, by means of a *ConnectionControl object*. The ConnectionControl object is responsible for negotiating and maintaining network resource reservations for flows that are received or forwarded by means of the Session's Connection object.
ConnectionControl objects are highly dependent on the particular operating system interfaces and networking techniques used by a Session's Connection object. A ConnectionControl implementation could for example use the RAPI mentioned in Section 3.1.5 on page 19.

A ProtocolCoDec implements a particular transport protocol suitable for the transmission of multimedia data stream. Hence, the ProtocolCoDec is the central element of a Session. Its tasks include:

- Creation and Initialisation of new LocalSource objects for every locally generated multimedia data stream that is intended to be transmitted over the network.

- Creation and Initialisation of RemoteSource objects for all remote senders whose multimedia data streams the ProtocolCoDec receives by means of its data Connection object.

- Encapsulation of media elements in protocol data units of the implemented transport protocol and forwarding of the encoded packets by means of the data Connection object.

- Reception of packets from remote sources using the data Connection object and decoding of protocol data units. Extraction of header information, such as timestamps or source identifiers.

The ProtocolCoDec is also responsible for performing all protocol-specific tasks, which can be effected automatically; depending on the protocol, this might include generation of receiver or sender statistics and forwarding them through the control Connection, as well as receiving and processing of reports from other session participants.

A Profile object is used to configure the ProtocolCoDec and Session objects. Most of all, the Profile specifies the payload type that is conveyed in a session. The Profile also lays down the maximum packet rate and the bandwidth share of a session. Besides, the profile contains protocol-specific parameters.

Generally, a Session object manages the operation of its ProtocolCoDec, Connections, LocalSources, and RemoteSources. In particular, the management comprises the following tasks:

- *Bandwidth division.* The Session object divides the session's network bandwidth among all Producers associated with the Session's LocalSource objects.

- *Session startup.* On request, the Session object performs all steps necessary to start the session. This includes starting the threads of control that generate or process multimedia data streams, as well as triggering the threads that are involved in the operation of the ProtocolCoDec.

- *Session termination and cleanup.* On termination of a session, the Session object is responsible for terminating all threads of control associated with objects operating within the context of the session. Besides, the termination code must release all resources held by the Session or associated objects. In addition, protocol- or connection-specific clean-up actions must be triggered, such as sending RTCP bye packets, or leaving multicast host groups.

SessionManagers administrate the inter-application data flow. They maintain lists of Session, ProtoSession, LocalSource, and RemoteSource objects mainly; besides, they induce the SessionAgent to export session descriptions on special occasions, such as Session creation or termination, or to notify remote session participants of changes in the application's session membership.

5.4.2 Session Description and Announcement

The demand for session announcement and session description as well as current protocol solutions were described in Section 3.3.1. Of course, the architecture presented in this Section must allow the integration of session announcement techniques.

Every application built on the proposed architecture contains a *SessionAgent object.* On behalf of the SessionManager, the SessionAgent manages the import and export of session description, as shown in Figure 5.5.

On request, the SessionAgent instantiates or terminates *SessionImporter* and *SessionExporter objects.* Such objects each have a private thread of control assigned to them. A SessionExporter uses a *SessionMapper object* to create network-transmittable session description packets of the sessions, which the application currently participates in. The SessionExporter forwards session description packets by means of a Connection object. This is done periodically or on explicit request of the SessionManager, for example at session-creation time. The format of a session description packet depends on the session description protocol that a particular SessionMapper implements.

The SessionImporter in turn receives session description packets by means of a Connection object. The received session description packets are decoded by a SessionMapper that is capable of understanding the used session description protocol. The result of the decoding operation is a

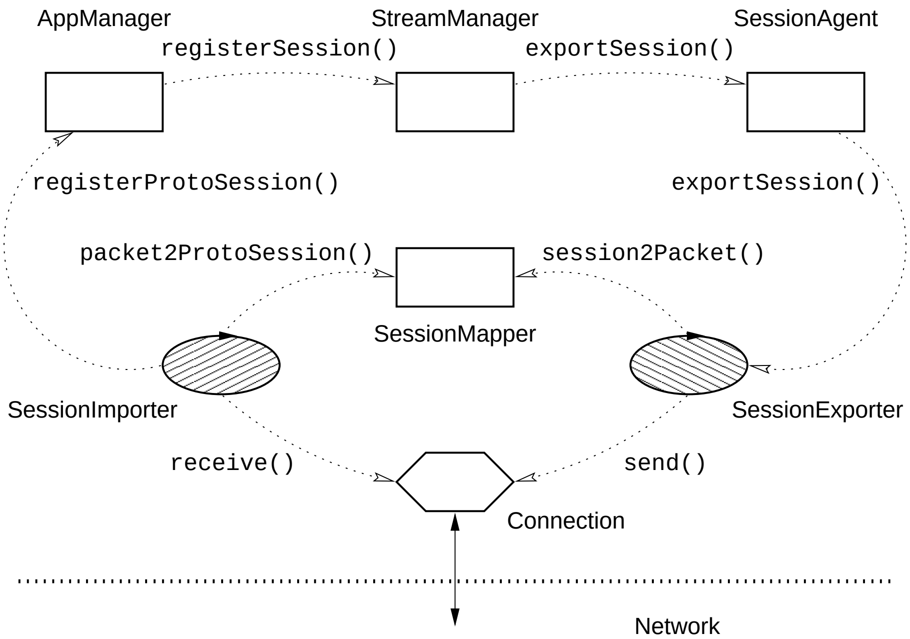

Figure 5.5: Export and import of session descriptions.

ProtoSession object, which the SessionImporter reports to the AppManager.
A ProtoSession object constructed from a session description packet contains all information that is necessary to create a Session object that is parametrised for participating in the described session. To decide whether the application is capable of joining an advertised session, the SessionMapper must know the protocols and payload types that the locally available Connection, ProtocolCoDec, Consumer, and Producer classes can handle. This information is stored in *TypeHandlerTable objects*, which are configured during the start-up phase of the application.

Though Figure 5.5 shows only one SessionExporter and one SessionImporter for the sake of simplicity, there can be an arbitrary number of these objects running within an application to communicate session descriptions with different remote partners, or to make use of different protocols. However, only one SessionExporter and one SessionImporter can share a single Connection object, whereas a SessionMapper can be accessed by all SessionExporters and SessionImporters that intend to use the SessionMapper's session description protocol.

5.5 User Interface

The GuiManager is responsible for the operation of the user interface. Its tasks fall into two categories:

- *Reporting events to the user.* The GuiManager is the central instance for reporting events to the user, such as the arrival of new session descriptions or the end of the availability of new RemoteSources. The GuiManager hides the details of a particular user interface from the rest of application, and offers user notification methods with standardised interfaces instead.
- *Administration of component-specific GUI elements.* The GuiManager maintains tables of specific configuration or presentation GUI elements. For example, if a Connection object of a certain type must be configured or if the output of a particular Consumer is to be displayed, the GuiManager is queried for the appropriate GUI element. Therefore, it is necessary that all available GUI elements must be registered with the GuiManager during the start-up phase of the application.

Chapter 6

Implementation in Java

In this chapter, I describe the `mmstream` implementation[1] of the architecture, which I presented in the previous Chapter 5. This *prototype* implementation is intended primarily as a *proof of concept* for the architectural design.
I decided to use the Java language environment for the implementation. The reasons for choosing Java are summarised in Section 6.1. The remaining sections of this chapter explain the Java implementation of major architectural components.

To distinguish objects and classes that are part of the programming language-independent elements of the design of the architecture from their counterparts of the Java implementation, I will use normal fonts to denote design components, and `typewriter` fonts for Java interfaces or classes, which implement the design components with the same names. For example, "Connection" refers to an object of the architecture design, whereas "`Connection`" denotes a corresponding Java construct.

6.1 Programming Language and Runtime Environment

The Java programming language and the Java runtime environment have been developed by Sun Microsystems, Inc. Gosling and McGilton, who were involved in the development of Java, claim in their Java white paper [GM95] that Java is a *simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, and dynamic* language. In comparison to other programming languages, they position Java in the middle between high-level and portable but slow scripting languages such as TCL on the one hand, and low level and fast but non-portable and unreliable compiled languages such as C++ on the other hand.
For the purpose of implementing an object-oriented multimedia data stream transmission architecture, the portability, architecture neutrality, object orientation, multithreading, and distribution of Java seem especially advantageous.

Java includes two possibilities of executing bytecoded Java programmes. One method is to distribute the Java bytecode via the World Wide Web by integrating the bytecoded programmes as *Java applets* in HTML pages. It is then the task of the displaying WWW browsers to start a Java bytecode interpreter for running the applet. This mode of operation is very suitable in case the presented architecture is used as a base for a multimedia chat system, for example.
However, the security managers incorporated in the Java interpreters of current WWW browsers limit the network access of Java applets in a very restrictive way: applets are only permitted to connect back to the WWW server they were downloaded from. However, distributed multimedia applications, such as for example conferencing systems, must be able to exchange data with partners on hosts residing on hosts all over the network. Consequently, the applet mechanism must be reject. Instead, I employed the other method of running Java programmes: the Java interpreter

[1] On hosts belonging to IMMD network, the Java source code can be found in the directory `/proj/i4pm/MultiMedia/javasrc/`.

executes the bytecode as a stand-alone *Java application*, which does not incur any security-related limitations. As a drawback of the application solution, the bytecode must be installed manually like any normal piece of software, and cannot be downloaded dynamically over a network by means of the applet mechanism.

