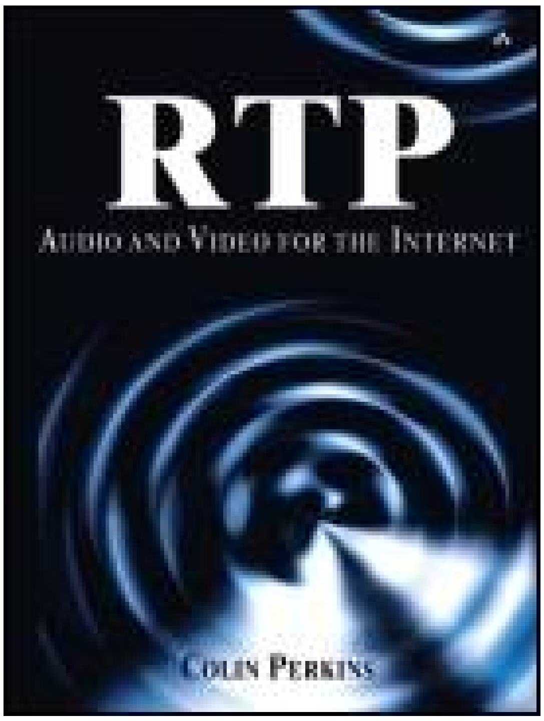


RTP: Audio and Video for the Internet By Colin Perkings

Publisher: Addison Wesley June 12,2003 ISBN:0-672-32249-8 Page: 432

The Real-time Transport Protocol (RTP) provides a framework for delivery of audio and video across IP networks and unprecedented quality and reliability. In *RTP: Audio and Video for the Internet*, Colin Perkins, a leader of the RTP standardization process in the IETF, offers readers detailed technical guidance for designing, implementing, and managing any RTP-based system.

By bringing together crucial information that was previously scattered or difficult to find, Perkins has created an incredible resource that enables professionals to leverage RTP's benefits in a wide range of Voice-over IP(VolP) and streaming media applications. He demonstrates how RTP supports audio/video transmission in IP networks, and shares strategies for maximizing performance, robustness, security, and privacy.

Comprehensive, exceptionally clear, and replete with examples, this book is the definitive RTP reference for every audio/video application designer, developer,researcher, and administrator.

Acknowledgments

A book such as this is not written in isolation; rather it is shaped by the author's experiences and interactions with other researchers and practitioners. In large part, I gained my experience while working at University College London. I am grateful to Vicky Hardman, Peter Kirstein, and Angela Sasse for the opportunity to work on their projects, and to Anna Bouch, Ian Brown, Anna Conniff, Jon Crowcroft, Panos Gevros, Atanu Ghosh, Mark Handley, Tristan Henderson, Orion Hodson, Nadia Kausar, Isidor Kouvelas, Piers O'Hanlon, Louise Sheeran, Lorenzo Vicisano, and everyone else associated with G11, for providing a stimulating working environment, and a distracting social scene.

I wish to thank my colleagues at the USC Information Sciences Institute for their support, in particular Alec Aakesson, Helen Ellis, Jarda Flidr, Ladan Gharai, Tom Lehman, Dan Massey, and Nikhil Mittal. Allison Mankin provided the opportunity to work at USC/ISI, for which I am grateful.

On a personal note, Peter Phillips, Stewart Cambridge, Sonja Krugmann, and Alison Gardiner each helped me make the big move, in their own special way. I thank you.

The staff at Addison-Wesley did an excellent job in the production of this book. In particular, Dayna Isley and Jessica Goldstein provided encouragement to a new author and showed great patience during endless revisions. Thanks are also due to Amy Fleischer, Elizabeth Finney, Laurie McGuire, Cheri Clark, Rebecca Martin, and Stephanie Hiebert. The technical editors—Steve Casner and Orion Hodson—did sterling work, significantly improving the quality of the book, correcting many mistakes and contributing significantly to the text. Any errors that remain are mine alone.

Introduction

This book describes the protocols, standards, and architecture of systems that deliver real-time voice, music, and video over IP networks, such as the Internet. These systems include voice-over-IP, telephony, teleconferencing, streaming video, and webcasting applications. The book focuses on media transport: how to deliver audio and video reliably across an IP network, how to ensure high quality in the face of network problems, and how to ensure that the system is secure.

The book adopts a standards-based approach, based around the Real-time Transport Protocol (RTP) and its associated profiles and payload formats. It describes the RTP framework, how to build a system that uses that framework, and extensions to RTP for security and reliability.

Many media codecs are suitable for use with RTP—for example, MPEG audio and video; ITU H.261 and H.263 video; G.711, G.722, G.726, G.728, and G.729 audio; and industry standards such as GSM, QCELP, and AMR audio. RTP implementations typically integrate existing media codecs, rather than developing them specifically. Accordingly, this book describes how media codecs are integrated into an RTP system, but not how media codecs are designed.

Call setup, session initiation, and control protocols, such as SIP, RTSP, and H.323, are also outside the scope of this book. Most RTP implementations are used as part of a complete system, driven by one of these control protocols. However, the interactions between the various parts of the system are limited, and it is possible to understand media transport without understanding the signaling. Similarly, session description using SDP is not covered, because it is part of the signaling.

Resource reservation is useful in some situations, but it is not required for the correct operation of RTP. This book touches on the use of resource reservation through both the Integrated Services and the Differentiated Services frameworks, but it does not go into details.

That these areas are not covered in this book does not mean that they are unimportant. A system using RTP will use a range of media codecs and will employ some form of call setup, session initiation, or control. The way this is done depends on the application, though: The needs of a telephony system are very different from those of a webcasting application. This book describes only the media transport layer that is common to all those systems.

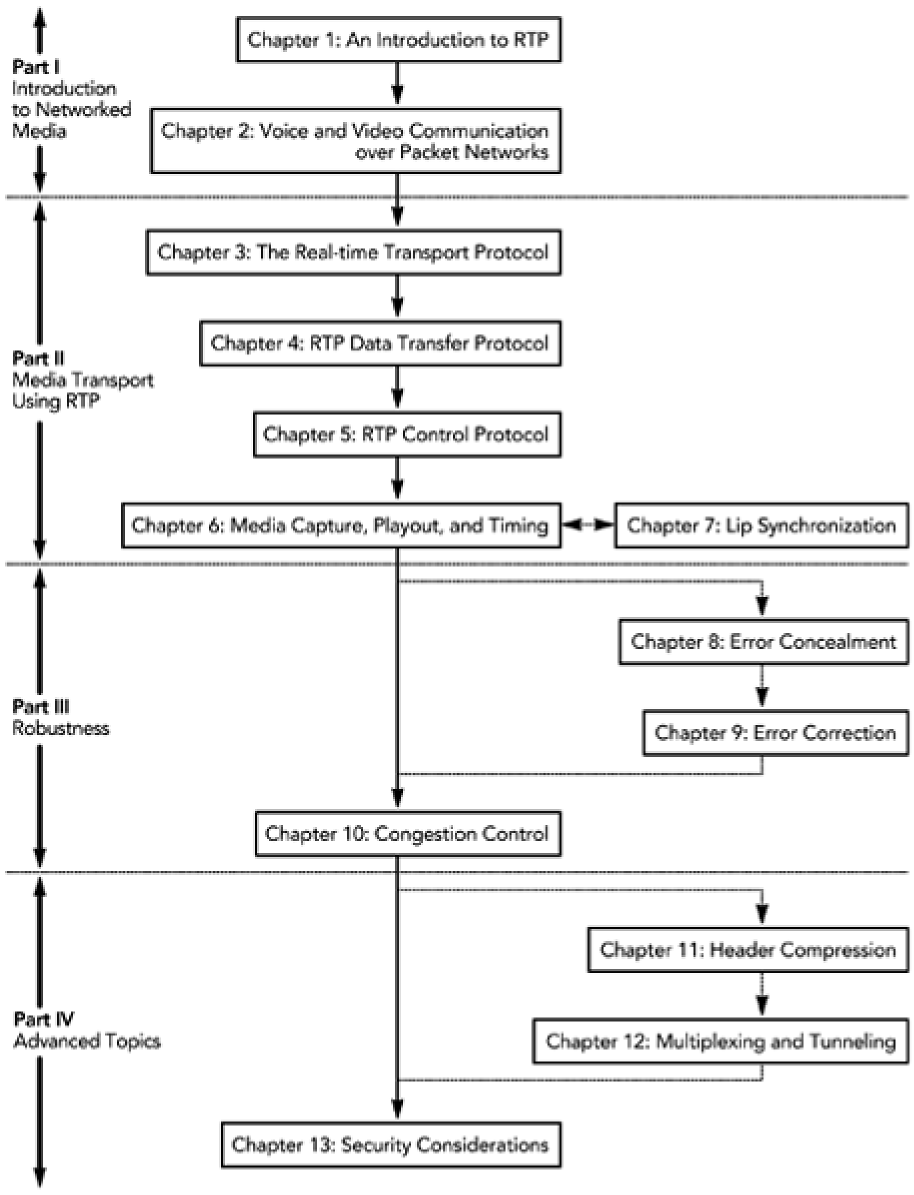
Organization of the Book

The book is logically divided into four parts: Part I, Introduction to Networked Multimedia, introduces the problem space, provides background, and outlines the properties of the Internet that affect audio/video transport:

- Chapter 1, An Introduction to RTP, gives a brief introduction to the Real-time Transport Protocol, outlines the relationship between RTP and other standards, and describes the scope of the book.
- Chapter 2, Voice and Video Communication over Packet Networks, describes the unique environment provided by IP networks, and how this environment affects packet audio/video applications.

The next five chapters, which constitute Part II, Media Transport Using RTP, discuss the basics of the Real-time Transport Protocol.

Road Map for This Book


Part I
Introduction to Networked Media
Chapter 1: An Introduction to RTP
Chapter 2: Voice and Video Communication over Packet Networks
Part II
Media Transport Using RTP
Chapter 3: The Real-time Transport Protocol
Chapter 4: RTP Data Transfer Protocol
Chapter 5: RTP Control Protocol
Chapter 6: Media Capture, Playout, and Timing
Chapter 7: Lip Synchronization
Part III
Robustness
Chapter 8: Error Concealment
Chapter 9: Error Correction
Chapter 10: Congestion Control
Part IV
Advanced Topics
Chapter 11: Header Compression
Chapter 12: Multiplexing and Tunneling
Chapter 13: Security Considerations

You will need this information to design and build a tool for voice-over-IP, streaming music or video, and so on.

- Chapter 3, The Real-time Transport Protocol, introduces RTP and related standards, describes how they fit together, and outlines the design philosophy underpinning the protocol.
- Chapter 4, RTP Data Transfer Protocol, gives a detailed description of the transport protocol used to convey audiovisual data over IP networks.
- Chapter 5, RTP Control Protocol, describes the control protocol, which provides reception quality feedback, membership control, and synchronization.
- Chapter 6, Media Capture, Playout, and Timing, explains how a receiver can reconstruct the audiovisual data and play it out to the user with correct timing.

- Chapter 7, Lip Synchronization, addresses a related problem: how to synchronize media streams—for example, to get lip synchronization.

Part III, Robustness, discusses how to make your application reliable in the face of network problems, and how to compensate for loss and congestion in the network. You can build a system without using these techniques, but the audio will sound a lot better, and the pictures will be smoother and less susceptible to corruption, if you apply them.

- Chapter 8, Error Concealment, addresses the issue of concealing errors caused by incomplete reception, describing several techniques a receiver can use to hide network problems.
- Chapter 9, Error Correction, describes techniques that can be used to repair damaged media streams, where the sender and receiver cooperate in repairing the media stream.
- Chapter 10, Congestion Control, discusses the way the Internet responds to congestion, and how this affects the design of audio/video applications.

Finally, Part IV, Advanced Topics, describes various techniques that have more specialized use. Many implementations do not use these features, but they can significantly improve performance in some cases:

- Chapter 11, Header Compression, outlines a technique that can significantly increase the efficiency of RTP on low-speed network links, such as dial-up modems or cellular radio links.
- Chapter 12, Multiplexing and Tunneling, describes how several media streams can be combined into one. The intent is to improve efficiency when many similar streams are to be transported between gateway devices.
- Chapter 13, Security Considerations, describes how encryption and authentication technology can be used to protect RTP sessions; it also describes common security and privacy issues.

Intended Audience

This book describes audio/video transport over IP networks in considerable detail. It assumes some basic familiarity with IP network programming and the operation of network protocol stacks, and it builds on this knowledge to describe the features unique to audio/video transport. An extensive list of references is included, pointing readers to additional information on specific topics and to background reading material.

Several classes of readers might be expected to find this book useful:

- **Engineers**. The primary audience is those building voice-over-IP applications, teleconferencing systems, and streaming media and webcasting applications. This book is a guide to the design and implementation of the media engine of such systems. It should be read in conjunction with the relevant technical standards, and it builds on those standards to show how a system is built. This book does not discuss signaling (for example, SIP, RTSP, or H.323), which is a separate subject worthy of a book in its own right. Instead it talks in detail about media transport, and how to achieve good-quality audio and smooth-motion video over IP networks.
- **Students**. The book can be read as an accompaniment to a course in network protocol design or telecommunications, at either a graduate or an advanced undergraduate level. Familiarity with IP networks and layered protocol architectures is assumed. The unique aspects of protocols for real-time audio/video transport are highlighted, as are the differences from a typical layered system model. The cross-disciplinary nature of the subject is highlighted, in particular the relation between the psychology of human perception and the demands of robust media delivery.
- **Researchers**. Academics and industrial researchers can use this book as a source of information about the standards and algorithms that constitute the current state of the art in real-time audio/video transport over IP networks. Pointers to the literature are included in the References section, and they will be useful starting points for those seeking further depth and areas where more research is needed.
- **Network administrators**. An understanding of the technical protocols underpinning the common streaming audio/video applications is useful for those administering computer networks—to show how those applications can affect the behavior of the network, and how the network can be engineered to suit those applications better. This book includes extensive discussion of the most common network behavior (and how applications can adapt to it), the needs of congestion control, and the security implications of real-time audio/video traffic.

In summary, this book can be used as a reference, in conjunction with the technical standards, as a study guide, or as part of an advanced course on network protocol design or communication technology.

Chapter 1. An Introduction to RTP

A Brief History of Audio/Video Networking

A Snapshot of RTP

Related Standards

Overview of an RTP Implementation

The Internet is changing: Static content is giving way to streaming video, text is being replaced by music and the spoken word, and interactive audio and video is becoming commonplace. These changes require new applications, and they pose new and unique challenges for application designers.

This book describes how to build these new applications: voice-over-IP, telephony, teleconferencing, streaming video, and webcasting. It looks at the challenges inherent in reliable delivery of audio and video across an IP network, and it explains how to ensure high quality in the face of network problems, as well as how to ensure that the system is secure. The emphasis is on open standards, in particular those devised by the Internet Engineering Task Force (IETF) and the International Telecommunications Union (ITU), rather than on proprietary solutions.

This chapter begins our examination of the Real-time Transport Protocol (RTP) with a brief look at the history of audio/video networking and an overview of RTP and its relation to other standards.

A Brief History of Audio/Video Networking

The idea of using packet networks—such as the Internet—to transport voice and video is not new. Experiments with voice over packet networks stretch back to the early 1970s. The first RFC on this subject—the Network Voice Protocol (NVP)[1]—dates from 1977. Video came later, but still there is over ten years of experience with audio/video conferencing and streaming on the Internet.

Early Packet Voice and Video Experiments

The initial developers of NVP were researchers transmitting packet voice over the ARPANET, the predecessor to the Internet. The ARPANET provided a reliable-stream service (analogous to TCP/IP), but this introduced too much delay, so an "uncontrolled packet" service was developed, akin to the modern UDP/IP datagrams used with RTP. The NVP was layered directly over this uncontrolled packet service. Later the experiments were extended beyond the ARPANET to interoperate with the Packet Radio Network and the Atlantic Satellite Network (SATNET), running NVP over those networks.

All of these early experiments were limited to one or two voice channels at a time by the low bandwidth of the early networks. In the 1980s, the creation of the 3-Mbps Wideband Satellite Network enabled not only a larger number of voice channels but also the development of packet video. To access the one-hop, reserved-bandwidth, multicast service of the satellite network, a connection-oriented inter-network protocol called the Stream Protocol (ST) was developed. Both a second version of NVP, called NVP-II, and a companion Packet Video Protocol were transported over ST to provide a prototype packet-switched video teleconferencing service.

In 1989–1990, the satellite network was replaced with the Terrestrial Wideband Network and a research network called DARTnet while ST evolved into ST-II. The packet video conferencing system was put into scheduled production to support geographically distributed meetings of network researchers and others at up to five sites simultaneously.

ST and ST-II were operated in parallel with IP at the inter-network layer but achieved only limited deployment on government and research networks. As an alternative, initial deployment of conferencing using IP began on DARTnet, enabling multiparty conferences with NVP-II transported over multicast UDP/IP. At the March 1992 meeting of the IETF, audio was transmitted across the Internet to 20 sites on three continents over multicast "tunnels"—the Mbone (which stands for "multicast backbone")—extended from DARTnet. At that same meeting, development of RTP was begun.

Audio and Video on the Internet

Following from these early experiments, interest in video conferencing within the Internet community took hold in the early 1990s. At about this time, the processing power and multimedia capabilities of workstations and PCs became sufficient to enable the simultaneous capture, compression, and playback of audio and video streams. In parallel, development of IP multicast allowed the transmission of real-time data to any number of recipients connected to the Internet.

Video conferencing and multimedia streaming were obvious and well-executed multicast applications. Research groups took to developing tools such as vic and vat from the Lawrence Berkeley Laboratory,[87] nevot from the University of Massachusetts, the INRIA video conferencing system, nv from Xerox PARC, and rat from University College London.[77] These tools followed a new approach to conferencing, based on connectionless protocols, the end-to-end argument, and application-level framing.[65], [70], [76] Conferences were minimally managed, with no admission or floor control, and the transport layer was thin and adaptive. Multicast was used both for wide-area data transmission and as an interprocess communication mechanism between applications on the same machine (to exchange synchronization information between audio and video tools). The resulting collaborative environment consisted of lightly coupled applications and highly distributed participants.

The multicast conferencing (Mbone) tools had a significant impact: They led to widespread understanding of the problems inherent in delivering real-time media over IP networks, the need for scalable solutions, and error and congestion control. They also directly influenced the development of several key protocols and standards.

RTP was developed by the IETF in the period 1992–1996, building on NVP-II and the protocol used in the original vat tool. The multicast conferencing tools used RTP as their sole data transfer and control protocol; accordingly, RTP not only includes facilities for media delivery, but also supports membership management, lip synchronization, and reception quality reporting.

In addition to RTP for transporting real-time media, other protocols had to be developed to coordinate and control the media streams. The Session Announcement Protocol (SAP)[35] was developed to advertise the existence of multicast data streams. Announcements of sessions were themselves multicast, and any multicast-capable host could receive SAP announcements and learn what meetings and transmissions were happening. Within announcements, the Session Description Protocol (SDP)[15] described the transport addresses, compression, and packetization schemes to be used by senders and receivers in multicast sessions. Lack of multicast deployment, and the rise of the World Wide Web, have largely superseded the concept of a distributed multicast directory, but SDP is still used widely today.

Finally, the Mbone conferencing community led development of the Session Initiation Protocol (SIP).[28] SIP was intended as a lightweight means of finding participants and initiating a multicast session with a specific set of participants. In its early incarnation, SIP included little in the way of call control and negotiation support because such aspects were not used with the Mbone conferencing environment. It has since become a more comprehensive protocol, including extensive negotiation and control features.

ITU Standards

In parallel with the early packet voice work was the development of the Integrated Services Digital Network (ISDN)—the digital version of the plain old telephone system—and an associated set of video conferencing standards. These standards, based around ITU recommendation H.320, used circuit-switched links and so are not directly relevant to our discussion of packet audio and video. However, they did pioneer many of the compression algorithms used today (for example, H.261 video).

The growth of the Internet and the widespread deployment of local area networking equipment in the commercial world led the ITU to extend the H.320 series of protocols. Specifically, they sought to make the protocols suitable for "local area networks which provide a non-guaranteed quality of service," IP being a classic protocol suite fitting the description. The result was the H.323 series of recommendations.

H.323 was first published in 1997[62] and has undergone several revisions since. It provides a framework consisting of media transport, call signaling, and conference control. The signaling and control functions are defined in ITU recommendations H.225.0 and H.245. Initially the signaling protocols focused principally on interoperating with ISDN conferencing using H.320, and as a result suffered from a cumbersome session setup process that was simplified in later versions of the standard. For media transport, the ITU working group adopted RTP. However, H.323 uses only the media transport functionality of RTP and makes little use of the control and reporting elements.

H.323 met with reasonable success in the marketplace, with several hardware and software products built to support the suite of H.323 technologies. Development experience led to complaints about its complexity, in particular the complex setup procedure of H.323 version 1 and the use of binary message formats for the signaling. Some of these issues were addressed in later versions of H.323, but in the intervening period interest in alternatives grew.

One of those alternatives, which we have already touched on, was SIP. The initial SIP specification was published by the IETF in 1999,[28] as the outcome of an academic research project with virtually no commercial interest. It has since come to be seen as a replacement for H.323 in many quarters, and it is being applied to more varied applications, such as text messaging systems and voice-over-IP. In addition, it is under consideration for use in third-generation cellular telephony systems,[115] and it has gathered considerable industry backing.

The ITU has more recently produced recommendation H.332, which combines a tightly coupled H.323 conference with a lightweight multicast conference. The result is useful for scenarios such as an online seminar, in which the H.323 part of the conference allows close interaction among a panel of speakers while a passive audience watches via multicast.

Audio/Video Streaming

In parallel with the development of multicast conferencing and H.323, the World Wide Web revolution took place, bringing glossy content and public acceptance to the Internet. Advances in network bandwidth and end-system capacity made possible the inclusion of streaming audio and video along with Web pages, with systems such as RealAudio and QuickTime leading the way. The growing market in such systems fostered a desire to devise a standard control mechanism for streaming content. The result was the Real-Time Streaming Protocol (RTSP),[14] providing initiation and VCR-like control of streaming presentations; RTSP was standardized in 1998. RTSP builds on existing standards: It closely resembles HTTP in operation, and it can use SDP for session description and RTP for media transport.

A Snapshot of RTP

The key standard for audio/video transport in IP networks is the Real-time Transport Protocol (RTP), along with its associated profiles and payload formats. RTP aims to provide services useful for the transport of real-time media, such as audio and video, over IP networks. These services include timing recovery, loss detection and correction, payload and source identification, reception quality feedback, media synchronization, and membership management. RTP was originally designed for use in multicast conferences, using the lightweight sessions model. Since that time, it has proven useful for a range of other applications: in H.323 video conferencing, webcasting, and TV distribution; and in both wired and cellular telephony. The protocol has been demonstrated to scale from point-to-point use to multicast sessions with thousands of users, and from low-bandwidth cellular telephony applications to the delivery of uncompressed High-Definition Television (HDTV) signals at gigabit rates.

RTP was developed by the Audio/Video Transport working group of the IETF and has since been adopted by the ITU as part of its H.323 series of recommendations, and by various other standards organizations. The first version of RTP was completed in January 1996.[6] RTP needs to be profiled for particular uses before it is complete; an initial profile was defined along with the RTP specification,[7] and several more profiles are under development. Profiles are accompanied by several payload format specifications, describing the transport of a particular media format. Development of RTP is ongoing, and a revision is nearing completion at the time of this writing.[49], [50]

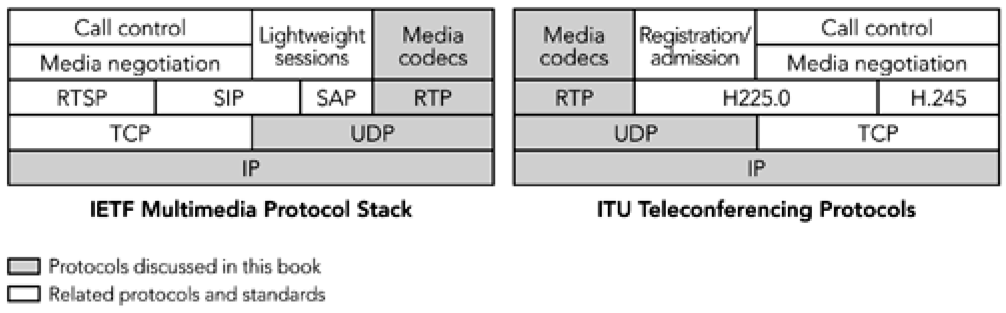
A detailed introduction to RTP is provided in Chapter 3, The Real-time Transport Protocol, and most of this book discusses the design of systems that use RTP and its various extensions.

Related Standards

In addition to RTP, a complete system typically requires the use of various other protocols and standards for session announcement, initiation, and control; media compression; and network transport.

Figure 1.1 shows how the negotiation and call control protocols, the media transport layer (provided by RTP), the compression-decompression algorithms (codecs), and the underlying network are related, according to both the IETF and ITU conferencing frameworks. The two parallel sets of call control and media negotiation standards use the same media transport framework. Like-wise, the media codecs are common no matter how the session is negotiated and irrespective of the underlying network transport.

Figure 1.1. IETF and ITU Protocols for Audio/Video Transport on the Internet



The relation between these standards and RTP is outlined further in Chapter 3, The Real-time Transport Protocol. However, the main focus of this book is media transport, rather than signaling and control.

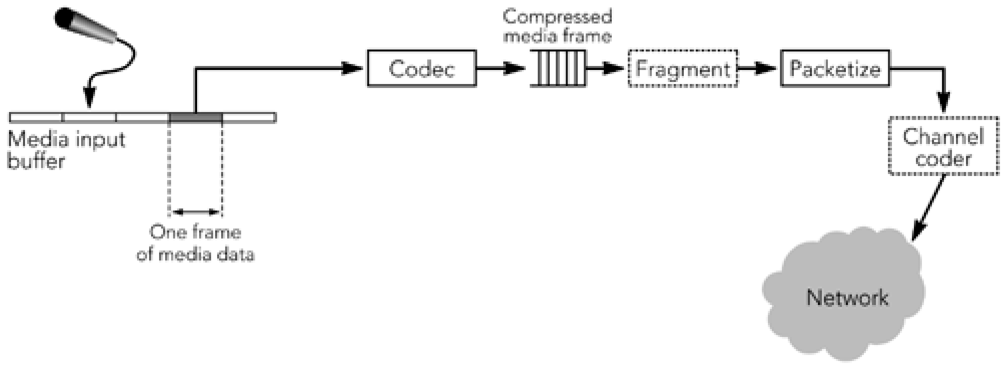
Overview of an RTP Implementation

As Figure 1.1 shows, the core of any system for delivery of real-time audio/video over IP is RTP: It provides the common media transport layer, independent of the signaling protocol and application. Before we look in more detail at RTP and the design of systems using RTP, it will be useful to have an overview of the responsibilities of RTP senders and receivers in a system.

Behavior of an RTP Sender

A sender is responsible for capturing and transforming audiovisual data for transmission, as well as for generating RTP packets. It may also participate in error correction and congestion control by adapting the transmitted media stream in response to receiver feedback. A diagram of the sending process is shown in Figure 1.2.

Figure 1.2. Block Diagram of an RTP Sender



Uncompressed media data—audio or video—is captured into a buffer, from which compressed frames are produced. Frames may be encoded in several ways depending on the compression algorithm used, and encoded frames may depend on both earlier and later data.

Compressed frames are loaded into RTP packets, ready for sending. If frames are large, they may be fragmented into several RTP packets; if they are small, several frames may be bundled into a single RTP packet. Depending on the error correction scheme in use, a channel coder may be used to generate error correction packets or to reorder packets before transmission.

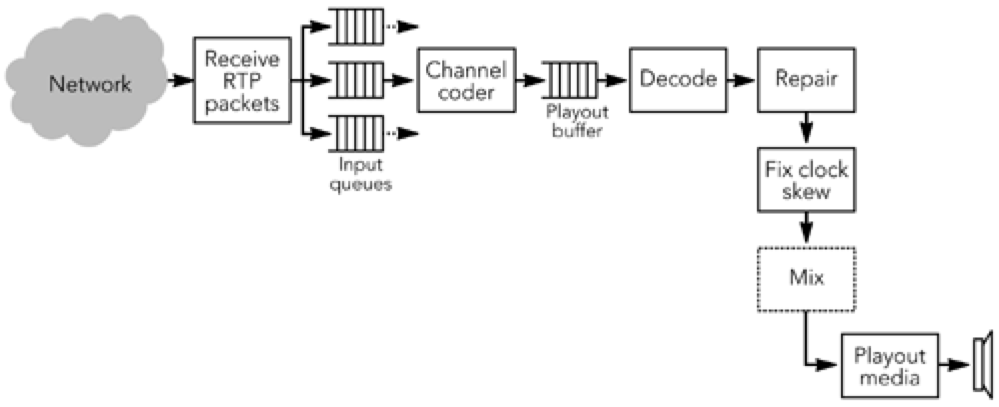
After the RTP packets have been sent, the buffered media data corresponding to those packets is eventually freed. The sender must not discard data that might be needed for error correction or for the encoding process. This requirement may mean that the sender must buffer data for some time after the corresponding packets have been sent, depending on the codec and error correction scheme used.