Because I wanted to preserve the portability offered by Java, I refrained from writing any native C or C++ code, but relied on "pure" Java only. This decision was made with a potential applet implementation in mind, because the applet mechanism does not include the downloading of natively implemented C code, because this is extremely unfavourable from the security viewpoint. Consequently, the implementation is entirely built on the class packages provided with Sun's *Java Development Kit (JDK)*, version 1.0.2.

To achieve modularity and expandability, I made intensive use of Java's `interface` construct. Interfaces define a *type hierarchy*. Classes that declare that they conform to a particular type must provide an *implementation* of the methods defined by the type's `interface` definition. For similar reasons, I structured the classes of the architecture using Java's `package` concept. The top-level package name is `mmstream`; the subpackages are:

- `mmstream.util`
- `mmstream.address`
- `mmstream.connection` and `mmstream.connection.ip`
- `mmstream.protocols` and `mmstream.protocols.rtp`
- `mmstream.source`
- `mmstream.config` and `mmstream.session`
- `mmstream.stream`
- `mmstream.consumer` and `mmstream.producer`
- `mmstream.apps`

6.2 Intra-Application Data Flow Components

The Java Development Kit release 1.0.2 does not include any continuous-media data types. For the purpose of testing the architecture, I simulated a video frame stream by interpreting `byte` arrays as video frames encoded as 8-bit or 24-bit RGB images. Such images can be displayed using the `java.awt.image.MemoryImageSource` class.

6.2.1 StreamExporters

The `StreamExporter` interface requires of its implementations to contain among others the following methods:

- `registerStream()`. This method adds a new entry to the `StreamExporter`'s list of `Stream` objects, to which the `StreamExporter` exports `Chunk` objects.
- `exportChunk()`. An invocation of this method triggers the data export mechanism. The `StreamExporter` is intended at least to invoke the `importChunk()` method of all its registered `Stream` objects. Further actions, such as network transmission, depend on the particular implementation class.
- `createSlaveClock()`. `Stream` implementations call this method to obtain a `Clock` object. It is initialised with the timestamp–to–wall-clock-time ratio that the `StreamExporter`'s `Master_Clock` uses. Each `Stream` object uses a `Clock` object to calculate playout synchronisation delays.

6.2.2 Streams

`Stream` Interface

The `Stream` interface comprises the following methods:

- `importChunk()`. The `StreamExporter` object associated with the origin of a stream invokes the `importChunk()` method to feed a new `Chunk` object into the `Stream`'s queue of media element objects. The `StreamExporter` can determine by means of a `Stream` object's `getCopy()` method if the `Stream` object requires exclusive copies of the imported `Chunk` objects or if a shared reference is sufficient.
- `get()`. The `Stream` interface defines four different `get()` methods:
  - `getChunk()`. The simple `getChunk()` method extracts the first `Chunk` object from the `Stream`'s media element queue without any conversions or playout synchronisation delays.
  - `getData()`. The `getData()` method returns without any synchronisation delay, too. However, the retrieved object is not a mere `Chunk` object. Instead, the method constructs objects of a more complex type from the basic `Chunk` objects by decoding and interpreting the `Chunks`' `byte` array object. The processing steps and the type of the returned object depend on the class of the `Stream` object. The returned object is intended to be more suitable for the attached `Consumer` object than the basic `Chunk`, from which the complex object was built; for example, the `getData()` method could return objects of the type used by a player device object.
  - `getTimedChunk()`. This method returns `TimedObject` which contains a reference to the `Chunk` element extracted from the `Stream`'s queue. The invocation of the `TimedObject`'s `sync()` method blocks the caller until the point of time is reached that corresponds to the local playout time mapping of the `Chunk`'s timestamp.
  - `getTimedData()`. The `getTimedData()` method behaves like a combination of the `getData()` and `getTimedChunk()` methods. Actually, it returns a `TimedObject` initialised with the timing information computed from the `Chunk` that was extracted from the `Stream's` queue. The included object reference however denominates a complex object, which was constructed from the extracted `Chunk` in the same way as the `getData()` method does.
- `setMasterStream()`. An invocation of this method synchronises the playout delay of the invoked `Stream` object to the playout delay of the `Stream` object given as argument to the method.

`Stream` Implementations

The current `mmstream.stream` package includes two implementations of the `Stream` interface. First, the simple `Chunk_Stream` class, whose `getData()` and `getTimedData()` methods behave exactly like the `getChunk()` and `getChunk()` methods.
Next, the `MemoryImageSource_Stream` class' `getData()` method returns objects belonging to the `java.awt.image.MemoryImageSource` class; similarly, the `getTimedData()` method returns `TimedObjects`, which reference `MemoryImageSource` objects.

Both `Stream` implementations contain a `Clock` object, which is generated by a `Stream` objects' `StreamExporter`. The `Clock` maps the timestamp contained in `Chunk` objects to local playout time instances; the playout offset necessary for this computation is set on behalf of user interactions or by the `Stream`'s `setMasterStream()` method.

6.2.3 Producers

`ProducerControl` Interface

Producers must comply to the `mmstream.producer.ProducerControl` interface. The interface is an extension of the `java.lang.Runnable` interface. Hence, a `java.lang.Thread` object can be generated from implementations of the `ProducerControl` interface. Because Producers are multimedia objects, the `ProducerControl` interface comprises a `Start()`, a `Stop()`, and a `Finish()` method.
To facilitate control over the quantity of the Producer's data generation, the `ConsumerControl` interface contains a pair of accessor functions for the maximum allowed bandwidth, the maximum length of the produced `Chunk byte` array, and the maximum allowed `Chunk` rate.

`ProducerControl` Implementations

At the moment, the `mmstream.producer` package includes two `ProducerControl` implementations. The simple `Byte_Producer` class generates `Chunk` objects whose `byte` array elements all contain a number that is incremented for each produced `Chunk` object. The generated `Chunk` objects can be interpreted as RGB images in order to simulate a video stream.
The `File_Producer` forwards `Chunk` objects previously stored in a `SessionFile` object by a `File_Consumer` object.

6.2.4 Consumers

`ConsumerControl` Interface

Like the `ProducerControl` interface, the `mmstream.consumer.ConsumerControl` interface extends Java's `Runnable` interface and offers a `Start()`, a `Stop`, and a `Finish()` method. In addition, intra-stream playout synchronisation can be enabled or disabled with the `setPlayOutSync()` method.

`ConsumerControl` Implementations

Instances of the very simple `mmstream.consumer.Byte_Consumer` class use `Chunk_Stream` objects to obtain new `Chunk` objects, and simply output the number carried in the elements of the `Chunks`' `byte` arrays.
The `File_Consumer` class is the counterpart to the `mmstream.producer.File_Producer` class. A `File_Consumer` object also employs a `Chunk_Stream` object and stores the `Chunk` objects by means of a `SessionFile` object.
An `Image_Consumer` object relies on the service of a `MemoryImageSource_Stream`, which constructs `java.awt.image.MemoryImageSource` objects from the imported `Chunk` objects and delivers them on invocation of the `getData()` or the `getTimedData()` methods to the invoking `Image_Consumer` for displaying.

6.3 Inter-Application Data Flow Components

6.3.1 Connections

`Connection` Interface

The `mmstream.connection.Connection` interface is used to hide network type-specific details from the rest of the application. Its most important methods are the overloaded `send()` and `receive()` methods. The destination address, to which a `Connection` object sends the data, is set or queried with a pair of accessor functions, `getRemoteAddress()` and `setRemoteAddress()`. ConnectionControl objects and resource reservation mechanisms are *not* part of the architecture's implementation up to now.

UDP Based `Connection` Implementations

The `mmstream.connection.ip.UDP_Connection` class is built around the `DatagramSocket` class provided by the JDK `java.net` package. The package implements the `DatagramSocket` class by native C methods, which in turn use UDP sockets provided by the operating system. The `send()` method of the `Connection` interface is implemented by constructing a `java.net.DatagramPacket` object from the arguments to the `send()` method and sending this packet by means of the socket. The `receive()` method is implemented by returning the data, which is received as `DatagramPacket` from the socket, in an appropriate form, for example as a `Chunk` object.

The `UDP_Multicast_Connection` class is derived from the `UDP_Connection` class, but uses a `sun.net.MulticastSocket` object instead of a `DatagramSocket` object. The `MulticastSocket` object joins the IP multicast host group specified by the destination address, which must be an IP multicast group address. Furthermore, each `UDP_Multicast_Connection` object contains a TTL field, which determines the transmission range of the transmitted `DatagramPacket` objects.

6.3.2 Sources

The `mmstream.source` package includes the abstract `Source` class and its `LocalSource`, and `RemoteSource` abstract subclasses. They all comply to the `StreamExporter` interface, and therefore have a `Master_Clock` object as well as `registerStream()` and `exportChunk()` methods. The `Source` class offers implementations of those methods that take all necessary measures for registering a `Stream` object and for exporting `Chunk` objects to the known `Stream` objects.
A `RemoteSource` object represents a data sending entity located on a remote host, whose multimedia data stream the `RemoteSource` object feeds into the local data flow. `LocalSource` objects are responsible for the network transmission of a locally generated multimedia data stream.