The sender is responsible for generating periodic status reports for the media streams it is generating, including those required for lip synchronization. It also receives reception quality feedback from other participants and may use that information to adapt its transmission.

Behavior of an RTP Receiver

A receiver is responsible for collecting RTP packets from the network, correcting any losses, recovering the timing, decompressing the media, and presenting the result to the user. It also sends reception quality feedback, allowing the sender to adapt the transmission to the receiver, and it maintains a database of participants in the session. A possible block diagram for the receiving process is shown in Figure 1.3; implementations sometimes perform the operations in a different order depending on their needs.

Figure 1.3. Block Diagram of an RTP Receiver



The first step of the receive process is to collect packets from the network, validate them for correctness, and insert them into a sender-specific input queue. Packets are collected from the input queue and passed to an optional channel-coding routine to correct for loss. Following the channel coder, packets are inserted into a source-specific playout buffer. The playout buffer is ordered by timestamp, and the process of inserting packets into the buffer corrects any reordering induced during transport. Packets remain in the playout buffer until complete frames have been received, and they are additionally buffered to remove any variation in interpacket timing caused by the network. Calculation of the amount of delay to add is one of the most critical aspects in the design of an RTP implementation. Each packet is tagged with the desired playout time for the corresponding frame.

After their playout time is reached, packets are grouped to form complete frames, and any damaged or missing frames are repaired. Following any necessary repairs, frames are decoded (depending on the codec used, it may be necessary to decode the media before missing frames can be repaired). At this point there may be observable differences in the nominal clock rates of the sender and receiver. Such differences manifest themselves as drift in the value of the RTP media clock relative to the playout clock. The receiver must compensate for this clock skew to avoid gaps in the playout.

Finally, the media data is played out to the user. Depending on the media format and output device, it may be possible to play each stream individually—for example, presenting several video streams, each in its own window. Alternatively, it may be necessary to mix the media from all sources into a single stream for playout—for example, combining several audio sources for playout via a single set of speakers.

As is evident from this brief overview, the operation of an RTP receiver is complex, and it is somewhat more involved than the operation of a sender. This increased complexity is largely due to the variability of IP networks: Much of the complexity comes from the need to compensate for packet loss, and to recover the timing of a stream affected by jitter.

Summary

This chapter has introduced the protocols and standards for real-time delivery of multimedia over IP networks, in particular the Real-time Transport Protocol (RTP). The remainder of this book discusses the features and use of RTP in detail. The aim is to expand on the standards documents, explaining both the rationale behind the standards and possible implementation choices and their trade-offs.

Chapter 2. Voice and Video Communication Over Packet Networks

- TCP/IP and the OSI Reference Model
- Performance Characteristics of an IP Network
- Measuring IP Network Performance
- Effects of Transport Protocols
- Requirements for Audio/Video Transport in Packet Networks

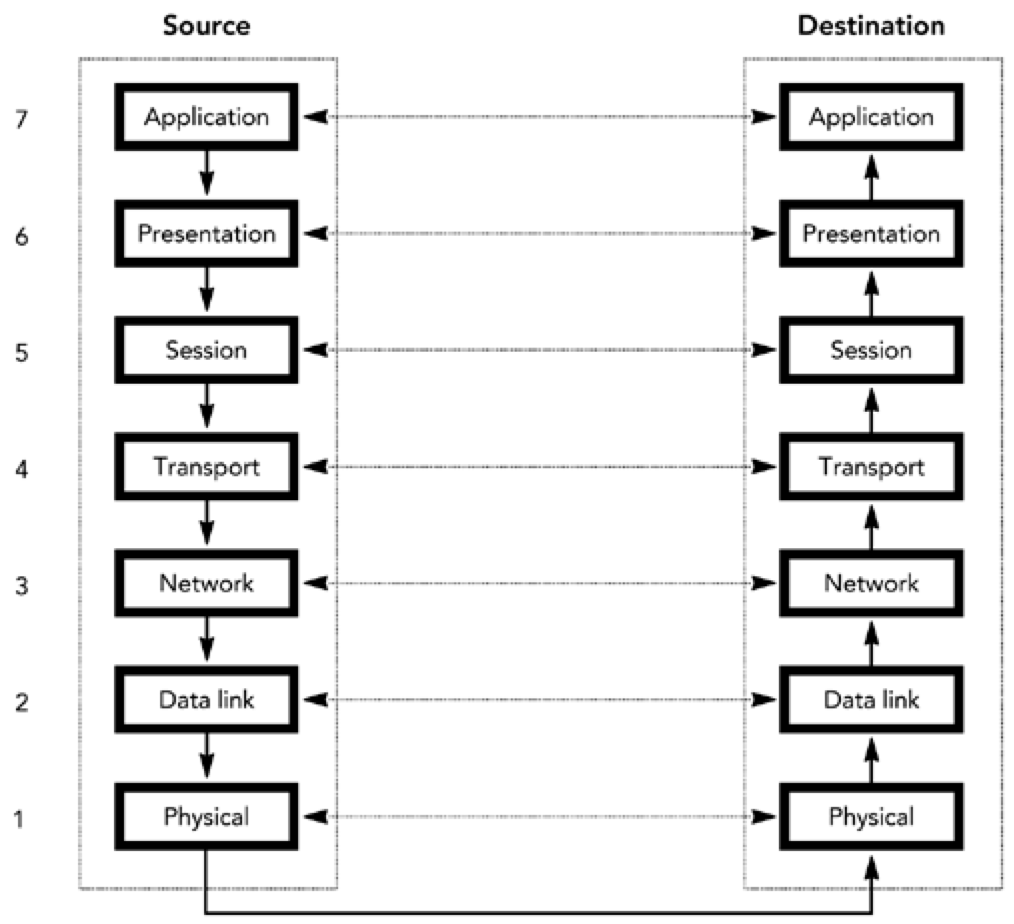
Before delving into details of RTP, you should understand the properties of IP networks such as the Internet, and how they affect voice and video communication. This chapter reviews the basics of the Internet architecture and outlines typical behavior of a network connection. This review is followed by a discussion of the transport requirements for audio and video, and how well these requirements are met by the network.

IP networks have unique characteristics that influence the design of applications and protocols for audio/video transport. Understanding these characteristics is vital if you are to appreciate the trade-offs involved in the design of RTP, and how they influence applications that use RTP.

TCP/IP and the OSI Reference Model

When you're thinking about computer networks, it is important to understand the concepts and implications of protocol layering. The OSI reference model,[93] illustrated in Figure 2.1, provides a useful basis for discussion and comparison of layered systems.

Figure 2.1. The OSI Reference Model



The model comprises a set of seven layers, each building on the services provided by the lower layer and, in turn, providing a more abstract service to the layer above. The functions of the layers are as listed here:

1. **Physical layer**. The lowest layer—the physical layer—includes the physical network connection devices and protocols, such as cables, plugs, switches, and electrical standards.
2. **Data link layer**. The data link layer builds on the physical connection; for example, it turns a twisted-pair cable into Ethernet. This layer provides framing for data transport units, defines how the link is shared among multiple connected devices, and supplies addressing for devices on each link.
3. **Network layer**. The network layer connects links, unifying them into a single network. It provides addressing and routing of messages through the network. It may also provide control of congestion in the switches, prioritization of certain messages, billing, and so on. A network layer device processes messages received from one link and dispatches them to another, using routing information exchanged with its peers at the far ends of those links.
4. **Transport layer**. The transport layer is the first end-to-end layer. It takes responsibility for delivery of messages from one system to another, using the services provided by the network layer. This responsibility includes

providing reliability and flow control if they are needed by the session layer and not provided by the network layer.

5. **Session layer**. The session layer manages transport connections in a fashion meaningful to the application. Examples include the Hypertext Transport Protocol (HTTP) used to retrieve Web pages, the Simple Mail Transfer Protocol (SMTP) negotiation during an e-mail exchange, and the management of the control and data channels in the File Transfer Protocol (FTP).
6. **Presentation layer**. The presentation layer describes the format of the data conveyed by the lower layers. Examples include the HTML (Hypertext Markup Language) used to describe the presentation of Web pages, the MIME (Multipurpose Internet Mail Extensions) standards describing e-mail message formats, and more mundane issues, such as the difference between text and binary transfers in FTP.
7. **Application layer**. The applications themselves—Web browsers and e-mail clients, for example—compose the top layer of the system, the application layer.

At each layer in the model, there is a logical communication between that layer on one host and the equivalent layer on another. When an application on one system wants to talk to an application on another system, the communication proceeds down through the layers at the source, passes over the physical connection, and then goes up the protocol stack at the destination.

For example, a Web browser *application* renders an HTML *presentation*, which is delivered using an HTTP *session*, over a TCP *transport* connection, via an IP *network*, over an Ethernet *data link*, using twisted-pair *physical* cable. Each step can be viewed as the instantiation of a particular layer of the model, going down through the protocol stack. The result is the transfer of a Web page from application (Web server) to application (Web browser).

The process is not always that simple: There may not be a direct physical connection between source and destination, in which case the connection must partially ascend the protocol stack at an intermediate gateway system. How far does it need to ascend? That depends on what is being connected. Here are some examples:

- The increasingly popular IEEE 802.11b wireless network uses base stations that connect one physical layer, typically wired Ethernet, to another—the wireless link—at the data link layer.
- An IP router provides an example of a gateway in which multiple data links are connected at the network level.
- Viewing a Web page on a mobile phone often requires the connection to ascend all the way to the presentation layer in the gateway, which converts from HTML to the Wireless Markup Language (WML) and relays the connection down to the different lower layers.

As the preceding discussion suggests, we can use the OSI reference model to describe the Internet. The fit is not perfect: The architecture of the Internet has evolved over time, in part pre-dating the OSI model, and often it exhibits much less strict layering than has been implied. Nevertheless, it is instructive to consider the relation of the Internet protocol suite to the OSI model, in particular the role taken by IP as a universal network layer.[113]

The lowest two layers of the OSI reference model can be related directly to the Internet, which works over a wide range of links such as dial-up modems, DSL, Ethernet, optical fiber, wireless, and satellite. Each of these links can be described in terms of the data link/physical layer split of the OSI model.

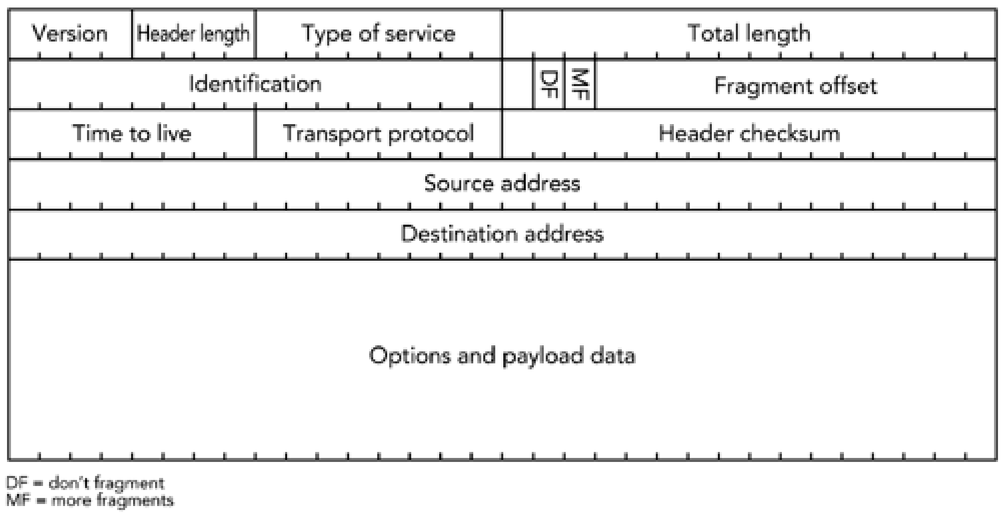
At the network layer, one particular protocol transforms a disparate set of private networks into the global Internet. This is the Internet Protocol (IP). The service provided by IP to the upper layers is simple: best-effort delivery of datagram packets to a named destination. Because this service is so simple, IP can run over a wide range of link layers, enabling the rapid spread of the Internet.

The simplicity comes with a price: IP does not guarantee any particular timeliness of delivery, or that a datagram will be delivered at all. Packets containing IP datagrams may be lost, reordered, delayed, or corrupted by the lower layers. IP does not attempt to correct these problems; rather it passes the datagrams to the upper layers exactly as they arrive. It does, however, provide the following services:

- Fragmentation, in case the datagram is larger than the maximum transmission unit of the underlying link layer
- A time-to-live field that prevents looped packets from circling forever
- A type-of-service label that can be used to provide priority for certain classes of packets
- An upper-layer protocol identifier to direct the packet to the correct transport layer
- Addressing of the endpoints—including multicast to address a group of receivers—and routing of datagrams to the correct destination

The format of an IP header, showing how these services map onto a packet, is illustrated in Figure 2.2.