In addition, the `Source` class serves as a deposit for user information, such as cnames or phone numbers. Moreover, `Source` objects maintain transmission statistics; therefore,the `Source` class itself defines the abstract `getSenderStats()` method, which returns a `SenderStatistics` object, containing the number of sent chunks and octets as well as the current transmission bandwidth and chunk rate. The `RemoteSource` class adds the `getReceptionStats()` method. The `ReceptionStatistics` object returned by this method describes the reception of data from the sender represented by the `RemoteSource` object. The `ReceptionStatistics` object contains a jitter value, the number of received chunks and octets, an estimate of the absolute number and fraction of packets lost, a jitter calculation, and the current bandwidth and chunk rate of the received traffic.

Instantiatable subclasses of the `LocalSource` and `RemoteSource` classes are provided by additional protocol-specific packages, such as the `mmstream.protocols.rtp` package.

6.3.3 Profiles

Objects belonging to a protocol-specific subclass of the `mmstream.config.Profile` abstract class are used to parametrise `ProtocolCoDec` and `Session` objects. The `Profile` base class specifies the `Session`'s conveyed multimedia data type, the `Session`'s bandwidth, the maximum allowed data packet length and chunk rate, and the timestamps-to-seconds rate that must be used for the generation or interpretation of timestamps.

6.3.4 CoDecs

`ProtocolCoDec` Interface

The `mmstream.protocols.ProtocolCoDec` interface defines the access to the coding and transmission services of a ProtocolCoDec object:

- `startReceiving()` and `startSending()`. These methods start up the threads that send and receive data.

- `transmitChunk()`. The `transmitChunk()` method takes a `Chunk` object as argument. It encodes the `Chunk`'s payload data and timestamp in protocol data units of the protocol that a particular `ProtocolCoDec` object implements.
- `getRemoteSouce()`. The `getRemoteSouce()` method is intended to block its caller until the `ProtocolCoDec` receives a new multimedia data stream by means of the `ProtocolCoDec`'s data `Connection` object. On reception of a new stream, the method creates and returns a new `RemoteSource` object, which represents the remote sender.
- `createLocalSource()`. On invocation of this method, the `ProtocolCoDec` returns a new `LocalSource` object, which will use the transmission service of the creating *ProtocolCoDec*.
- Accessor functions for the data and control `Connection` objects.

`RTP_CoDec` Class

The `mmstream.protocols.rtp.RTP_CoDec` class conforms to the `ProtocolCoDec` interface and implements the Real-Time Transport Protocol (RTP) described in Section 3.2.2.
The `mmstream.protocols.rtp` package contains additional classes that support the `RTP_CoDec` class:

- `RTProtocol` and `RTCProtocol` classes. These classes contain protocol constants as well as `static` methods that generate and check RTP and RTCP packet headers.
- `RTP_LocalSource` and `RTP_RemoteSource` classes. These classes are instantiatable subclasses of the abstract `RemoteSource` and `LocalSource` classes described in Section 6.3.2. Instance of these classes act as `StreamExporter` objects in the local data flow.

  `RTP_RemoteSource` objects represent remote RTP data senders, and are used to export the contents of the RTP packets received by the associated `RTP_CoDec` to local `Stream` objects. `RTP_LocalSource` objects on the other hand do not only export their chunks to local `Stream` objects, but also trigger the RTP encoding and subsequent network transmission of `Chunk` objects by calling the `transmitChunk()` method of their associated `RTP_CoDec` object.

  In addition, both RTP source classes produce general and RTP-specific transmission statistics objects, which summarise information about the sent packets of `RTP_LocalSource` objects or about the received packets of `RTP_RemoteSource` objects, respectively. The calculations and estimations are implemented according to the algorithms presented in [RFC1889, S. 6.3.1, A.3].
  In particular, an `RTP_ReceptionStatistics` object contains the number of received chunks and octets, the chunk rate and bandwidth of the received packet stream, the highest received extended sequence, an estimation of the fraction and absolute number of lost packets, a calculation of the reception jitter, and time information about the latest received RTCP sender report from the remote source.
  An `RTP_SenderStatistics` object conveys the number of sent chunks and octets, the sending bandwidth and chunk rate, the latest sent extended sequence number, and time information about the most recently sent RTCP sender report.
- `RTP_SeqNum_Registration` class. `RTP_RemoteSource` objects check the sequence number of incoming RTP packets with the help of `RTP_SeqNum_Registration` objects . The check guarantees that the sequence numbers are monotonously increasing. In addition, the check declares remote sources as invalid if the packet loss or the misordering exceeds certain limits. The implementation follows the algorithm described in [RFC1889, A.1].
- `RTP_SourceTable` class. `RTP_SourceTable` objects, one per `RTP_CoDec` object, create and keep track of the `RTP_RemoteSource` and `LocalSource` objects that use the codec. The information stored in the table is used to identify the sender of a packet by means of the SSRC identifier carried in each RTP and RTCP packet.

An instance of each of the following three thread classes runs on behalf of each `RTP_CoDec` object:

- `RTP_Receiver_Thread` class. The `RTP_Receiver_Thread` thread repeats the following steps in an endless loop:
  1. Reception of a new RTP packet by calling the `receive()` method of the codec's data `Connection` object.
  2. Check of the header of the received packet by invoking the `RTProtocol.checkHeader()` method.
  3. Selection of the `RTP_RemoteSouce` object that corresponds to the SSRC identifier carried in the packet by means of the `RTP_CoDec`'s `RTP_SourceTable`.
  4. Handing over of a `Chunk` object to the selected `RTP_RemoteSource` object for the sequence number check and a potential feed into the local data flow.
- `RTCP_Receiver_Thread` class. The `RTCP_Receiver_Thread` thread receives RTCP packets through the codec's control `Connection` object. A header check is performed and the sender is determined. The packet is then passed to the appropriate source object for further RTCP packet type specific processing.
- `RTCP_Sender_Thread` class. The `RTCP_Sender_Thread` thread generates RTCP SDES, SR, and RR packets, and forwards them by means of the codec's control `Connection` object. The thread computes its RTCP transmission interval according to the algorithm given in [RFC1889, A.7].

6.3.5 Sessions

The `mmstream.session.Session` class manages the associated `ProtocolCoDec` object, the data and control `Connection` objects, and all `LocalSource` and `RemoteSource` objects that operate within the session. The `Session` class contains a `startSending()`, a `startReception()` and a `finish()` method for starting and terminating the activities of a `Session` object.

A `Session` object controls the behaviour of all Producers associated with the `Session`'s `LocalSource` objects. Therefore, the `Session` class offers appropriate accessor functions that enforce a maximum chunk rate and chunk data length on all the registered Producers by means of the corresponding methods contained in the `ProducerControl` interface.
Similarly, the aggregate bandwidth of all Producers operating within a Session can be controlled with the `setMaxBandwidth()` method, which divides the assigned bandwidth among the registered Producers.

6.4 Session Announcement and Description Objects

6.4.1 SessionAgents

An application built on the presented architecture can make use of an object that implements the `mmstream.sessionSessionAgent` interface to manage session description import and export. The interface prescribes an `addSessionImport()` and an `addSessionExport()` method, which both pass a `Connection` and a `SessionMapper` object as arguments. The methods are intended to start threads that perpetually receive and forward session descriptions by means of the `Connection` object. The importing and exporting threads must use specified the `SessionMapper` object to decode and encode the session descriptions from and into a session description protocol representation. The interface also defines a `exportSession()` methods, which is used to transmit a single description of a particular session and its state, for example at the time of the session's creation or termination.

The `mmstream.session.TOMS_SessionAgent` class, conforming to the `SessionAgent` interface, is provided with the `mmstream.session` package. Its implementation is straightforward. It starts a `SessionExporter` or `SessionImporter` thread whenever a `addSessionImport()` or `addSessionExport()` method is called, respectively.

6.4.2 SessionMappers

The `mmstream.session.SessionMapper` interface is simple: it comprises the `session2Packet` method, which derives a transmittable description of a Session, and the `packet2ProtoSession` method, which uses a received session description packet to initialise a `ProtoSession` object. `ProtoSession` objects contain all the information that is necessary to parametrise a `Session` object for participation in the described multimedia session. The information comprises for example the type of the communicated payload, the transmission protocol used, and network addresses. In addition, they contain the cnames of the session's initiator and all known participants.

The `mmstream.session.TOMS_SessionMapper` class implements a small and simple session description protocol, which encodes the session parameters according to a type-length-value coding scheme. It was developed for testing the session description import and export mechanisms, and is tailored to the needs of the architecture. It conveys the connection type (currently either `UDP` or `UDP_MULTICAST`), the data and control addresses, the maximum transfer unit supported by the connection type, the transmission protocol (currently only `RTP`), profile-specific parameters, the payload type, the current state of the application (mainly `RUNNING` or `FINISHED`), the cname of the session's initiator, the cname of the session participant that generated the session description packet, and a session ID, the combination of which with the initiator's cname must be unique.

6.5 Managers

6.5.1 AppManagers

An AppManager object is the central event dispatching instance of the architecture. Therefore, the `mmstream.apps.AppManager` interface lays down several methods for registering the creation or the deletion of architecture components as well as a `finish()` method. `AppManager` objects also offer tables which map implementation classes to the network techniques, protocols, session description, formats or payload types they can handle, and vice versa. Besides, objects can obtain a reference to the other management objects of the architecture by calling appropriate `get()` methods of the `AppManager` object.