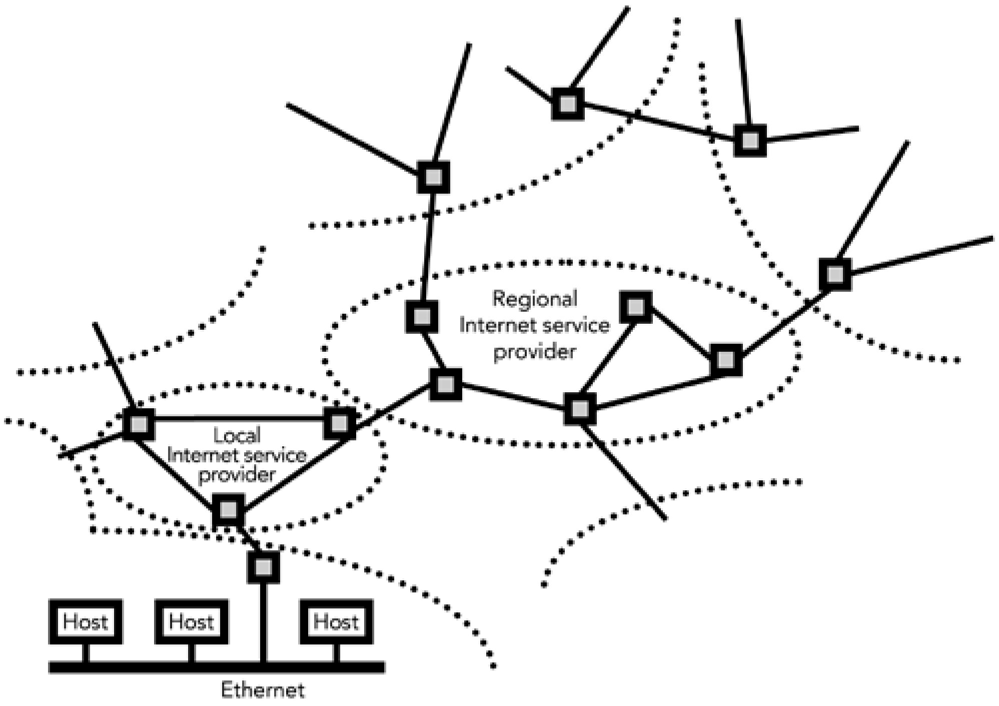
Figure 2.2. Format of an IP Header



The header in Figure 2.2 is an IPv4 header, the current standard on the Internet. There is a move to transition to IPv6, which provides essentially the same features but with a vastly increased address space (128-bit addresses, rather than 32-bit). If this transition occurs—a long-term prospect because it involves changes to every host and router connected to the Internet—it will provide for growth of the network by enabling many more machines to connect, but it will not significantly change the service in other ways.

The Internet Protocol provides the abstraction of a single network, but this does not change the underlying nature of the system. Even though it appears to be a single network, in reality the Internet consists of many separate networks, connected by gateways—now more commonly called *routers*—and unified by the single service and address space of IP. Figure 2.3 shows how individual networks make up the larger Internet. The different Internet service providers choose how to run their own portions of the global network: Some have high-capacity networks, with little congestion and high reliability; others do not.

Figure 2.3. An IP Inter-network



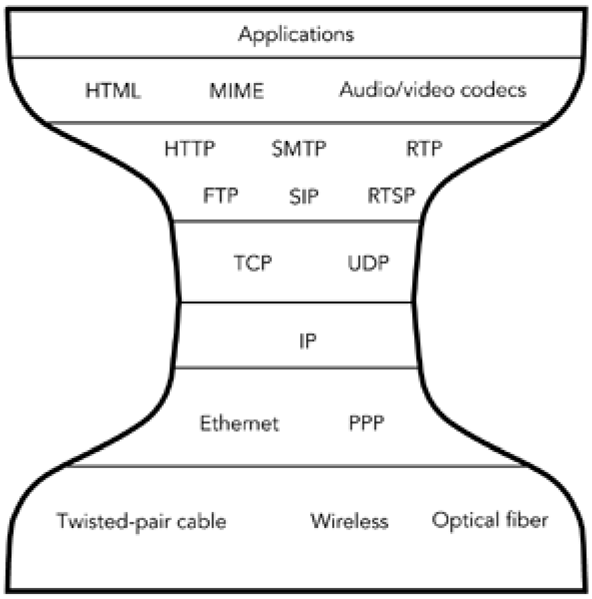
Within the maze of interconnected networks, the packets containing IP datagrams are individually routed to their destinations. Routers are not required to send a packet immediately; they may queue it for a short time if another packet is being transmitted on the outgoing link. They may also discard packets at times of congestion. The route taken by IP packets may change if the underlying network changes (for example, because of a link failure), possibly causing changes in delivery quality that can be observed by the upper-layer protocols.

Layered on top of IP in the Internet architecture are two common transport protocols: Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). TCP adapts the raw IP service to provide reliable, ordered delivery between service ports on each host, and it varies its transmission rate to match the characteristics of the network. UDP, on the other hand, provides a service similar to the raw IP service, with only the addition of service ports. TCP and UDP are discussed in more detail later in this chapter.

Above these transport protocols sit the familiar session protocols of the Internet universe, such as HTTP for Web access and SMTP to deliver e-mail. The stack is completed by various presentation layers (HTML, MIME) and the applications themselves.

What should be clear from this discussion is that IP plays a key role in the system: It provides an abstraction layer, hiding the details of the underlying network links and topology from the applications, and insulating the lower layers from the needs of the applications. This architecture is known as the hourglass model, as illustrated in Figure 2.4.

Figure 2.4. The Hourglass Model of the Internet Architecture



The primary factor determining the performance of a system communicating across the Internet is the IP layer. The higher-layer protocols can adapt to, and compensate for, the behavior of the IP layer to a certain degree, but poor performance there will result in poor performance for the entire system. The next two sections discuss the performance of the IP layer in some detail, noting its unique properties and the potential problems and benefits it brings.

Performance Characteristics of an IP Network

As is apparent from the hourglass model of the Internet architecture, an application is hidden from the details of the lower layers by the abstraction of IP. This means it's not possible to determine directly the types of networks across which an IP packet will have traveled—it could be anything from a 14.4-kilobit cellular radio connection to a multi-gigabit optical fiber—or the level of congestion of that network. The only means of discovering the performance of the network are observation and measurement.

So what do we need to measure, and how do we measure it? Luckily, the design of the IP layer means that the number of parameters is limited, and that number often can be further constrained by the needs of the application. The most important questions we can ask are these:

- What is the probability that a packet will be lost in the network?
- What is the probability that a packet will be corrupted in the network?
- How long does a packet take to traverse the network? Is the transit time constant or variable?

- What size of packet can be accommodated?
- What is the maximum rate at which we can send packets?

The next section provides some sample measurements for the first four listed parameters. The maximum rate is closely tied to the probability that packets are lost in the network, as discussed in Chapter 10, Congestion Control.

What affects such measurements? The obvious factor is the location of the measurement stations. Measurements taken between two systems on a LAN will clearly show properties different from those of a transatlantic connection! But geography is not the only factor; the number of links traversed (often referred to as the number of *hops*), the number of providers crossed, and the times at which the measurements are taken all are factors. The Internet is a large, complex, and dynamic system, so care must be taken to ensure that any measurements are representative of the part of the network where an application is to be used.

We also have to consider what sort of network is being used, what other traffic is present, and how much other traffic is present. To date, the vast majority of network paths are fixed, wired (either copper or optical fiber) connections, and the vast majority of traffic (96% of bytes, 62% of flows, according to a recent estimate[123]) is TCP based. The implications of these traffic patterns are as follows:

- Because the infrastructure is primarily wired and fixed, the links are very reliable, and loss is caused mostly by congestion in the routers.
- TCP transport makes the assumption that packet loss is a signal that the bottleneck bandwidth has been reached, congestion is occurring, and it should reduce its sending rate. A TCP flow will increase its sending rate until loss is observed, and then back off, as a way of determining the maximum rate a particular connection can support. Of course, the result is a temporary overloading of the bottleneck link, which may affect other traffic.

If the composition of the network infrastructure or the traffic changes, other sources of loss may become important. For example, a large increase in the number of wireless users would likely increase the proportion of loss due to packet corruption and interference on the wireless links. In another example, if the proportion of multimedia traffic using transports other than TCP increased, and those transports did not react to loss in the same way as TCP does, the loss patterns would probably change because of variation in the dynamics of congestion control.

As we develop new applications that run on IP, we have to be aware of the changes we are bringing to the network, to ensure that we don't cause problems to other users. Chapter 10, Congestion Control, discusses this issue in more detail.

Measuring IP Network Performance

This section outlines some of the available data on IP network performance, including published results on average packet loss, patterns of loss, packet corruption and duplication, transit time, and the effects of multicast.

Several studies have measured network behavior over a wide range of conditions on the public Internet. For example, Paxson reports on the behavior of 20,000 transfers among 35 sites in 9 countries;[124], [95] Handley[122] and Bolot[67], [66] report on the behavior of multicast sessions; and Yajnik, Moon, Kurose, and Towsley report on the temporal dependence in packet loss statistics.[89], [108], [109] Other sources of data include the traffic archives maintained by CAIDA (the Cooperative Association for Internet Data Analysis),[117] the NLANR (National Laboratory for Applied Network Research),[119] and the ACM (Association for Computing Machinery).[116]

Average Packet Loss

Various packet loss metrics can be studied. For example, the average loss rate gives a general measure of network congestion, while loss patterns and correlation give insights into the dynamics of the network.

The reported measurements of average packet loss rate show a range of conditions. For example, measurements of TCP/IP traffic taken by Paxson in 1994 and 1995 show that 30% to 70% of flows, depending on path taken and date, showed no packet loss, but of those flows that did show loss, the average loss ranged from 3% to 17% (these results are summarized in Table 2.1). Data from Bolot, using 64-kilobit PCM-encoded audio, shows similar patterns, with loss rates between 4% and 16% depending on time of day, although this data also dates from 1995. More recent results from Yajnik et al., taken using simulated audio traffic in 1997–1998, show lower loss rates of 1.38% to 11.03%. Handley's results—two sets of approximately 3.5 million packets of data and reception report statistics for multicast video sessions in May and September 1996—show loss averaged over five-second intervals varying between 0% and 100%, depending on receiver location and time of day. A sample for one particular receiver during a ten-hour period on May 29, 1996, plotted in Figure 2.5, shows the average loss rate, sampled over five-second intervals, varying between 0% and 20%.

Figure 2.5. Loss Rate Distribution versus Time[122]

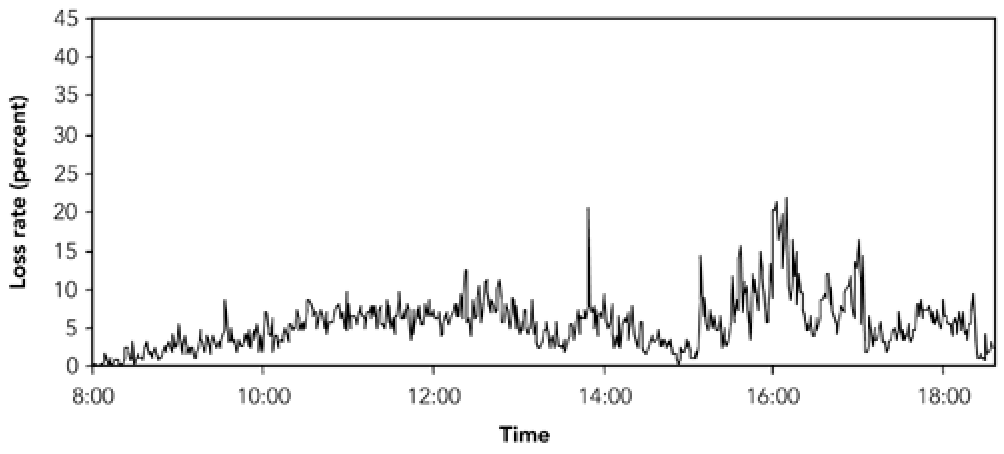
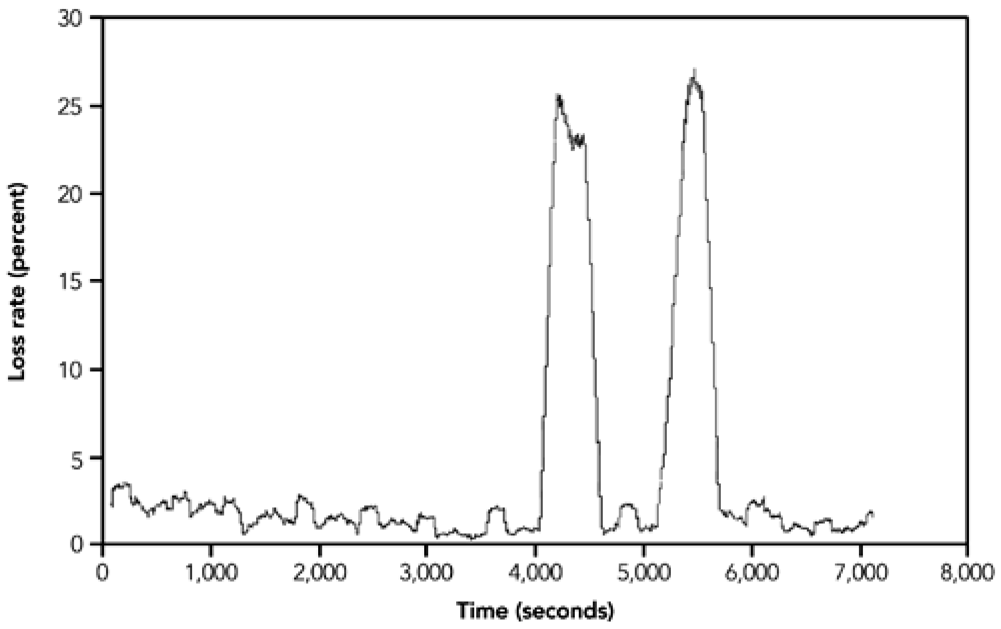


Table 2.1. Packet Loss Rates for Various Regions[95]

	Fraction of Flows Showing No Loss		Average Loss Rate for Flows with Loss	
Region	**Dec. 1994**	**Dec. 1995**	**Dec. 1994**	**Dec. 1995**
Within Europe	48%	58%	5.3%	5.9%
Within U.S.	66%	69%	3.6%	4.4%
U.S. to Europe	40%	31%	9.8%	16.9%
Europe to U.S.	35%	52%	4.9%	6.0%

The observed average loss rate is not necessarily constant, nor smoothly varying. The sample shows a loss rate that, in general, changes relatively smoothly, although at some points a sudden change occurs. Another example, from Yajnik et al., is shown in Figure 2.6. This case shows a much more dramatic change in loss rate: After a slow decline from 2.5% to 1% over the course of one hour, the loss rate jumps to 25% for 10 minutes before returning to normal—a process that repeats a few minutes later.

Figure 2.6. Loss Rate Distribution versus Time (Adapted from M. Yajnik, S. Moon, J. Kurose, and D. Towsley, "Measurement and Modeling of the Temporal Dependence in Packet Loss," Technical Report 98-78, Department of Computer Science, University of Massachusetts, Amherst, 1998. © 1999 IEEE.)



How do these loss rates compare with current networks? The conventional wisdom at the time of this writing is that it's possible to engineer the network backbone such that packet loss never occurs, so one might expect recent data to illustrate this. To some extent this is true; however, even if it's possible to engineer part of the network to be loss free, that possibility does not imply that the entire network will behave in the same way. Many network paths show loss today, even if that loss represents a relatively small fraction of packets.

The Internet Weather Report,[120] a monthly survey of the loss rate measured in a range of routes across the Internet, showed average packet loss rates within the United States ranging from 0% to 16%, depending on the ISP, as of May 2001. The monthly average loss rate is about 2% in the United States, but for the Internet as a whole, the average is slightly higher, at about 3%.

What should we learn from this? Even though some parts of the network are well engineered, others can show significant loss. Remember, as Table 2.1 shows, 70% of network paths within the United States had no packet loss in 1995, yet the average loss rate for the others was almost 5%, a rate that is sufficient to cause significant degradation in audio/video quality.

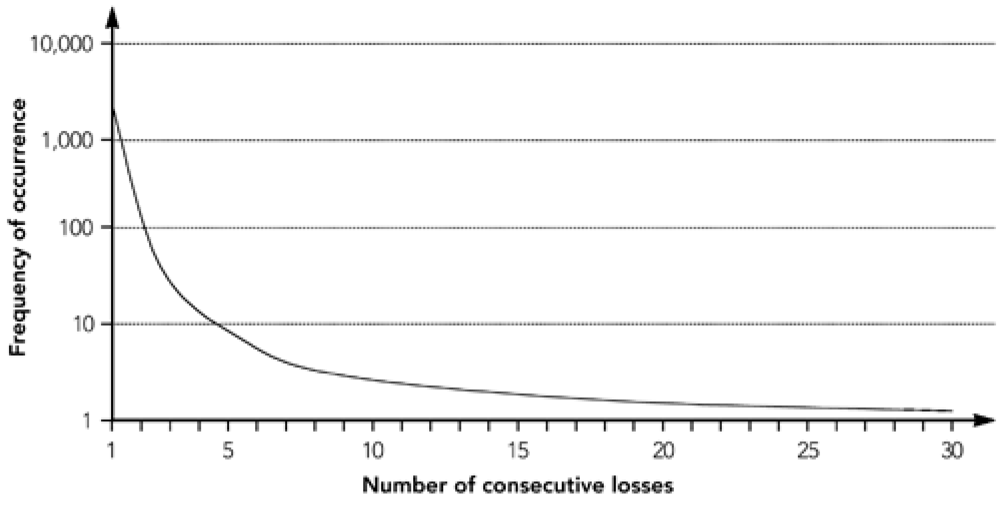
Packet Loss Patterns

In addition to looking at changes in the average loss rate, it is instructive to consider the short-term patterns of loss. If we aim to correct for packet loss, it helps to know if lost packets are randomly distributed within a media flow, or if losses occur in bursts.

If packet losses are distributed uniformly in time, we should expect that the probability that a particular packet is lost is the same as the probability that the previous packet was lost. This would mean that packet losses are most often isolated events, a desirable outcome because a single loss is easier to recover from than a burst. Unfortunately, however, the probability that a particular packet is lost often increases if the previous packet was also lost. That is, losses tend to occur back-to-back. Measurements by Vern Paxson have shown, in some cases, a five- to tenfold increase in loss probability for a particular packet if the previous packet was lost, clearly implying that packet loss is not uniformly distributed in time.

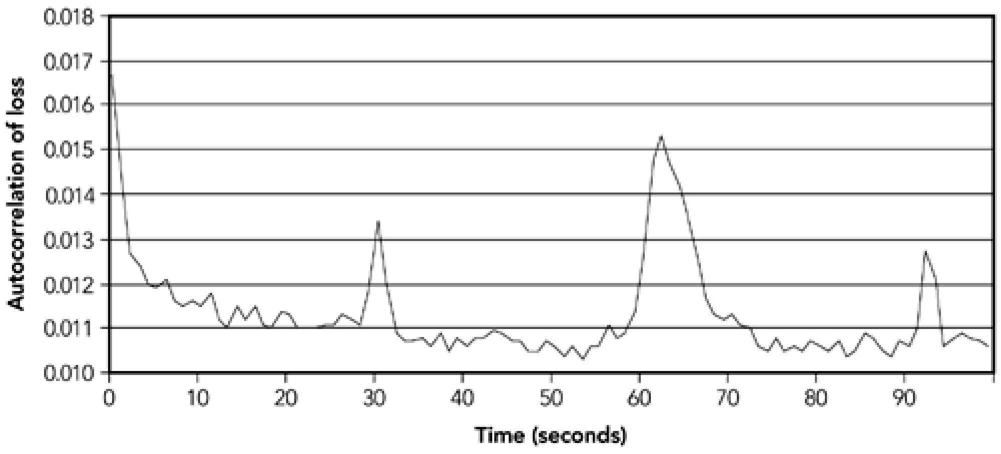
Several other studies—for example, measurements collected by Bolot in 1995, by Handley and Yajnik et al. in 1996, and by me in 1999—have confirmed the observation that packet loss probabilities are not independent. In all cases, these studies show that the vast majority of losses are of single packets, constituting about 90% of loss events. The probability of longer bursts of loss decreases as illustrated in Figure 2.7; it is clear that longer bursts occur more frequently than they would if losses were independent.

Figure 2.7. Frequency Distribution of the Number of Consecutively Lost Packets



Observed loss patterns also show significant periodicity in some cases. For example, Handley has reported bursts of loss occurring approximately every 30 seconds on measurements taken in 1996 (see Figure 2.8), and similar problems were reported in April 2001.[121] Such reports are not universal, and many traces show no such effect. It is conjectured that the periodicity is due to the overloading of certain systems by routing updates, but this is not certain.

Figure 2.8. Autocorrelation of Packet Loss



Packet Duplication

If packets can be lost in the network, can they be duplicated? Perhaps surprisingly, the answer is yes! It is possible for a source to send a single packet, yet for the receiver to get multiple copies of that packet. The most likely reason for duplication is faults in routing/switching elements within the network; duplication should not happen in normal operation.

How common is duplication? Paxson's measurements revealing the tendency of back-to-back packet losses also showed a small amount of packet duplication. In measuring 20,000 flows, he found 66 duplicate packets, but he also noted, "We have observed traces . . . in which more than 10% of the packets were replicated. The problem was traced to an improperly configured bridging device."[95] A trace that I took in August 1999 shows 131 duplicates from approximately 1.25 million packets.

Packet duplication should not cause problems, as long as applications are aware of the issue and discard duplicates (RTP contains a sequence number for this purpose). Frequent packet duplication wastes bandwidth and is a sign that the network contains badly configured or faulty equipment.

Packet Corruption

Packets can be corrupted, as well as lost or duplicated. Each IP packet contains a checksum that protects the integrity of the packet header, but the payload is not protected at the IP layer. However, the link layer may provide a checksum, and both TCP and UDP enable a checksum of the entire packet. In theory, these protocols will detect most corrupted packets and cause them to be dropped before they reach the application.

Few statistics on the frequency of packet corruption have been reported. Stone[103] quotes Paxson's observation that approximately one in 7,500 packets failed their TCP or UDP checksum, indicating corruption of the packet. Other measurements in that work show average checksum failure rates ranging from one in 1,100 to one in 31,900 packets. Note that this result is for a wired network; wireless networks can be expected to have significantly different properties because corruption due to radio interference may be more severe than that due to noise in wires.

When the checksum fails, the packet is assumed to be corrupted and is discarded. The application does not see corrupted packets; they appear to be lost. Corruption is visible as a small increase in the measured loss rate.

In some cases it may be desirable for an application to receive corrupt packets, or to get an explicit indication that the packets are corrupted. UDP provides a means of disabling the checksum for these cases. Chapter 8, Error Concealment, and Chapter 10, Congestion Control, discuss this topic in more detail.

Network Transit Time

How long does it take packets to transit the network? The answer depends on the route taken, and although a short path does take less time to transit than a long one, we need to be careful of the definition of *short*.

The components that affect transit time include the speed of the links, the number of routers through which a packet must pass, and the amount of queuing delay imposed by each router. A path that is short in physical distance may be long in the number of hops a packet must make, and often the queuing delay in the routers at each hop is the major factor. In network terms a short path is most often the one with the least hops, even if it covers a longer physical distance. Satellite links are the obvious exception, where distance introduces a significant radio propagation delay. Some measures of average round-trip time, taken in May 2001, are provided in Table 2.2 for comparison. Studies of telephone conversations subject to various round-trip delays have shown that people don't notice delays of less than approximately 300 milliseconds (ms). Although this is clearly a subjective measurement, dependent on the person and the task, the key point is that the measured network round-trip times are mostly within this limit. (The trip from London to Sydney is the exception, but the significant increase here is probably due to a satellite hop in the path.)

Table 2.2. Sample Round-Trip Time Measurements

Source	Destination	Hop Count	Round-Trip Time (ms)
Washington, DC	London	20	94

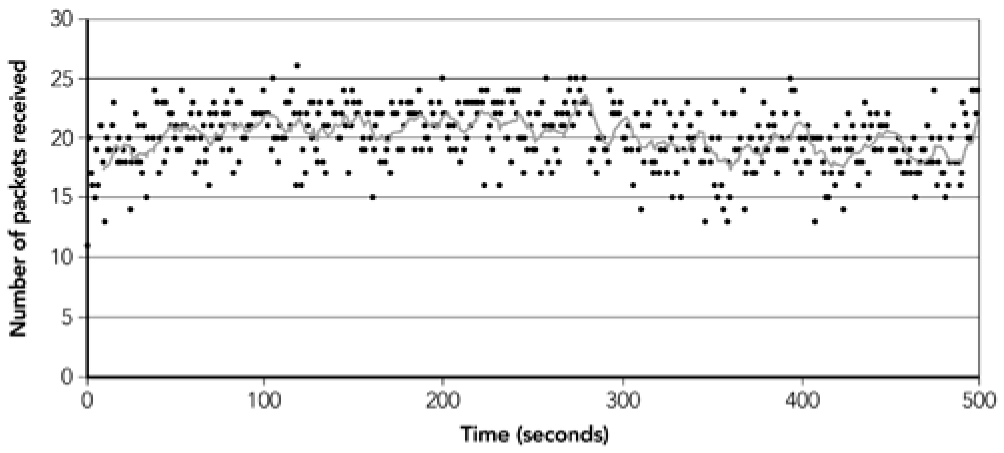
Table 2.2. Sample Round-Trip Time Measurements

Source	Destination	Hop Count	Round-Trip Time (ms)
Washington, DC	Berkeley, CA	21	88
London	Berkeley, CA	22	136
London	Sydney	28	540

Measures of the delay in itself are not very interesting because they so clearly depend on the location of the source and the destination. Of more interest is how the network transit time varies from packet to packet: A network that delivers packets with constant delay is clearly easier for an application to work with than one in which the transit time can vary, especially if that application has to present time-sensitive media.

A crude measure of the variation in transit time (*jitter*) is the arrival rate of packets. For example, Figure 2.9 shows the measured arrival rate for a stream sent at a constant rate; clearly the arrival rate varies significantly, indicating that the transit time across the network is not constant.

Figure 2.9. The Packet Arrival Rate



A better measure, which doesn't assume constant rate, is to find the transit time by measuring the difference between arrival time and departure time for each packet. Unfortunately, measuring the absolute transit time is difficult because it requires accurate synchronization between the clocks at source and destination, which is not often possible. As a result, most traces of network transit time include the clock offset, and it becomes impossible to study anything other than the variation in delay (because it is not possible to determine how much of the offset is due to unsynchronized clocks and how much is due to the network).

Some sample measurements of the variation in transit time, including the offset due to unsynchronized clocks, are presented in Figure 2.10 and Figure 2.11. I took these measurements in August 1999; similar measurements have been presented by Ramjee et al. (in 1994)[96] and by Moon et al.[90] Note the following:

- The slow downward slope of the measurements is due to clock skew between the source and the destination. One machine's clock is running slightly faster than the other's, leading to a gradual change in the perceived transit time.
- Several step changes in the average transit time, perhaps due to route changes in the network, can be observed.
- The transit time is not constant; rather it shows significant variation throughout the session.

Figure 2.10. Variation in Network Transit Time: 400-Second Sample

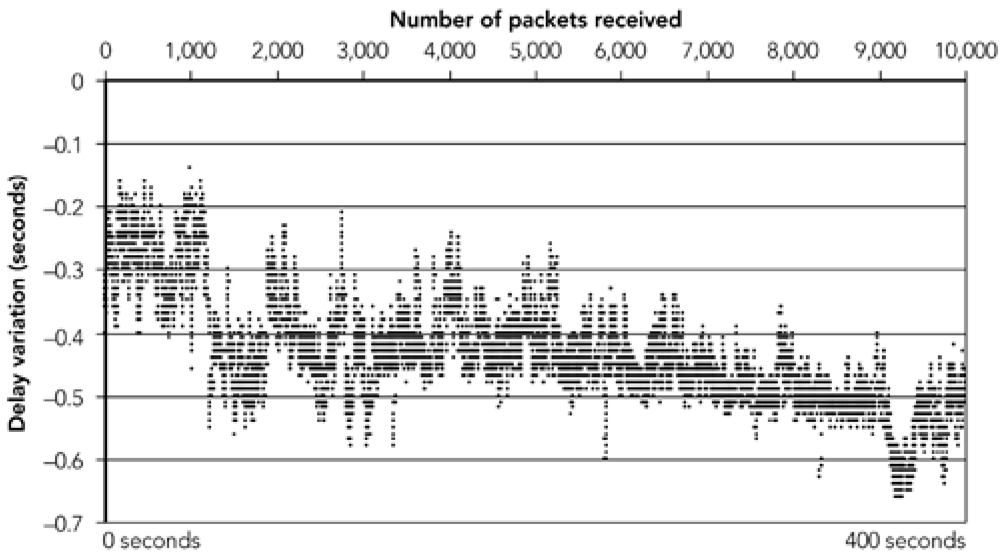
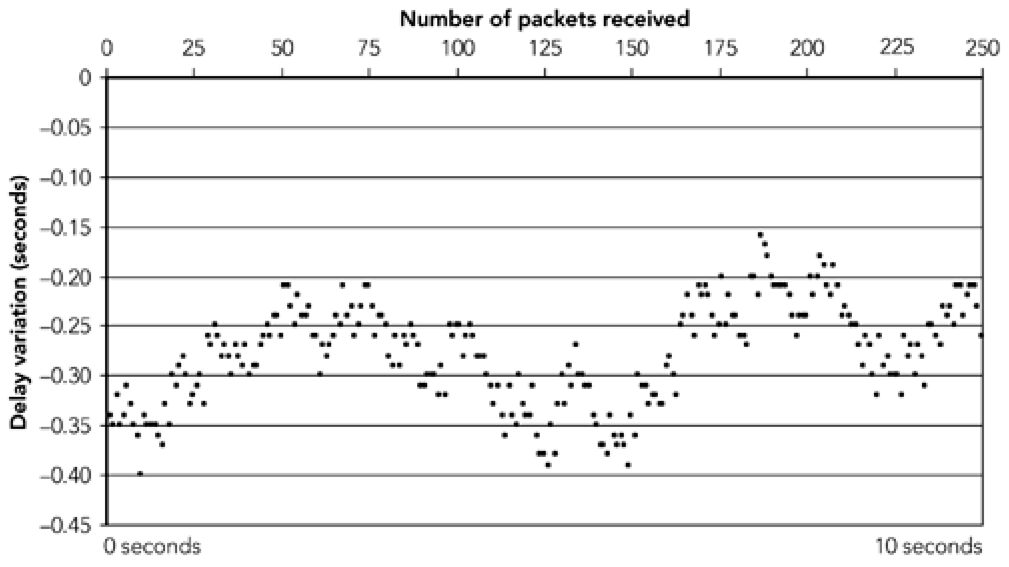