The `TOMS_AppManager` class implements the registration and deletion methods by reporting the events to the associated `StreamManager` and `SessionManager` objects, which takes appropriate management measures, and to the `GuiManager` object, which is responsible for notifying the user of the events.

6.5.2 GuiManagers

Implementations of the `mmstream.gui.GuiManager` interface are responsible for displaying the state and behaviour of architecture components. Therefore, the interface contains methods to inform a `GuiManager` object of the creation or termination of architecture components, for example a `registerProducer()` or a `deleteSession()` method. On invocation of such methods, the `GuiManager` must take measures to create, update, or delete appropriate elements of the application's user interface. The notification methods are normally invoked by the `AppManager` object.

The other task of a `GuiManager` object is to serve as a repository of architecture component-specific GUI components. The `GuiManager` interface comprises various methods for registering configuration or display GUI elements. For example, there are the `ConnectionOptions`, the `ProtocolCoDecOptions`, the `ConsumerOptions`, and the `ProducerOptions` interfaces. The GUI elements are stored in `java.util.Hashtable` objects, together with a key name describing the "type of things" that the corresponding architecture component can handle. The following example invocation registers the `mmstream.gui.UDP_ConnectionOptions` class, which conforms to the `mmstream.gui.ConnectionOptions` interface and which can be used to configure and instantiate `Connection` objects that are capable of operating UDP.

```
guiMan.registerConnectionOptionsGui(
        mmstream.config.Connection_TypeConfiguration.UDP_Name,
        Class.forName("mmstream.gui.UDP_ConnectionOptions")
       );
```

The `mmstream.gui` package offers a sample GUI implementation built on the classes provided by `java.awt` and `java.awt.image` packages. In particular, it contains the `TOMS_GuiManager` implementation, the `MainFrame` class, which is the root window of the grpahical user interface, the `ExportImport_Dialog`, which is used to add or delete session description import or export channels, and a collection of minor support classes. In addition, there are the `NewSession_Dialog`, `NewProducer_Dialog`, and `NewConsumer_Dialog` classes as well as component specific implementations of the various `Options` interfaces, which create parametrised instances of the respective components; furthermore, the package comprises the `Session_Dialog`, `ProtoSession_Dialog`, `LocalSource_Dialog`, `RemoteSource_Dialog`, `Producer_Dialog`, and `Consumer_Dialog` classes for displaying the state of the respective components.

6.5.3 StreamManagers

The purpose of a `StreamManager` object is to maintain lists of an application's `StreamExporter` and `Stream` objects. The necessary methods are defined in the `mmstream.stream.StreamManager` interface. Besides, the interface comprises an overloaded `importStream()` method, which is intended to create new `Stream` objects and register them with the `StreamExporter` object specified as an argument to the method.

The `mmstream.stream` package provides `TOMS_StreamManager` class, which implements the `StreamManager` interface in a straightforward way.

6.5.4 SessionManagers

`SessionManager` objects keep track of the `LocalSource`, `RemoteSource`, `Session`, `ProtoSession`, and `SessionAgent` objects of an application. Therefore, the `mmstream.session.SessionManager` interface lays down methods to register or delete such objects from a `SessionManager` object's repository. Like the `StreamManager` interface, the `SessionManager` interface contains also an `importStream()` method; however, the parameters and semantics differ: the invoking object does not specify an argument `StreamExporter` object, but instead a `Session` object. The `SessionManager` object is intended to use the next otherwise unused `LocalSource` object available from the argument `Session` object.

The `TOMS_SessionManager` implementation is also simple. The `importStream()` method is implemented by determining the `ProtocolCoDec` object belonging to the specified `Session` object and invoking the potentially blocking `getRemoteSource()` method of the codec. The returned `RemoteSource` is used for an invocation of the `StreamManager` object's `importStream()` method.

Chapter 7

Conclusions

7.1 Summary

In Chapters 1 and 2, I introduced multimedia data streams and examined their characteristics. In particular, I showed in Section 2.3 that, due to the inherent *real-time requirements* of multimedia data, *QoS control mechanisms, specialised multimedia transmission protocols,* and *multi-point communication structures* are essential for the transmission of multimedia data streams.

Next, I reviewed techniques that have been developed for the transmission of time-dependent data in Chapter 3. I deliberately concentrated on solutions provided by the Internet Engineering Task Force. Particularly, I presented the *Internet Integrated Services Architecture, the Resource ReSerVation Protocol, Internet multicasting protocols, the Real-Time Transport Protocol,* and *the meta-conference–management protocols* developed by the Multiparty Multimedia Session Control (MMUSIC) working group of the IETF.

Chapter 4 linked the transmission of multimedia data streams with the object-oriented design of multimedia applications. I focused on design approaches based on the *data-stream* and *object-stream models*, because they are especially suitable for the *integration of network communication.*

The second part of this thesis, consisting of Chapters 5 and 6, saw the presentation of the design and implementation of an *object-oriented architecture for the transmission of multimedia data streams* that I developed based on the basis of the ideas of the preceding chapters. The architecture provides a communication framework for distributed, interactive multimedia applications. The `mmstream` prototype implementation, programmed using Sun's *Java Development Kit*, serves as a proof of concept for the architectural design. The implementation currently incorporates the Real-Time Transport Protocol, a self-defined session description protocol, a sample graphical user interface, and data-producing and data-consuming elements emulating real multimedia-data-stream sources and sinks for testing the transmission components. If the necessity should arise, the implementation can be easily adapted to accommodate other protocols.

7.2 Performance Evaluation of the `mmstream` Implementation

Performance tests were carried out on Sun UltraSPARC workstations, using the Java interpreter contained in Sun's Java Development Kit 1.0.2. The test application used `Byte_Producer` objects to generate media chunks, `RTP_CoDec` objects for encoding and decoding the chunks, and `UDP_Multicast_Connection` objects for forwarding and reception packets. The maximum numbers of produced or consumed chunks per second achieved with this configuration are shown in Table 7.1 for different sizes of the `Chunk` objects' `byte` array.

The observed chunk rates are lower than required for media such as full-motion video. This behaviour has two reasons:

Chunk Size	Byte_Producer	Byte_Consumer	Image_Consumer
40	9.2	8.9	5.5
20	15.5	15.5	11.7
10	24.1	23.8	16.6
5	32.1	29.2	21.2
2	41.5	29.8	26.4

Table 7.1: Performance of the `mmstream` implementation. The `Chunk` size is given in kB, whereas the respective performance for the Producer and the Consumers is measured in `Chunk`s per second.

- *Interpretation.* It must be kept in mind that the above chunk rates were achieved using *interpreted* Java bytecode. However, interpreting bytecode is slower by orders of magnitude than executing native machine code. The situation will improve with the advance of *just-in-time* compilers, which were not included in the JDK 1.0.2. Just-in-time compiler will be integrated into Java interpreters to generate native machine code from bytecode on the fly. Sun [GM95, Sec. 8.1] claims, that the performance of machine code generated from Java bytecode will turn out roughly equivalent to the performance of machine code compiled from C++ source code in traditional ways.
- *Implementation style.* The current implementation puts the main emphasis on gathering and displaying performance and network state data. For example, on reception of each RTP packet the complete GUI component for the corresponding `RTP_RemoteSource` object is updated with newly computed reception statistics information. This strategy entails a considerable performance penalty.

7.3 Future Work

The current implementation excludes some topics, which are subject to further work:

Resource reservation mechanisms. An ConnectionControl object implementation based on the RSVP API described Section 3.1.5 appears promising and feasible. However, RSVP implementations themselves are available only for a very limited set of machine and operating-system architectures, currently including mainly Sun workstations under Solaris and SGI workstations under IRIX. In addition, calls of C-language RAPI functions from Java introduces into the implementation native C code, which restricts the architecture independence and portability.

Transmission-rate adjustment. Since resource reservation and QoS control mechanisms have not been widely deployed yet, dynamic adjustment of the transmission rates is necessary to avoid packet loss due to network congestion. The algorithms I reviewed in Section 3.2.2 on page 27 appear suitable, especially the light-weight approach proposed by Busse, Deffner, and Schulzrinne [BDS96].

Session description, announcement, and invitation protocols. Distance conferencing and lecturing applications require the integration of session and conference management protocols. The set of such protocols developed recently by the IETF Multiparty Multimedia Session Control working group, whose activities I described in Section 3.3.1, seems suitable for this purpose; the `mmstream` class collection could be expanded by a `SDP_SessionMapper` class implementing the Session Description Protocol, and by extensions of the `SessionExporter` and `SessionImporter` classes implementing the Session Announcement Protocol and the Session Initiation Protocol.

Media-on-demand mechanisms. Media-on-demand client applications require the incorporation of means by which the transmission of multimedia data streams from remote media-on-demand servers can be requested. Again, the MMUSIC group offers a protocol that easily fits into the `mmstream` architecture: the set of threads that a `SessionAgent` object administrates can be extended by `SessionRequester` threads, which use the Real-Time Streaming Protocol to request and control the on-demand delivery of multimedia data streams from a remote media server.

Appendix A

RTP and RTCP

A.1 RTP Message Format

An RTP packet as defined in [RFC1889] consists of a common header, a list of contributing source identifiers, a potential profile-specific header extension, the actual payload, and some padding octets if required for encryption or by the underlying protocol.