Figure 2.11. Variation in Network Transit Time: 10-Second Sample

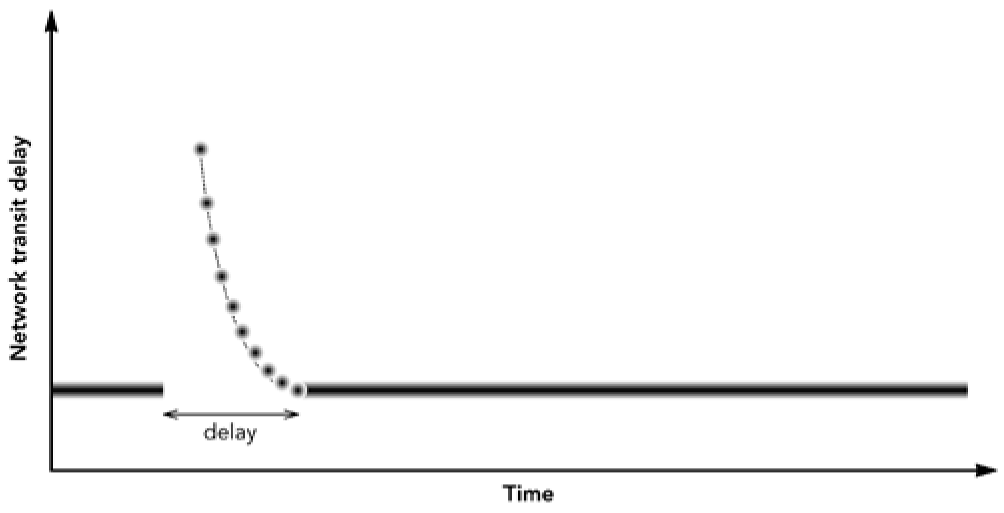


These are all problems that an application, or higher-layer protocol, must be prepared to handle and, if necessary, correct.

It is also possible for packets to be reordered within the network—for example, if the route changes and the new route is shorter. Paxson[95] observed that a total 2% of TCP data packets were delivered out of order, but that the fraction of out-of-order packets varied significantly between traces, with one trace showing 15% of packets being delivered out of order.

Another characteristic that may be observed is "spikes" in the network transit time, such as those shown in Figure 2.12. It is unclear whether these spikes are due to buffering within the network or in the sending system, but they present an interesting challenge if one is attempting to smooth the arrival time of a sequence of packets.

Figure 2.12. Spikes in Network Transit Time (Adapted from Moon et al., 1998).[91]



Finally, the network transit time can show periodicity (as noted, for example, in Moon et al. 1998[89]), although this appears to be a secondary effect. We expect this periodicity to have causes similar to those of the loss periodicity noted earlier, except that the events are less severe and cause queue buildup in the routers, rather than queue overflow and hence packet loss.

Acceptable Packet Sizes

The IP layer can accept variably sized packets, up to 65,535 octets in length, or to a limit determined by the maximum transmission unit (MTU) of the links traversed. The MTU is the size of the largest packet that can be accommodated by a link. A common value is 1,500 octets, which is the largest packet an Ethernet can deliver. Many applications assume that they can send packets up to this size, but some links have a lower MTU. For example, it is not unusual for dial-up modem links to have an MTU of 576 octets.

In most cases the bottleneck link is adjacent to either sender or receiver. Virtually all backbone network links have an MTU of 1,500 octets or more.

IPv4 supports fragmentation, in which a large packet is split into smaller fragments when it exceeds the MTU of a link. It is generally a bad idea to rely on this, though, because the loss of any one fragment will make it impossible for the receiver to reconstruct the packet. The resulting loss multiplier effect is something we want to avoid.

In virtually all cases, audio packets fall within the network MTU. Video frames are larger, and applications should split them into multiple packets for transport so that each packet fits within the MTU of the network.

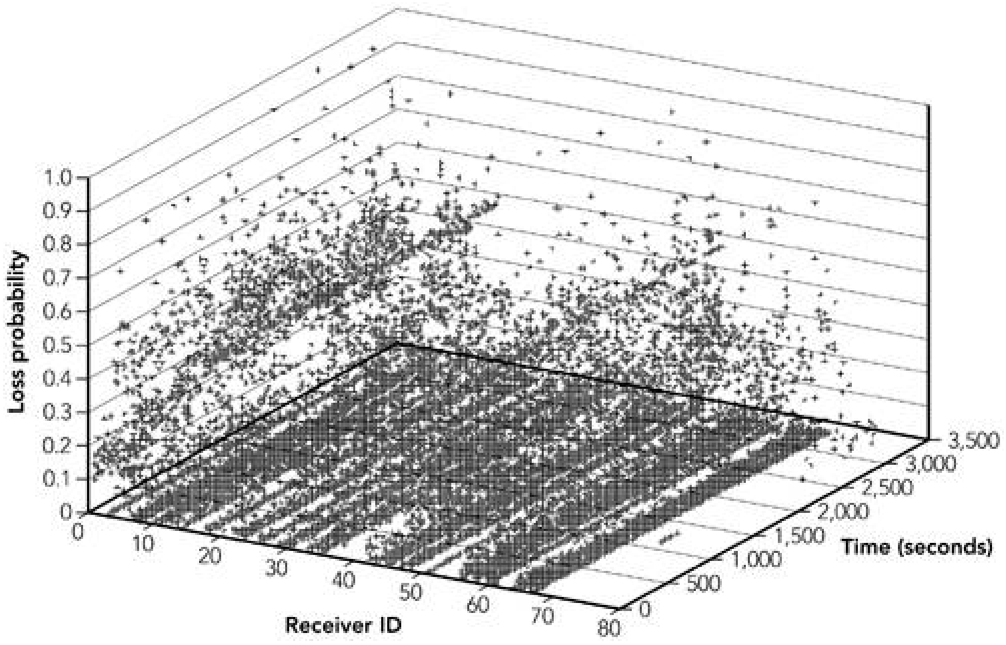
Effects of Multicast

IP multicast enables a sender to transmit data to many receivers at the same time. It has the useful property that the network creates copies of the packet as needed, such that only one copy of the packet traverses each link. IP multicast provides for extremely efficient group communication, provided it is supported by the network, making the cost of sending data to a group of receivers independent of the size of the group.

Support for multicast is an optional, and relatively new, feature of IP networks. At the time of this writing, it is relatively widely deployed in research and educational environments and in the network backbone, but it is uncommon in many commercial settings and service providers.

Sending to a group means that more things can go wrong: Reception quality is no longer affected by a single path through the network, but by the path from the source to each individual receiver. The defining factor in the measurement of loss and delay characteristics for a multicast session is *heterogeneity*. Figure 2.13 illustrates this concept, showing the average loss rate seen at each receiver in a multicast session measured by me.

Figure 2.13. Loss Rates in a Multicast Session



Multicast does not change the fundamental causes of loss or delay in the network. Rather it enables each receiver to experience those effects while the source transmits only a single copy of each packet. The heterogeneity of the network makes it difficult for the source to satisfy all receivers: Sending too fast for some and too slow for others is common. We will discuss these issues further in later chapters. For now it's sufficient to note that multicast adds yet more heterogeneity to the system.

Effects of Network Technologies

The measurements that have been presented so far have been of the public, wide area, Internet. Many applications will operate in this environment, but a significant number will be used in private intranets, on wireless networks, or on networks that support enhanced quality of service. How do such considerations affect the performance of the IP layer?

Many private IP networks (often referred to as *intranets*) have characteristics very similar to those of the public Internet: The traffic mix is often very similar, and many intranets cover wide areas with varying link speeds and congestion levels. In such cases it is highly likely that the performance will be similar to that measured on the public Internet. If, however, the network is engineered specifically for real-time multimedia traffic, it is possible to avoid many of the problems that have been discussed and to build an IP network that has no packet loss and minimal jitter.

Some networks support enhanced quality of service (QoS) using, for example, integrated services/RSVP[11] or differentiated services.[24] The use of enhanced QoS may reduce the need for packet loss and/or jitter resilience in an application because it provides a strong guarantee that certain performance bounds are met. Note, however, that in many cases the guarantee provided by the QoS scheme is statistical in nature, and often it does not completely eliminate packet loss, or perhaps more commonly, variations in transit time.

Significant performance differences can be observed in wireless networks. For example, cellular networks can exhibit significant variability in their performance over short time frames, including noncongestive loss, bursts of loss, and high bit error rates. In addition, some cellular systems have high latency because they use interleaving at the data link layer to hide bursts of loss or corruption.

The primary effect of different network technologies is to increase the heterogeneity of the network. If you are designing an application to work over a restricted subset of these technologies, you may be able to leverage the facilities of the underlying network to improve the quality of the connection that your application sees. In other cases the underlying network may impose additional challenges on the designer of a robust application.

A wise application developer will choose a robust design so that when an application moves from the network where it was originally envisioned to a new network, it will still operate correctly. The challenge in designing audiovisual applications to operate over IP is making them reliable in the face of network problems and unexpected conditions.

Conclusions about Measured Characteristics

Measuring, predicting, and modeling network behavior are complex problems with many subtleties. This discussion has only touched on the issues, but some important conclusions are evident.

The first point is that the network can and frequently does behave badly. An engineer who designs an application that expects all packets to arrive, and to do so in a timely manner, will have a surprise when that application is deployed on the Internet. Higher-layer protocols—such as TCP—can hide some of this badness, but there are *always* some aspects visible to the application.

Another important point to recognize is the *heterogeneity* in the network. Measurements taken at one point in the network will not represent what's happening at a different point, and even "unusual" events happen all the time. There were approximately 100 million systems on the Net at the end of 2000,[118] so even an event that occurs to less than 1% of hosts will affect many hundreds of thousands of machines. As an application designer, you need to be aware of this heterogeneity and its possible effects.

Despite this heterogeneity, an attempt to summarize the discussion of loss and loss patterns reveals several "typical" characteristics:

- Although some network paths may not exhibit loss, these paths are not usual in the public Internet. An application should be designed to cope with a small amount of packet loss—say, up to 5%.
- Isolated packet losses compose most observed loss events.
- The probability that a packet is lost is not uniform: Even though most losses are of isolated packets, bursts of consecutively lost packets are more common than can be explained by chance alone. Bursts of loss are typically short; an application that copes with two to three consecutively lost packets will suffice for most burst losses.
- Rarely, long bursts of loss occur. Outages of a significant fraction of a second, or perhaps even longer, are not unknown.
- Packet duplication is rare, but it can occur.
- Similarly, in rare cases packets can be corrupted. The over-whelming majority of these are detected by the TCP or UDP checksum (if enabled), and the packet is discarded before it reaches the application.

The characteristics of transit time variation can be summarized as follows:

- Transit time across the network is not uniform, and jitter is observed.
- The vast majority of jitter is reasonably bounded, but the tail of the distribution can be long.
- Although reordering is comparatively rare, packets can be reordered in transit. An application should not assume that the order in which packets are received corresponds to the order in which they were sent.

These are not universal rules, and for every one there is a network path that provides a counterexample.[71] They do, however, provide some idea of what we need to be aware of when designing higher-layer protocols and applications.

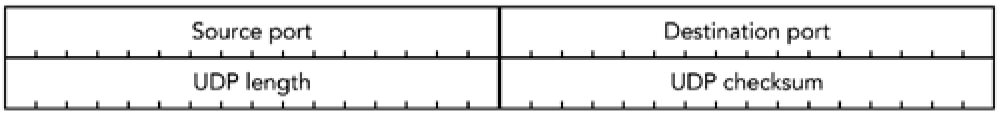
Effects of Transport Protocols

Thus far, our consideration of network characteristics has focused on IP. Of course, programmers almost never use the raw IP service. Instead, they build their applications on top of one of the higher-layer transport protocols, typically either UDP or TCP. These protocols provide additional features beyond those provided by IP. How do these added features affect the behavior of the network as seen by the application?

UDP/IP

The User Datagram Protocol (UDP) provides a minimal set of extensions to IP. The UDP header is shown in Figure 2.14. It comprises 64 bits of additional header representing source and destination port identifiers, a length field, and a checksum.

Figure 2.14. Format of a UDP Header



The source and destination ports identify the endpoints within the communicating hosts, allowing for multiplexing of different services onto different ports. Some services run on well-known ports; others use a port that is dynamically negotiated during call setup. The length field is redundant with that in the IP header. The checksum is used to detect corruption of the payload and is optional (it is set to zero for applications that have no use for a checksum).

Apart from the addition of ports and a checksum, UDP provides the raw IP service. It does not provide any enhanced reliability to the transport (although the checksum does allow for detection of payload errors that IP does not detect), nor does it affect the timing of packet delivery. An application using UDP provides data packets to the transport layer, which delivers them to a port on the destination machine (or to a group of machines if multicast is used). Those packets may be lost, delayed, or misordered in transit, exactly as observed for the raw IP service.