A.1.1 RTP Common Header

The common header begins with a fixed-length block (`V` to `SSRC`), followed by a variable-length CSRC list.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|V=2|P|X|  CC   |M|     PT      |       sequence number SN      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           timestamp TS                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           synchronisation source (SSRC) identifier            |
+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
|            contributing source (CSRC) identifiers             |
|                             ....                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

`V`: 2 bits
: This field identifies the version of RTP, currently 2.

`P`: 1 bit
: The padding bit indicates whether the packet carries one or more padding octets at the end which are not part of the payload. The number of padding octets to be ignored is contained in the last padding octet.

`X`: 1 bit
: The extension bit indicates whether the fixed header is followed by a (single) header extension.

`CC`: 4 bits
: The CSRC count contains the number of CSRC identifiers that follow the SSRC identifier.

`M`: (normally) 1 bit
: The interpretation of the marker is profile-dependent. It can be used to mark significant

events such as frame boundaries. A profile can trade additional marker bits for additional `PT` bits.

`PT`: (normally) 7 bits
The payload type field identifies the format of the RTP payload, as specified in a profile. Byte codes for an initial set of payload types are defined in [RFC1890] and [RFC1700]. An RTP sender emits only a single RTP payload type at any given time; this field is not intended for multiplexing separate media streams.

`SN`: 16 bits
A sender increments the value of this sequence number field by one for each RTP data packet sent, starting with a random initial value. A receiver might use this field to detect packet loss, and to restore the original sequence of packets arrived out of order.

`TS`: 32 bits
The timestamp reflects the sampling instant of the first octet in the RTP data packet, derived from a clock that increments monotonically and linearly in time. The timestamp allows the receiver to do playout synchronisation and jitter calculations.

`SSRC`: 32 bits
The `SSRC` field identifies the synchronisation source. Although the SSRC identifiers are chosen randomly, it is not permissible that an RTP session participant chooses an SSRC identifier used by another session participant. Every RTP implementation must be programmed to detect and resolve any SSRC collision.

`CSRC`: 0 to 15 items, 32 bits each
CSRC identifiers are inserted by mixers, using the SSRC identifiers of sources that have contributed to the payload in the packet. The number of identifiers is given by the `CC` field.

A.1.2 RTP Header Extension

The existence of a header extension is indicated by the `X` bit in the fixed header. Header extension should only be used for experimental purposes.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      defined by profile       |           length              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
.                        header extension                       .
.                             ....                              .
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

A.2 RTCP Message Formats

Each RTCP packet carries in its header one of the following packet type codes:

- `200 = SR` Sender Report packet
- `201 = RR` Receiver Report packet
- `202 = SDES` Source Description packet
- `203 = BYE` Goodbye packet
- `204 = APP` Application-defined packet

A.2.1 Sender Report RTCP Packets (SR)

A Sender Report message consists of the header, the sender information block, a variable number of receiver report blocks, and potentially a profile-specific extensions.

The header:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|V=2|P|    RC   |   PT=SR=200   |             length L          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         SSRC of sender                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

`V, P`:
: As described for RTP data packets, see above.

`RC`: 5 bits
: The number of reception report blocks contained in this packet.

`PT`: 8 bits
: The packet type constant 200 designates an RTCP `SR` packet.

`L`: 16 bits
: The length of this RTCP packet in 32-bit words minus one, including the header and any padding.

`SSRC`: 32 bits
: The synchronisation source identifier for the sender of this `SR` packet.

The sender information block:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|            NTP timestamp, most significant word NTS           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             NTP timestamp, least significant word             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       RTP timestamp RTS                       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   sender's packet count SPC                   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    sender's octet count SOC                   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

`NTS`: 64 bits
: The NTP timestamp[1] indicates the point of time measured in wall clock time when this report was sent. In combination with timestamps returned in reception reports from the respective receivers, it can be used to estimate the round-trip propagation time to and from the receivers.

`RTS`: 32 bits
: The RTP timestamp resembles the same time as the NTP timestamp (above), but is measured in the same units and with the same random offset as the RTP timestamps in data

[1] See [RFC1305] for the specification of Network Time Protocol (NTP).

packets. This correspondence may be used for intra- and inter-media synchronisation for sources whose NTP timestamps are synchronised, and may be used by media-independent receivers to estimate the nominal RTP clock frequency.

`SPC`: 32 bits
The sender's packet count totals up the number of RTP data packets transmitted by the sender since joining the RTP session. This field can be used to estimate the average data packet rate.

`SOC`: 32 bits
The total number of payload octets (i.e., not including the header or any padding) transmitted in RTP data packets by the sender since starting up transmission. This field can be used to estimate the average payload data rate.

The receiver report blocks:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                 SSRC_1 (SSRC of first source)                 |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|fraction lost F|      cumulative number of packets lost C      |
-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         extended highest sequence number received  EHSN       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      inter-arrival jitter J                   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           last SR LSR                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     delay since last SR DLSR                  |
+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
|                 SSRC_2 (SSRC of second source)                |
:                               ...                             :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

`SSRC_n`: 32 bits
The SSRC identifier of the sender whose reception is reported in this block.

`F`: 8 bits
The sender of the receiver report estimates the fraction of the RTP data packets from source SSRC_n that it assumes to be lost since it sent the previous `SR` or `RR` packet.

`C`: 24 bits
The sender of a receiver report blocks also tries to estimate the total number of RTP data packets from source SSRC_n that have been lost since the beginning of reception. Packets that arrive late are not counted as lost, and the loss may be negative if there are duplicates.

`EHSN`: 32 bits
The low 16 bits of the extended highest sequence number contain the highest sequence number received in an RTP data packet from source SSRC_n, and the most significant 16 bits extend that sequence number with the corresponding count of sequence number cycles.

`J`: 32 bits
An estimate of the statistical variance of the RTP data packet inter-arrival time, measured in timestamp units and expressed as an unsigned integer.

The extensions:

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  profile-specific extensions                    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

A.2.2 Receiver Report RTCP Packets (RR)

Receiver Reports are structured in the same way as Sender Reports. Of course, they include no sender information block, and the packet type code is 201.

```
 0               1               2               3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|V=2|P|    RC   |   PT=RR=201   |             length L            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     SSRC of packet sender                       |
+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
|                 SSRC_1 (SSRC of first source)                   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| fraction lost |       cumulative number of packets lost         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           extended highest sequence number received             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      inter-arrival jitter                       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         last SR (LSR)                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   delay since last SR (DLSR)                    |
+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
|                 SSRC_2 (SSRC of second source)                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:                               ...                               :
+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
|                  profile-specific extensions                    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

A.2.3 Goodbye RTCP Packets (BYE)

A participant sends a `BYE` packet to indicate that one or more sources are no longer active, optionally giving a reason for leaving.

```
 0               1               2               3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|V=2|P|    SC   |   PT=BYE=203  |             length L            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           SSRC/CSRC                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:                              ...                                :
+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
|     length    |               reason for leaving (opt)        ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

`V, P, PT, L, SSRC/CSRC:`
As described for `SR` packets, with the packet type code 202 set.

`SC`: 5 bits
The number of SSRC/CSRC identifiers contained in this `BYE` packet.

A.2.4 Source Description RTCP Packets (SDES)

A `SDES` packet consists of a `SDES` header and a variable number of chunks for the described sources. Each chunk in turn consists of a SSRC/CSRC identifier and a collection of `SDES` items. `SDES` items themselves consists of a `SDES` item type code (8 bits), a length field (8 bits) and as much text octets as the length field indicates.

SDES Header

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|V=2|P|    SC   |  PT=SDES=202  |             length L          |
+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
|                          SSRC/CSRC_1                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           SDES items                          |
|                              ...                              |
+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
|                          SSRC/CSRC_2                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           SDES items                          |
|                              ...                              |
+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
```

`V, P, PT, L`:
As described for `SR` packets, with the packet type code set to 202.

`SC`: 5 bits
The number of SSRC/CSRC chunks contained in this `SDES` packet.

The different `SDES` items are encoded according to a type-length-value scheme. Currently, `CNAME`, `NAME`, `EMAIL`, `PHONE`, `LOC`, `TOOL`, `NOTE`, and `PRIV` items are defined in [RFC1889].
The `CNAME` item is mandatory in every `SDES` packet, which in turn is mandatory part of every compound RTCP packet.
Like the SSRC identifier, a `CNAME` must differ from the `CNAME`s of every other session participants. But instead of choosing the `CNAME` identifier randomly, the `CNAME` should allow both a person or a program to locate the source by means of the `CNAME` contents.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    CNAME=1    |     length    | user and domain name        ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

A.2.5 Application-Defined RTCP Packets (APP)

The `APP` packet is intended for experimental use at developing new applications and features. Once a new `APP` RTCP packet type is test and found useful, it should be registered with the Internet Assigned Numbers Authority (IANA) as an original packet type.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|V=2|P|   ST    |   PT=APP=204  |             length L          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                              SSRC                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         name (ASCII) N                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   application-dependent data A              ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

`V, P, PT, L:`
: As described for `SR` packets, with the packet type code set to 204 for verb—APP— packets.

`ST:` 5 bits
: The subtype field allows a set of `APP` packets to be defined under one unique name, or provides any application-dependent data.

`N:` 4 octets
: A name chosen by the person defining the set of `APP` packets to be unique with respect to other `APP` packets this application might receive.

`A:` variable length
: Application-dependent data may or may not appear in an `APP` packet.

Appendix B

RSVP

B.1 RSVP Object Formats and Construction Rules

Each RSVP object is constructed from the following fundamental elements in accordance with the specifications in [BZBHJ96].

B.1.1 Per-Object Header

Every object starts with the following 32-bit message header:

```
MSB                                                         LSB
31    28 27                   16 15                           0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   V   |    Unused             |     Overall Length            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

`V:` 4 bits
  Message format version; currently 0.