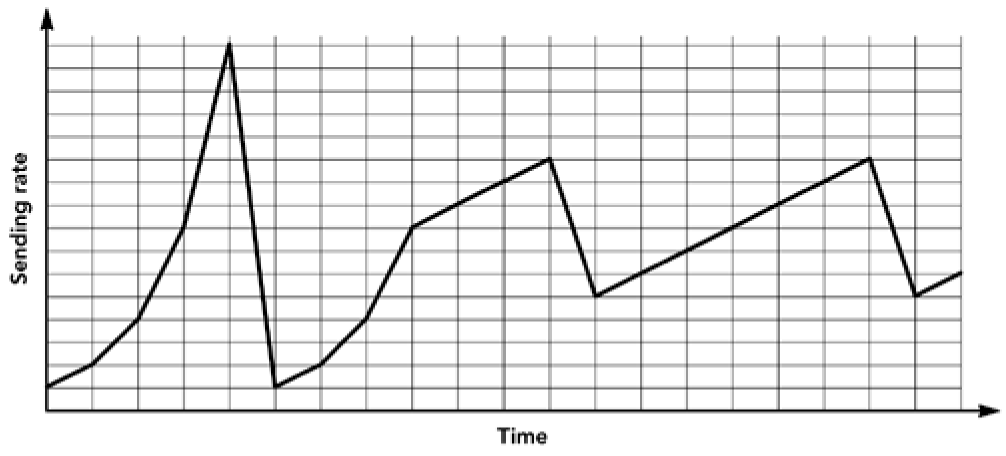
TCP/IP

The most common transport protocol on the Internet is TCP. Although UDP provides only a small set of additions to the IP service, TCP adds a significant amount of additional functionality: It abstracts the unreliable packet delivery service of IP to provide reliable, sequential delivery of a byte stream between ports on the source and a single destination host.

An application using TCP provides a stream of data to the transport layer, which fragments it for transmission in appropriately sized packets, and at a rate suitable for the network. Packets are acknowledged by the receiver, and those that are lost in transit are retransmitted by the source. When data arrives, it is buffered at the receiver so that it can be delivered in order. This process is transparent to the application, which simply sees a "pipe" of data flowing across the network.

As long as the application provides sufficient data, the TCP transport layer will increase its sending rate until the network exhibits packet loss. Packet loss is treated as a signal that the bandwidth of the bottleneck link has been exceeded and the connection should reduce its sending rate to match. Accordingly, TCP reduces its sending rate when loss occurs. This process continues, with TCP continually probing the sustainable transmission rate across the network; the result is a sending rate such as that illustrated in Figure 2.15.

Figure 2.15. Sample TCP Sending Rate



This combination of retransmission, buffering, and probing of available bandwidth has several effects:

- TCP transport is reliable, and if the connection remains open, all data will eventually be delivered. If the connection fails, the endpoints are made aware of the failure. This is a contrast to UDP, which offers the sender no information about the delivery status of the data.
- An application has little control over the timing of packet delivery because there is no fixed relation between the time at which the source sends data and the time at which that data is received. This variation differs from the variation in transit time exhibited by the raw IP service because the TCP layer must also account for retransmission and variation in the sending rate. The sender can tell if all outstanding data has been sent, which may enable it to estimate the average transmission rate.
- Bandwidth probing may cause short-term overloading of the bottleneck link, resulting in packet loss. When this overloading causes loss to the TCP stream, that stream will reduce its rate; however, it may also cause loss to other streams in the process.

Of course, there are subtleties to the behavior of TCP, and much has been written on this subject. There are also some features that this discussion has not touched upon, such as push mode and urgent delivery, but these do not affect the basic behavior. What is important for our purpose is to note the fundamental difference between TCP and UDP: the trade-off between reliability (TCP) and timeliness (UDP).

Requirements for Audio/Video Transport in Packet Networks

So far, this chapter has explored the characteristics of IP networks in some detail, and has looked briefly at the behavior of the transport protocols layered above them. We can now relate this discussion to real-time audio and video transport, consider the requirements for delivery of media streams over an IP network, and determine how well the network meets those requirements.

When we describe media as *real-time*, we mean simply that the receiver is playing out the media stream as it is received, rather than simply storing the complete stream in a file for later play-back. In the ideal case, playout at the receiver is immediate and synchronous, although in practice some unavoidable transmission delay is imposed by the network.

The primary requirement that real-time media places on the transport protocol is for predictable variation in network transit time. Consider, for example, an IP telephony system transporting encoded voice in 20-millisecond frames: The source will transmit one packet every 20 milliseconds, and ideally we would like those to arrive with the same spacing so that the speech they contain can be played out immediately. Some variation in transit time can be accommodated by the insertion of additional buffering delay at the receiver, but this is possible only if that variation can be characterized and the receiver can adapt to match the variation (this process is described in detail in Chapter 6, Media Capture, Playout, and Timing).

A lesser requirement is reliable delivery of all packets by the network. Clearly, reliable delivery is desirable, but many audio and video applications can tolerate some loss: In our IP telephony example, loss of a single packet will result in a dropout of one-fiftieth of a second, which, with suitable error concealment, is barely noticeable. Because of the time-varying nature of media streams, some loss is usually acceptable because its effects are quickly corrected by the arrival of new data. The amount of loss that is acceptable depends on the application, the encoding method used, and the pattern of loss. Chapter 8, Error Concealment, and Chapter 9, Error Correction, discuss loss tolerance.

These requirements drive the choice of transport protocol. It should be clear that TCP/IP is not appropriate because it favors reliability over timeliness, and our applications require timely delivery. A UDP/IP-based transport should be suitable, provided that the variation in transit time of the network can be characterized and loss rates are acceptable.

The standard Real-time Transport Protocol (RTP) builds on UDP/IP, and provides timing recovery and loss detection, to enable the development of robust systems. RTP and associated standards will be discussed in extensive detail in the remainder of this book.

Despite TCP's limitations for real-time applications, some audio/video applications use it for their transport. Such applications attempt to estimate the average throughput of the TCP connection and adapt their send rate to match. This approach can be made to work when tight end-to-end delay bounds are not required and an application has several seconds worth of buffering to cope with the variation in delivery time caused by TCP retransmission and congestion control. It does not work reliably for interactive applications, which need short end-to-end delay, because the variation in transit time caused by TCP is too great.

The primary rationale for the use of TCP/IP transport is that many firewalls pass TCP connections but block UDP. This situation is changing rapidly, as RTP-based systems become more prevalent and firewalls smarter. I strongly recommend that new applications be based on RTP-over-UDP/IP. RTP provides for higher quality by enabling applications to adapt in a way that is appropriate for real-time media, and by promoting interoperability (because it is an open standard).

Benefits of Packet-Based Audio/Video

At this stage you may be wondering why anyone would consider a packet-based audio or video application over an IP network. Such a network clearly poses challenges to the reliable delivery of real-time media streams. Although these challenges are real, an IP network has some distinct advantages that lead to the potential for significant gains in efficiency and flexibility, which can outweigh the disadvantages.

The primary advantage of using IP as the bearer service for realtime audio and video is that it can provide a unified and converged network. The same network can be used for voice, music, and video as for e-mail, Web access, file and document transfers, games, and so on. The result can be significant savings in infrastructure, deployment, support, and management costs.

A unified packet network enables statistical multiplexing of traffic. For example, voice activity detection can be used to prevent packet voice applications from transmitting during silence periods, and traffic using TCP/IP as its transport will adapt to match the variation in available capacity. Provided that care is taken to design audio and video applications to tolerate the effects of this adaptation, the applications will not be adversely affected. Thus we can achieve much higher link utilization, which is important in resource-limited systems.

Another important benefit is provided by IP multicast, which allows inexpensive delivery of data to a potentially large group of receivers. IP multicast enables affordable Internet radio and television, and other group communication services, by making the cost of delivery independent of the size of the audience.

The final, and perhaps most compelling, case for audio and video over IP is that IP enables new services. Convergence allows rich interaction between real-time audio/video and other applications, enabling us to develop systems that would not previously have been possible.

Summary

The properties of an IP network are significantly different from those of traditional telephony, audio, or video distribution networks. When designing applications that work over IP, you need to be aware of these unique characteristics, and make your system robust to their effects.

The remainder of this book will describe an architecture for such systems, explaining RTP and its model for timing recovery and lip synchronization, error correction and concealment, congestion control, header compression, multiplexing and tunneling, and security.

Chapter 3. The Real-Time Transport Protocol

This chapter describes the design of the RTP framework starting with the philosophy and background of the design, gives an overview of the applicable standards, and explains how those standards interrelate. It concludes with a discussion of possible future directions for the development of those standards.

Fundamental Design Philosophies of RTP

The challenge facing the designers of RTP was to build a mechanism for robust, real-time media delivery above an unreliable transport layer. They achieved this goal with a design that follows the twin philosophies of *application-level framing* and the *end-to-end principle*.

Application-Level Framing

The concepts behind application-level framing were first elucidated by Clark and Tennenhouse[65] in 1990. Their central thesis is that only the application has sufficient knowledge of its data to make an informed decision about how that data should be transported. The implication is that a transport protocol should accept data in application-meaningful units (application data units, ADUs) and expose the details of their delivery as much as possible so that the application can make an appropriate response if an error occurs. The application partners with the transport, cooperating to achieve reliable delivery.

Application-level framing comes from the recognition that there are many ways in which an application can recover from network problems, and that the correct approach depends on both the application and the scenario in which it is being used. In some cases it is necessary to retransmit an exact copy of the lost data. In others, a lower-fidelity copy may be used, or the data may have been superseded, so the replacement is different from the original. Alternatively, the loss can be ignored if the data was of only transient interest. These choices are possible only if the application interacts closely with the transport.

The goal of application-level framing is somewhat at odds with the design of TCP, which hides the lossy nature of the underlying IP network to achieve reliable delivery at the expense of timeliness. It does, however, fit well with UDP-based transport and with the characteristics of real-time media. As noted in Chapter 2, Voice and Video Communication over Packet Networks, real-time audio and visual media is often loss tolerant but has strict timing bounds. By using application-level framing with UDP-based transport, we are able to accept losses where necessary, but we also have the flexibility to use the full spectrum of recovery techniques, such as retransmission and forward error correction, where appropriate.

These techniques give an application great flexibility to react to network problems in a suitable manner, rather than being constrained by the dictates of a single transport layer.

A network that is designed according to the principles of application-level framing should not be specific to a particular application. Rather it should expose the limitations of a generic transport layer so that the application can cooperate with the network in achieving the best possible delivery. Application-level framing implies a weakening of the strict layers defined by the OSI reference model. It is a pragmatic approach, acknowledging the importance of layering, but accepting the need to expose some details of the lower layers.

The philosophy of application-level framing implies smart, network-aware applications that are capable of reacting to problems.

The End-to-End Principle

The other design philosophy adopted by RTP is the end-to-end principle.[70] It is one of two approaches to designing a system that must communicate reliably across a network. In one approach, the system can pass responsibility for the correct delivery of data along with that data, thus ensuring reliability hop by hop. In the other approach, the responsibility for data can remain with the endpoints, ensuring reliability end-to-end even if the individual hops are unreliable. It is this second end-to-end approach that permeates the design of the Internet, with both TCP and RTP following the end-to-end principle.

The main consequence of the end-to-end principle is that intelligence tends to bubble up toward the top of the protocol stack. If the systems that make up the network path never take responsibility for the data, they can be simple and do not need to be robust. They may discard data that they cannot deliver, because the endpoints will recover without their help. The end-to-end principle implies that intelligence is at the endpoints, not within the network.

The result is a design that implies smart, network-aware endpoints and a dumb network. This design is well suited to the Internet—perhaps the ultimate dumb network—but does require significant work on the part of an application designer. It is also a design unlike that of many other networks. The traditional telephone network, for example, adopts the model of an intelligent network and dumb endpoints, and the MPEG transport model allows dumb receivers with smart senders. This difference in design changes the style of the applications, placing greater emphasis on receiver design and making sender and receiver more equal partners in the transmission.

Achieving Flexibility

The RTP framework was designed to be sufficient for many scenarios, with little additional protocol support. In large part this design was based around the lightweight sessions model for video conferencing.[76] In this scenario the RTP control protocol provides all the necessary session management functions, and all that is needed to join the session is the IP address and the mapping from media definitions to RTP payload type identifiers. This model also works well for one to many scenarios—for example, Internet radio, where the feedback provided by the control protocol gives the source an estimate of the audience size and reception quality.

For unicast voice telephony, some have argued that RTP provides unnecessary features, and is heavyweight and inefficient for highly compressed voice frames. In practice, with the use of header compression this is not a strong argument, and the features provided enable extension to multimedia and multiparty sessions with ease. Still others—for example, the digital cinema community—have argued that RTP is underspecified for their needs and should include stronger quality-of-service and security support, more detailed statistics, and so on.

The strength of RTP is that it provides a unifying framework for real-time audio/video transport, satisfying most applications directly, yet being malleable for those applications that stretch its limits.

Standard Elements of RTP

The primary standard for audio/video transport in IP networks is the Real-time Transport Protocol (RTP), along with associated profiles and payload formats. RTP was developed by the Audio/Video Transport working group of the Internet Engineering Task Force (IETF), and it has since been adopted by the International Telecommunications Union (ITU) as part of its H.323 series of recommendations, and by several other standards organizations.

RTP provides a framework for the transport of real-time media and needs to be profiled for particular uses before it is complete. The RTP profile for audio and video conferences with minimal control was standardized along with RTP, and several more profiles are under development. Each profile is accompanied by several payload format specifications, each of which describes the transport of a particular media format.

The RTP Specification

RTP was published as an IETF proposed standard (RFC 1889) in January 1996,[6] and its revision for draft standard status is almost complete.[50] The first revision of ITU recommendation H.323 included a verbatim copy of the RTP specification; later revisions reference the current IETF standard.

> In the IETF standards process,[8] a specification undergoes a development cycle in which multiple *Internet drafts* are produced as the details of the design are worked out. When the design is complete, it is published as a *proposed standard* RFC. A proposed standard is generally considered stable, with all known design issues worked out, and suitable for implementation. If that proposed standard proves useful, and if there are independent and interoperable implementations of each feature of that standard, it can then be advanced to *draft standard* status (possibly involving changes to correct any problems found in the proposed standard). Finally, after extensive experience, it may be published as a full *standard* RFC. Advancement beyond proposed standard status is a significant hurdle that many protocols never achieve.

RTP typically sits on top of UDP/IP transport, enhancing that transport with loss detection and reception quality reporting, provision for timing recovery and synchronization, payload and source identification, and marking of significant events within the media stream. Most implementations of RTP are part of an application or library that is layered above the UDP/IP sockets interface provided by the operating system. This is not the only possible design, though, and nothing in the RTP protocol requires UDP or IP. For example, some implementations layer RTP above TCP/IP, and others use RTP on non-IP networks, such as Asynchronous Transfer Mode (ATM) networks.