`Overall Length:` 16 bits
  Message length in 32-bit words not including this header.

B.1.2 Per-Service Data Header

The above object header is followed by one or more service-specific blocks. Each block consists of a 32-bit header identifying a service and the service-specific QoS control information.

```
MSB                                                         LSB
31             24 23          16 15                           0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Svc_Number   |B| Reserved    |            Svc_Length         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

`Svc_Number:` 8 bits
  The service ID number as defined in a service specification, for example in [Wro96b] for Controlled-Load Service or in [SPG96] for guaranteed service.

`B:` 1 bit
  Break bit: set to 1 if the service is unsupported somewhere in the path, 0 if all hops support this service.

`Svc_Length:` 16 bits
  The service-specific data block length in 32-bit words, not including the header.

B.1.3 Parameter Encoding

The per-service blocks contain various parameters for configuring the service usage. Each parameter encoding consists of a parameter header shown below and the actual data representing the parameter value. Parameter values are encoded in one or more 32-bit words using the XDR external data representation.[1]

```
MSB                                                             LSB
 31            24 23            16 15                            0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Param_Num     |I   Flags       |          Param_Length          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Param_Num: 8 bits
: Parameter number as defined in service specifications.

Flags: 8 bits
: Per-parameter flags, currently only the meaning of bit 23 (I) is defined as INVALID.

Param_Length: 16 bits
: Length of the per-parameter data in 32-bit words, not including the header.

B.2 Controlled-Load and Guaranteed Services Objects

[Wro96a] defines the format of SENDER_TSPEC, FLOWSPEC, and ADSPEC objects for Guaranteed and Controlled-Load QoS control services.

B.2.1 RSVP SENDER_TSPEC Object

A SENDER_TSPEC reflects a sender's view of its generated traffic.

```
 31            24 23            16 15              8 7             0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| 0 (a) |    reserved           |             7 (b)               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    1  (c)     |0| reserved    |             6 (d)               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   127 (e)     |    0 (f)      |             5 (g)               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Token Bucket Rate [r] (32-bit IEEE floating point number)    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Token Bucket Size [b] (32-bit IEEE floating point number)    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Peak Data Rate [p] (32-bit IEEE floating point number)       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Minimum Policed Unit [m] (32-bit integer)                    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Maximum Packet Size [M]  (32-bit integer)                    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

(a) Message format version number (0).

(b) Overall length (7 words not including the header).

(c) Service header, service number 1 (default/global information).

(d) Length of data for service 1, 6 words not including the header.

[1] See [RFC1832] for a thorough description of XDR.

(e) Parameter ID, parameter 127 (Token_Bucket_TSpec).

(f) Parameter 127 flags (none set).

(g) Parameter 127 length, 5 words not including the header.

[r], [b], [p], [m], [M]: Token bucket parameters.

B.2.2 RSVP FLOWSPEC Objects

FLOWSPEC objects describe receiver's desired resource reservation parameters.

RSVP FLOWSPEC Object for Requesting Guaranteed Service

```
 31           24 23           16 15            8 7             0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| 0 (a) |    Unused             |            10 (b)             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    2  (c)     |0| reserved    |             9 (d)             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   127 (e)     |    0 (f)      |             5 (g)             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Token Bucket Rate [r] (32-bit IEEE floating point number)    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Token Bucket Size [b] (32-bit IEEE floating point number)    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Peak Data Rate [p] (32-bit IEEE floating point number)       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Minimum Policed Unit [m] (32-bit integer)                    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Maximum Packet Size [M]  (32-bit integer)                    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     130 (h)   |    0 (i)      |            2 (j)              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Rate [R]  (32-bit IEEE floating point number)                |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Slack Term [S]  (32-bit integer)                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

(a) Message format version number (0).

(b) Overall length, 9 words not including the header.

(c) Service header, service number 2 (Guaranteed).

(d) Length of per-service data, 9 words not including per-service header.

(e) Parameter ID, parameter 127 (Token Bucket TSpec).

(f) Parameter 127 flags (none set).

(g) Parameter 127 length, 5 words not including parameter header.

(h) Parameter ID, parameter 130 (Guaranteed Service RSpec).

(i) Parameter 130 flags (none set).

(j) Parameter 130 length, 2 words not including parameter header.

RSVP FLOWSPEC Object for Requesting Controlled-Load Service

```
 31           24 23           16 15            8 7             0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| 0 (a) |    reserved           |             7 (b)             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    5  (c)     |0| reserved    |             6 (d)             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   127 (e)     |    0 (f)      |             5 (g)             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Token Bucket Rate [r] (32-bit IEEE floating point number)    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Token Bucket Size [b] (32-bit IEEE floating point number)    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Peak Data Rate [p] (32-bit IEEE floating point number)       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Minimum Policed Unit [m] (32-bit integer)                    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Maximum Packet Size [M]  (32-bit integer)                    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

(a) Message format version number (0).

(b) Overall length, 7 words not including the header.

(c) Service header, service number 5 (Controlled-Load).

(d) Length of data for controlled-load service, 6 words not including per-service header.

(e) Parameter ID, parameter 127 (Token Bucket TSpec).

(f) Parameter 127 flags (none set).

(g) Parameter 127 length, 5 words not including per-service header.

B.2.3 RSVP ADSPEC Objects

RSVP ADSPEC Object Structure

An ADSPEC object is constructed from a default general parameter fragment and various service-specific data fragments.

```
31           24 23           16 15            8 7             0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| 0 (a) |      reserved         |  Msg length - 1 (b)           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Default General Parameters fragment (Service 1)  (c)       |
|    (Always Present)                                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Guaranteed Service Fragment (Service 2)    (d)             |
|    (Present if application might use Guaranteed Service)      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Controlled-Load Service Fragment (Service 5)  (e)          |
|    (Present if application might use Controlled-Load Service) |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

(a) Message format version number (0).

(b) Overall message length not including the header word.

(c, d, e) Data fragments.

Default General Characterisation Parameters Data Fragment for ADSPEC Objects

All RSVP `ADSPEC` objects must contain the general characterisation parameters defined in [SW96b].

```
31           24 23           16 15            8 7             0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    1  (c)     |x| reserved    |           8 (d)               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    4 (e)      |    (f)        |           1 (g)               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       IS hop count (32-bit unsigned integer)                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    6 (h)      |    (i)        |           1 (j)               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Path b/w estimate  (32-bit IEEE floating point number)       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     8 (k)     |    (l)        |           1 (m)               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        Minimum path latency (32-bit integer)                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     10 (n)    |      (o)      |           1 (p)               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      Composed MTU (32-bit unsigned integer)                   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

(c) Per-Service header, service number 1 (Default General Parameters).

(d) Global Break bit ([SW96b], Parameter 2) (marked `x`) and length of General Parameters data block.

(e) Parameter ID, parameter 4 (Number-of-IS-hops parameter).

(f) Parameter 4 flag byte.

(g) Parameter 4 length, 1 word not including the header.

(h) Parameter ID, parameter 6 (Path-BW parameter).

(i) Parameter 6 flag byte.

(j) Parameter 6 length, 1 word not including the header.

(k) Parameter ID, parameter 8 (minimum path latency).

(l) Parameter 8 flag byte.

(m) Parameter 8 length, 1 word not including the header.

(n) Parameter ID, parameter 10 (composed path MTU).

(o) Parameter 10 flag byte.

(p) Parameter 10 length, 1 word not including the header.

Controlled-Load Service ADSPEC Data Fragment

The Controlled-Load service does not require any extra `ADSPEC` data; the only service-specific `ADSPEC` data is the Controlled-Load break bit.

```
 31          24 23           16 15            8 7             0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     5 (a)     |x|  (b)        |            N-1 (c)            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
. Service-specific general parameter headers/values, if present .
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

(a) Per-Service header, service number 5 (Controlled-Load).

(b) Break bit.

(c) Length of per-service data in 32 bit words not including the header word.

Guaranteed Service ADSPEC Data Fragment

```
31            24 23           16 15            8 7             0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     2 (a)     |x|  reserved   |           N-1 (b)             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    133 (c)    |     0 (d)     |             1 (e)             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   End-to-end composed value for C [Ctot] (32-bit integer)     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     134 (f)   |       (g)     |             1 (h)             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   End-to-end composed value for D [Dtot] (32-bit integer)     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     135 (i)   |       (j)     |             1 (k)             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Since-last-reshaping point composed C [Csum] (32-bit integer) |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     136 (l)   |       (m)     |             1 (n)             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Since-last-reshaping point composed D [Dsum] (32-bit integer) |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
. Service-specific general parameter headers/values, if present .
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

(a) Per-Service header, service number 2 (Guaranteed).

(b) Break bit and Length of per-service data in 32-bit words not including the header word.

(c) Parameter ID, parameter 133 (Composed Ctot).

(d) Parameter 133 flag byte.

(e) Parameter 133 length, 1 word not including the header.

(f) Parameter ID, parameter 134 (Composed Dtot).

(g) Parameter 134 flag byte.

(h) Parameter 134 length, 1 word not including the header.

(i) Parameter ID, parameter 135 (Composed Csum).

(j) Parameter 135 flag byte.

(k) Parameter 135 length, 1 word not including the header.

(l) Parameter ID, parameter 136 (Composed Dsum).

(m) Parameter 136 flag byte.

(n) Parameter 136 length, 1 word not including the header.

B.3 RSVP Message Format

An RSVP message comprises a common header and a body assembled from a variable number of variable-length, typed RSVP objects from Appendix B.1 and B.2, as specified in [BZBHJ96].

B.3.1 RSVP Common Header Format

The common header considers of two 32-bit words of the following form:

```
       0             1              2             3
+-------------+-------------+-------------+-------------+
| Vers | Flags|  Msg Type   |       RSVP Checksum       |
+-------------+-------------+-------------+-------------+
|  Send_TTL   | (Reserved)  |        RSVP Length        |
+-------------+-------------+-------------+-------------+
```

`Vers:` 4 bits
The current protocol version number is currently 1.

`Flags:` 4 bits
0x01-0x08: Reserved
No flag bits are defined yet.

`Msg Type:` 8 bits
The current set of RSVP messages, comprises:

- `1 = PATH`
- `2 = RESV`
- `3 = PATHERR`
- `4 = RESVERR`
- `5 = PATHTEAR`
- `6 = RESVTEAR`
- `7 = RESVCONF`

`RSVP Checksum:` 16 bits
The one's complement of the one's complement sum of the message.

`Send_TTL:` 8 bits
The IP TTL value with which the message was sent.

`RSVP Length:` 16 bits
The sum of the lengths of the common header and all objects included in the message.

B.3.2 RSVP Object Encapsulation Format

Every RSVP object encapsulated in an RSVP message consists of a 32-bit word header and the object's contents as described above:

```
       0             1             2             3
+-------------+-------------+-------------+-------------+
|       Length (bytes)      |  Class-Num  |   C-Type    |
+-------------+-------------+-------------+-------------+
.                   (Object Contents)                   .
+-------------+-------------+-------------+-------------+
```

`Length:` 16 bits
Contains the total length of the object including the object's header, measured in bytes. Objects must be aligned to 32-bit words.

Class-Num: 8 bits
The class number is used to identify the class of an object. The following classes are currently defined:

- 1 = SESSION
- 3 = RSVP_HOP
- 4 = INTEGRITY
- 5 = TIME VALUES
- 6 = ERROR_SPEC
- 7 = SCOPE
- 8 = STYLE
- 9 = FLOWSPEC
- 10 = FILTER_SPEC
- 11 = SENDER_TEMPLATE
- 12 = SENDER_TSPEC
- 13 = ADSPEC
- 14 = POLICY_DATA
- 15 = RESV_CONFIRM

Object Contents: The body of an object corresponding to the class number.

B.3.3 Format of selected RSVP messages

For each RSVP message type, there is a set of rules for the permissible choice of object types. The rules for selected RSVP messages is given here, using the in Backus-Naur form:

PATH Messages

```
<Path Message> ::=          <Common Header> [ <INTEGRITY> ]
                                     <SESSION> <RSVP_HOP>
                                     <TIME_VALUES>
                                   [ <POLICY_DATA> ... ]
                                   [ <sender descriptor> ]

<sender descriptor> ::=     <SENDER_TEMPLATE> <SENDER_TSPEC>
                                   [ <ADSPEC> ]
```

RESV Messages

```
<Resv Message> ::=          <Common Header> [ <INTEGRITY> ]
                                     <SESSION>  <RSVP_HOP>
                                     <TIME_VALUES>
                                   [ <RESV_CONFIRM> ]  [ <SCOPE> ]
                                   [ <POLICY_DATA> ... ]
                                     <STYLE> <flow descriptor list>

<flow descriptor list> ::=  <empty> |
                            <flow descriptor list> <flow descriptor>
```

B.4 RAPI Function Prototypes

This section lists the C function prototypes of selected fundamental RAPI library functions defined in [BH96].

B.4.1 rapi_session Function

```
int rapi_session(
     struct sockaddr_in *Dest,   /* Session: (Dest addr, port) */
     int         Protid,         /* Protocol Id                */
     int         flags,          /* flags                      */
     int         (*Event_rtn)(), /* Address of callback rtn    */
     void        *Event_arg,     /* App argument to callback   */
     int         *errnop         /* Place to return error code*/
    )
```

B.4.2 rapi_sender Function

```
int rapi_sender(
     int          Sid,            /* Session ID                   */
     int          flags,          /* Flags                        */
     struct sockaddr_in *Host,    /* Local host: (Src addr, port)*/
     rapi_filter *SenderTemplate, /* Sender template              */
     rapi_tspec  *SenderTspec,    /* Sender Tspec                 */
     rapi_adspec *SenderAdspec,   /* Sender Adspec                */
     rapi_policy *SenderPolicy,   /* Sender policy data           */
     int          TTL;            /* TTL of multicast data        */
    )
```

B.4.3 rapi_reserve Function

```
int rapi_reserve(
     int           Sid,              /* Session ID  */
     int           flags,
     struct sockaddr_in
           *RHost,          /* Receive host addr (opt)*/
     int           StyleId,          /* Style ID */
     rapi_stylex_t *Style_ext,       /* Style extension */
     rapi_policy_t *Rcvr_policy,     /* Receiver policy */
     int           FilterSpecNo,     /* # of filter specs */
     rapi_filter  *FilterSpec_list, /* List of filter specs */
     int           FlowspecNo,       /* # of flowspecs */
     rapi_fspec   *Flowspec_list     /* List of flowspecs */
    )
```

B.4.4 rapi_release Function

```
int rapi_release(
     u_long Sid /* Session ID  */
    )
```

Bibliography

[BDS96] Ingo Busse, Bernd Deffner, Henning Schulzrinne.
"Dynamic QoS Control of Multimedia Applications based on RTP" in *Proceedings of the First International Workshop on High Speed Networks and Open Distributed Platforms*, St. Petersburg, Russia, June 1995.
Also[4] published in *Computer Communications*, January 1996.

[BH96] Bob Braden, Don Hoffman.
RSVP Application Programming Interface (RAPI) for SunOS/BSD: V4.0,
Internet Draft[3] `<draft-ietf-rsvp-bsdapi.txt>`.
Work in Progress, August 1996.

[Bla91] Gerold Blakowski.
"Concept of a Language for the Description of Transport and (Re-)presentation Properties of Multimedia Objects" in *Informatik Fachberichte no. 293*, p. 465–474.
Berlin, Germany: Springer, 1991.

[Boo94] Grady Booch.
Object-Oriented Analysis and Design with Applications, Second Edition.
Redwood City, CA: Benjamin/Cummings, 1994.

[BTW94] Jean-Chrysotome Bolot, Thierry Turletti, Ian Wakeman.
"Scalable Feedback Control for Multicast Video Distribution in the Internet" in *Proceedings of the SIGCOMM '94 Symposium on Communications Architectures and Protocols*, London, England, August 1994.
Computer Communications Review, Volume 24(4), p. 58–67, October 1994.

[BZBHJ96] Bob Braden, Lixia Zhang, Steve Berson, Shai Herzog, Sugih Jamin.
Resource ReSerVation Protocol (RSVP)—Version 1 Functional Specification,
Internet Draft[2] `<draft-ietf-rsvp-spec-14.txt>`.
Work in Progress, IETF RSVP Working Group, November 1996.

[CSZ92] David D. Clark, Scott Shenker, Lixia Zhang.
"Supporting Real-Time Applications in an Integrated Services Packet Network: Architecture and Mechanisms" in *Proceedings of the SIGCOMM '92 Symposium on Communications Architectures and Protocols*, Baltimore, MD, August 1992.
Computer Communications Review, Volume 22(4), October 1992.

[CT90] David D. Clark, David L. Tennenhouse.
"Architectural Considerations for a New Generation of Protocols" in *Proceedings of the SIGCOMM '90 Symposium on Communications Architectures and Protocols*, Philadelphia, PA, September 1990.
Computer Communications Review, Volume 20(4), p. 200–208, September 1990.

[GBT93] Simon Gibbs, Christian Breiteneder, Dennis Tsichritzis.
Data Modelling of Time-Based Media[6]. 1993.

[GDT92] Simon Gibbs, Laurent Dami, Dennis Tsichritzis.
"An Object-Oriented Framework for Multimedia Composition and Synchronisation" in *Proceedings of the Eurographics '92 Workshop on Multimedia*, Stockholm, Sweden, April 1991.
Berlin, Germany: Springer, 1992.

[Gib91] Simon Gibbs.
"Composite Multimedia and Active Objects" in *Proceedings of the OOPSLA '91 Conference*,

Phoenix, Arizona, October 1991.
ACM SIGPLAN Notices, Volume 26(11), p. 97-112, Nov 1991.

[GM95] James Gosling, Henry McGilton.
"The Java Language Environment: A White Paper"[5].
Sun Microsystems, Inc., 1995.

[Han96] Mark Handley.
SAP: Session Announcement Protocol,
Internet Draft[2] `<draft-ietf-mmusic-sap-00.txt>`.
Work in Progress, IETF MMUSIC Working Group, November 1996.

[Hau95] Franz J. Hauck.
Typen, Klassen und Vererbung in verteilten objektorientierten Systemen.
PhD thesis, IMMD IV, University of Erlangen-Nürnberg, Germany, May 1995.

[HEF+91] Franz J. Hauck, Thomas Eirich, Michael Fäustle, Jürgen Kleinöder, Rainer Pruy, Peter Schlenk.
Das PM Projekt.
Technical Report TR-I4-6-91, IMMD IV, University of Erlangen-Nürnberg, Germany, November 1991.

[Hei96] Bernd Heinrichs.
Multimedia im Netz.
Berlin, Germany: Springer, January 1996.

[HJ96] Mark Handley, Van Jacobson.
SDP: Session Description Protocol,
Internet Draft[2] `<draft-ietf-mmusic-sdp-02.txt>`.
Work in Progress, IETF MMUSIC Working Group, November 1996.

[HS91] Ralf Guido Herrtwich, Ralf Steinmetz.
"Towards Integrated Multimedia Systems: Why and How" in *Informatik Fachberichte no. 293*, p. 465–474.
Berlin, Germany: Springer, 1991.

[HSS96] Mark Handley, Henning Schulzrinne, Eve Schooler.
SIP: Session Initiation Protocol,
Internet Draft[2] `<draft-ietf-mmusic-sip-01.txt>`.
Work in Progress, IETF MMUSIC Working Group, December 1996.

[Hui95] Christian Huitema.
Routing in the Internet.
Englewood Cliffs, NJ: Prentice-Hall, 1995.

[Hus96] Dirk Husemann.
Multimedia Data Streams In Distributed Object-Oriented Operating Systems.
PhD thesis, IMMD IV, University of Erlangen-Nürnberg, Germany, February 1996.

[HWC95] Mark Handley, Ian Wakeman, Jon Crowcraft.
"The Conference Control Channel Protocol (CCCP): A scalable base for building conference control applications" in *Proceedings of the SIGCOMM '95 Symposium on Communications Architectures and Protocols*, Cambridge, MA, August 1995.
Computer Communications Review, Volume 25(4), October 1995.

[KNS90] Wolfgang Klas, Erich J. Neuhold, Michael Schrefl.
"Using an Object-Oriented Approach to Model Multimedia Data" in *Computer Communications*, Volume 13(4), p. 204–216, May 1990.

[Par96] Peter Parnes.
RTP Extension for Scalable Reliable Multicast,
Internet Draft[2] `<draft-parnes-rtp-ext-srm-01.txt>`.
Work in Progress, IETF, November 1996.

[RFC822] David H. Crocker.
RFC[1] 822, *Standard for the Format of ARPA Internet Text Messages.*
August 1982.

[RFC1075] D. Waitzman, Craig Partridge, Steve Deering.
RFC[1] 1075, *Distance Vector Multicast Routing Protocol.*
IETF Network Working Group, November 1988.

[RFC1112] Steve Deering.
RFC[1] 1112, *Host Extensions for IP Multicasting.*
IETF Network Working Group, August 1989.

[RFC1301] Susan M. Armstrong, Alan O. Freier, Keith A. Marzullo.
RFC[1] 1301, *Multicast Transport Protocol.*
IETF Network Working Group, February 1992.

[RFC1305] David L. Mills.
RFC[1] 1305, *Network Time Protocol (Version 3)—Specification, Implementation and Analysis.*
IETF Network Working Group, March 1992.

[RFC1363] Craig Partridge.
RFC[1] 1363, *A Proposed Flow Specification.*
IETF Network Working Group, September 1992.

[RFC1458] Bob Braudes, Steve Zabele.
RFC[1] 1458, *Requirements for Multicast Protocols.*
IETF Network Working Group, May 1993.

[RFC1577] Mark Laubach.
RFC[1] 1577, *Classical IP and ARP over ATM.*
IETF Network Working Group, January 1994.

[RFC1584] John Moy.
RFC[1] 1584, *Multicast Extensions to OSPF.*
IETF Network Working Group, March 1994.

[RFC1633] Bob Braden, David Clark, Scott Shenker.
RFC[1] 1633, *Integrated Services in the Internet Architecture: an Overview.*
IETF Network Working Group, June 1994.

[RFC1700] Joyce K. Reynolds, Jon Postel.
RFC[1] 1700, STD 2, *Assigned Numbers.*
IETF Network Working Group, October 1994.

[RFC1821] Marty Borden, Eric S. Crawley, Bruce S. Davie, Stephen G. Batsell.
RFC[1] 1821, *Integration of Real-time Services in an IP-ATM Network Architecture.*
IETF Network Working Group, August 1995.

[RFC1832] Rai Srinivasan.
RFC[1] 1832, *XDR: External Data Representation Standard.*
IETF Network Working Group, August 1995.

[RFC1889] Henning Schulzrinne, Stephen L. Casner, Ron Frederick, Van Jacobson.
RFC[1] 1889, *RTP: A Transport Protocol for Real-Time Applications.*
IETF Audio-Video Transport Working Group, January 1996.

[RFC1890] Henning Schulzrinne.
RFC[1] 1890, *RTP Profile for Audio and Video Conferences with Minimal Control.*
IETF Audio-Video Transport Working Group, January 1996.

[RFC2029] Michael F. Speer, Don Hoffman.
RFC[1] 2029, *RTP Payload Format of Sun's CellB Video Encoding.*
IETF Network Working Group, October 1996.

[RFC2032] Thierry Turletti, Christian Huitema.
RFC[1] 2032, *RTP Payload Format for H.261 Video Streams.*
IETF Network Working Group, October 1996.

[RFC2035] Lance M. Berc, William C. Fenner, Ron Frederick, Steve McCanne.
RFC[1] 2035, *RTP Payload Format for JPEG-Compressed Video.*
IETF Network Working Group, October 1996.

[RFC2038] Don Hoffman, Gerard Fernando, Vivek Goyal.
RFC[1] 2038, *RTP Payload Format for MPEG1/MPEG2 Video.*
IETF Network Working Group, October 1996.

[SB95] Scott Shenker, L. Breslau. "Two Issues in Reservation Establishment" in *Proceedings of the SIGCOMM '95 Symposium on Communications Architectures and Protocols*, Cambridge, MA, August 1995.
Computer Communications Review, Volume (4)25, October 1995.

[Sch96b] Henning Schulzrinne.
A real-time stream control protocol (RTSP'),
Internet Draft[2] `<draft-ietf-mmusic-stream-00.txt>`.
Work in Progress, IETF MMUSIC Working Group, November 1996.

[SF92] Ralf Steinmetz, J. Christian Fritzsche.
"Abstractions for Continuous-Media Programming" in *Computer Communications.*
Volume 15(6), p. 396–402, July/August 1992.

[SPG96] Scott Shenker, Craig Partridge, Roch Guerin.
Specification of Guaranteed Quality of Service,
Internet Draft[2] `<draft-ietf-intserv-guaranteed-svc-06.txt>`.
Work in Progress, IETF Integrated Services Working Group, August 1996.

[Ste94] Ralf Steinmetz.
Multimedia-Technologie: Einführung und Grundlagen.
Berlin, Germany: Springer, 1994.

[Ste95] Ralf Steinmetz.
"Analysing the Multimedia Operating System" in *IEEE Multimedia*, Volume 2(1), p. 68–84, Spring 1995.

[SW96a] Scott Shenker, John Wroclawski.
Network Element Service Specification Template,
Internet Draft[2] `<draft-ietf-intserv-svc-template-03.txt>`.
Work in Progress, IETF Integrated Services Working Group, November 1996.

[SW96b] Scott Shenker, John Wroclawski.
General Characterization Parameters for Integrated Service Network Element,
Internet Draft[2] `<draft-ietf-intserv-charac-02.txt>`.
Work in Progress, IETF Integrated Services Working Group, October 1996.

[Tan96] Andrew S. Tanenbaum.
Computer Networks, Third Edition.
Upper Saddle River, NJ: Prentice-Hall, 1996.

[VKBG95] Andreas Vogel, Brigitte Kerhervé, Gregor von Bochmann, Jan Gecsei.
"Distributed Multimedia and QOS: A Survey" in *IEEE MultiMedia*, Volume 2(2), p. 10–19, Summer 1995.

[Wro96a] John Wroclawski.
The Use of RSVP with IETF Integrated Services,
Internet Draft[2] `<draft-ietf-intserv-rsvp-use-01.txt>`.
Work in Progress, IETF Integrated Services Working Group, October 1996.

[Wro96b] John Wroclawski.
Specification of the Controlled-Load Network Element Service,
Internet Draft[2] `<draft-ietf-intserv-ctrl-load-svc-04.txt>`.
Work in Progress, IETF Integrated Services Working Group, November 1996.

[ZDESZ93] Lixia Zhang, Steve Deering, Deborah Estrin, Scott Shenker, Daniel Zappala.
"RSVP: A New Resource ReSerVation Protocol" in *IEEE Network*, Volume 7(5), p. 8–18, September 1993.

[1] All Internet RFCs are available from the FTP server at `ftp://ftp.nordu.net/rfc/`.
[2] Most Internet drafts are available at `ftp://ftp.nordu.net/internet-drafts/`.
[3] This draft is provided with the ISI RSVP implementation available at `ftp://ftp.isi.edu/rsvp/release/`.
[4] This article is also available at `http://www.fokus.gmd.de/step/acontrol/ac.html`.
[5] This document is available at `http://www.javasoft.com/doc/language_environment/`.
[6] This paper is available at `http://cuiwww.unige.ch/ftp/OO-articles/timeBasedMedia.ps.Z`.