There are two parts to RTP: the *data transfer protocol* and an associated *control protocol*. The RTP data transfer protocol manages delivery of real-time data, such as audio and video, between end systems. It defines an additional level of framing for the media payload, incorporating a sequence number for loss detection, timestamp to enable timing recovery, payload type and source identifiers, and a marker for significant events within the media stream. Also specified are rules for timestamp and sequence number usage, although these rules are somewhat dependent on the profile and payload format in use, and for multiplexing multiple streams within a session. The RTP data transfer protocol is discussed further in Chapter 4.

The RTP control protocol (RTCP) provides reception quality feedback, participant identification, and synchronization between media streams. RTCP runs alongside RTP and provides periodic reporting of this information. Although data packets are typically sent every few milliseconds, the control protocol operates on the scale of seconds. The information sent in RTCP is necessary for synchronization between media streams—for example, for lip synchronization between audio and video—and can be useful for adapting the transmission according to reception quality feedback, and for identifying the participants. The RTP control protocol is discussed further in Chapter 5.

RTP supports the notion of *mixers* and *translators*, middle boxes that can operate on the media as it flows between endpoints. These may be used to translate an RTP session between different lower-layer protocols—for example, bridging between participants on IPv4 and IPv6 networks, or bringing a unicast-only participant into a multicast group. They can also adapt a media stream in some way—for example, transcoding the data format to reduce the bandwidth, or mixing multiple streams together.

It is hard to place RTP in the OSI reference model. It performs many of the tasks typically assigned to a transport-layer protocol, yet it is not a complete transport in itself. RTP also performs some tasks of the session layer (spanning disparate transport connections and managing participant identification in a transport-neutral manner) and of the presentation layer (defining standard representations for media data).

RTP Profiles

It is important to be aware of the limits of the RTP protocol specification because it is deliberately incomplete in two ways. First, the standard does not specify algorithms for media playout and timing regeneration, synchronization between media streams, error concealment and correction, or congestion control. These are properly the province of the application designer, and because different applications have different needs, it would be foolish for the standard to mandate a single behavior. It does, of course, provide the necessary information for these algorithms to operate when they have been specified. Later chapters will discuss application design and the trade-offs inherent in providing these features.

Second, some details of the transport are left open to modification by profiles and payload format definitions. These include features such as the resolution of the timestamps, marking of interesting events within a media stream, and use of the payload type field. The features that can be specified by RTP profiles include the following:

- Mapping between the payload type identifier in the RTP header and the payload format specifications (which describe how individual media codecs are to be used with RTP). Each profile will reference multiple payload formats and may indicate how particular signaling protocols (for example, SDP[15]) are used to describe the mapping.
- The size of the payload type identifier field in the RTP header, and the number of bits used to mark events of interest within a media stream.
- Additions to the fixed RTP data transfer protocol header, if that header proves insufficient for a particular class of application.
- The reporting interval for the RTP control protocol—for example, to make feedback more timely at the expense of additional overhead.
- Limitations on the RTCP packet types that are to be used, if some of the information provided is not useful to that class of applications. In addition, a profile may define extensions to RTCP to report additional information.
- Additional security mechanisms—for example, new encryption and authentication algorithms.
- Mapping of RTP and RTCP onto lower-layer transport protocols.

At the time of this writing, there is a single RTP profile: the RTP profile for audio and video conferences with minimal control. This profile was published as a proposed standard (RFC 1890) along with the RTP specification in January 1996,[7] and its revision for draft standard status is almost complete.[49] Several new profiles are under development. Those likely to be available soon include profiles specifying additional security,[55] as well as feedback and repair mechanisms.[44]

RTP Payload Formats

The final piece of the RTP framework is the payload formats, defining how particular media types are transported within RTP. Payload formats are referenced by RTP profiles, and they may also define certain properties of the RTP data transfer protocol.

The relation between an RTP payload format and profile is primarily one of namespace, although the profile may also specify some general behavior for payload formats. The namespace relates the payload type identifier in the RTP packets to the payload format specifications, allowing an application to relate the data to a particular media codec. In some cases the mapping between payload type and payload format is static; in others the mapping is dynamic via an out-of-band control protocol. For example, the RTP profile for audio and video conferences with minimal control[7] defines a set of static payload type assignments, and a mechanism for mapping between a MIME type identifying a payload format, and a payload type identifier using the Session Description Protocol (SDP).

The relation between a payload format and the RTP data transfer protocol is twofold: A payload format will specify the use of certain RTP header fields, and it may define an additional payload header. The output produced by a media codec is translated into a series of RTP data packets—some parts mapping onto the RTP header, some into a payload header, and most into the payload data. The complexity of this mapping process depends on the design of the codec and on the degree of error resilience required. In some cases the mapping is simple; in others it is more complex.

At its simplest, a payload format defines only the mapping between media clock and RTP timestamp, and mandates that each frame of codec output is placed directly into an RTP packet for transport. An example of this is the payload format for G.722.1 audio.[36] Unfortunately, this is not sufficient in many cases because many codecs were developed without reference to the needs of a packet delivery system and need to be adapted to this environment. Others were designed for packet networks but require additional header information. In these cases the payload format specification defines an additional payload header, to be placed after the main RTP header, and rules for generation of that header.

Many payload formats have been defined, matching the diversity of codecs that are in use today, and many more are under development. At the time of this writing, the following audio payload formats are in common use, although this is by no means an exhaustive list: G.711, G.723.1, G.726, G.728, G.729, GSM, QCELP, MP3, and DTMF.[30, 34, 38, 49] The commonly used video payload formats include H.261, H.263, and MPEG.[9, 12, 22]

There are also payload formats that specify error correction schemes. For example, RFC 2198 defines an audio redundancy encoding scheme,[10] and RFC 2733 defines a generic forward error correction scheme based on parity coding.[32] In these payload formats there is an additional layer of indirection, the codec output is mapped onto RTP packets, and those packets themselves are mapped to produce an error-resilient transport. Error correction is discussed in more detail in Chapter 9, Error Correction.

Optional Elements

Two optional pieces of the RTP framework are worth mentioning at this stage: header compression and multiplexing.

Header compression is a means by which the overhead of the RTP and UDP/IP headers can be reduced on a per-link basis. It is used on bandwidth-constrained links—for example, cellular and dial-up links—and can reduce the 40-byte combination of RTP/UDP/IP headers to 2 bytes, at the expense of additional processing by the systems on the ends of the compressed link. Header compression is discussed further in Chapter 11.

Multiplexing is the means by which multiple related RTP sessions are combined into one. Once again, the motivation is to reduce overheads, except this time the procedure operates end-to-end. Multiplexing is discussed in Chapter 12, Multiplexing and Tunneling.

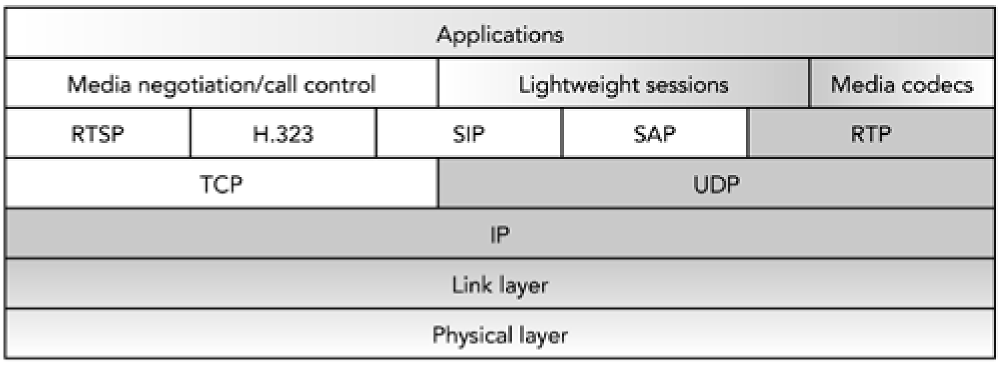
Both header compression and multiplexing can be considered to be part of the RTP framework. Unlike the profiles and payload formats, they are clearly special-purpose, optional parts of the system, and many implementations don't use either feature.

Related Standards

In addition to the RTP framework, a complete system will typically require the use of various other protocols and standards for call setup and control, session description, multiparty communication, and signaling quality-of-service requirements. Although this book does not cover the use of such protocols in detail, in this section it provides pointers to their specification and further reading.

The complete multimedia protocol stack is illustrated in Figure 3.1, showing the relationship between the RTP framework and the supporting setup and control protocols. The protocols and functions that are discussed in this book are highlighted.

Figure 3.1. The Multimedia Protocol Stack



Call Setup and Control

Various call setup, control, and advertisement protocols can be used to start an RTP session, depending on the application scenario:

- For the purpose of starting an interactive session, be it a voice telephony call or a video conference, there are two standards. The original standard in this area was ITU recommendation H.323,[62] and more recently the IETF has defined the Session Initiation Protocol (SIP).[28, 111]
- For the purpose of starting a noninteractive session—for example, video-on-demand—the main standard is the Real-Time Streaming Protocol (RTSP).[14]
- The original use of RTP was with IP multicast and the lightweight sessions model of conferencing. This design used the Session Announcement Protocol (SAP)[35] and IP multicast to announce ongoing sessions, such as seminars and TV broad-casts, that were open to the public.

The requirements for these protocols are quite distinct, in terms of the number of participants in the session and the coupling between those participants. Some sessions are very loosely coupled, with only limited membership control and knowledge of the participants. Others are closely managed, with explicit permission required to join, talk, listen, and watch.

These different requirements have led to very different protocols being designed for each scenario, with tremendous ongoing work in this area. RTP deliberately does not include session initiation and control functions, making it suitable for a wide range of applications.

As an application designer, you will have to implement some form of session initiation, call setup, or call control in addition to the media transport provided by RTP.

Session Description

Common to all setup and announcement protocols is the need for a means of describing the session. One commonly used protocol in this area is the Session Description Protocol (SDP),[15] although other mechanisms may be used.

Regardless of the format of the session description, certain information is always needed. It is necessary to convey the transport addresses on which the media flows, the format of the media, the RTP payload formats and profile that are to be used, the times when the session is active, and the purpose of the session.

SDP bundles this information into a text file format, which is human-readable and can be easily parsed. In some cases this file is passed directly to an RTP application, giving it enough information to join a session directly. In others, the session description forms a basis for negotiation, part of a call setup protocol, before a participant can enter a tightly controlled conference call.

SDP is discussed in more detail in Chapter 4, RTP Data Transfer Protocol.

Quality of Service

Although RTP is designed to operate over the best-effort service provided by IP, it is sometimes useful to be able to reserve network resources, giving enhanced quality of service to the RTP flows. Once again, this is not a service provided by RTP, and it is necessary to enlist the help of another protocol. At the time of this writing, there is no commonly accepted "best practice" for resource reservation on the Internet. Two standard frameworks exist, Integrated Services and Differentiated Services, with only limited deployment of each.

The Integrated Services framework provides for strict quality-of-service guarantees, through the use of the Resource ReSerVation Protocol (RSVP).[11] Routers are required to partition the available capacity into service classes, and to account for capacity used by the traffic. Before starting to transmit, a host must signal its requirements to the routers, which permit the reservation to succeed only if sufficient capacity is available in the desired service class. Provided that all the routers respect the service classes and do not overcommit resources, this requirement prevents overloading of the links, providing guaranteed quality of service. The available classes of service include guaranteed service (giving an assured level of bandwidth, a firm end-to-end delay bound, and no congestive packet loss) and controlled load (providing a service equivalent to that of a lightly loaded best-effort network).

The Integrated Services framework and RSVP suffer from the need to make reservations for every flow, and from the difficulty in aggregating these reservations. As a result, scaling RSVP to large numbers of heterogeneous reservations is problematic because of the amount of state that must be kept at the routers, and this constraint has limited its deployment.

The Differentiated Services framework[23], [24] takes a somewhat different approach to quality of service. Instead of providing end-to-end resource reservations and strict performance guarantees, it defines several per-hop queuing behaviors, which it selects by setting the type-of-service field in the IP header of each packet. These per-hop behaviors enable a router to prioritize certain types of traffic to give low probability of loss or delay, but because a router cannot control the amount of traffic admitted to the network, there is no absolute guarantee that performance bounds are met. The advantage of the Differentiated Services framework is that it does not require complex signaling, and the state requirements are much smaller than those for RSVP. The disadvantage is that it provides statistical guarantees only.

The combination of the Integrated and Differentiated Services frameworks is powerful, and future networks may combine them. RSVP can be used to signal application requirements to the edge routers, with those routers then mapping these requirements onto Differentiated Services traffic classes. The combination allows the edge routers to reject excessive traffic, improving the guarantees that can be offered by a Differentiated Services network, while keeping the state required for RSVP out of the core of the network.

Both frameworks have their place, but neither has achieved critical mass at the time of this writing. It is likely, but by no means certain, that future networks will employ some form of quality of service. Until then we are left with the task of making applications perform well in the best-effort network that is currently deployed.

Future Standards Development

With the revision of RTP for draft standard status, there are no known unresolved issues with the protocol specification, and RTP itself is not expected to change in the foreseeable future. This does not mean that the standards work is finished, though. New payload formats are always under development, and work on new profiles will extend RTP to encompass new functionality (for example, the profiles for secure RTP and enhanced feedback).

In the long term, we expect the RTP framework to evolve along with the network itself. Future changes in the network may also affect RTP, and we expect new profiles to be developed to take advantage of any changes. We also expect a continual series of new payload format specifications, to keep up with changes in codec technology and to provide new error resilience schemes.

Finally, we can expect considerable changes in the related protocols for call setup and control, resource reservation, and quality of service. These protocols are newer than RTP, and they are currently undergoing rapid development, implying that changes here will likely be more substantial than changes to RTP, its profile, and payload formats.

Summary

RTP provides a flexible framework for delivery of real-time media, such as audio and video, over IP networks. Its core philosophies—application-level framing and the end-to-end principle—make it well suited to the unique environment of IP networks.

This chapter has provided an overview of the RTP specification, profiles, and payload formats. Related standards cover call setup, control and advertisement, and resource reservation.

The two parts of RTP introduced in this chapter—the data transfer protocol and the control protocol—are covered in detail in the next two chapters.

Chapter 4. RTP Data Transfer Protocol

- RTP Sessions
- The RTP Data Transfer Packet
- Packet Validation
- Translators and Mixers

This chapter explains the RTP data transfer protocol, the means by which real-time media is exchanged. The discussion focuses on the "on-the-wire" aspects of RTP—that is, the packet formats and requirements for interoperability; the design of a system using RTP is explained in later chapters.

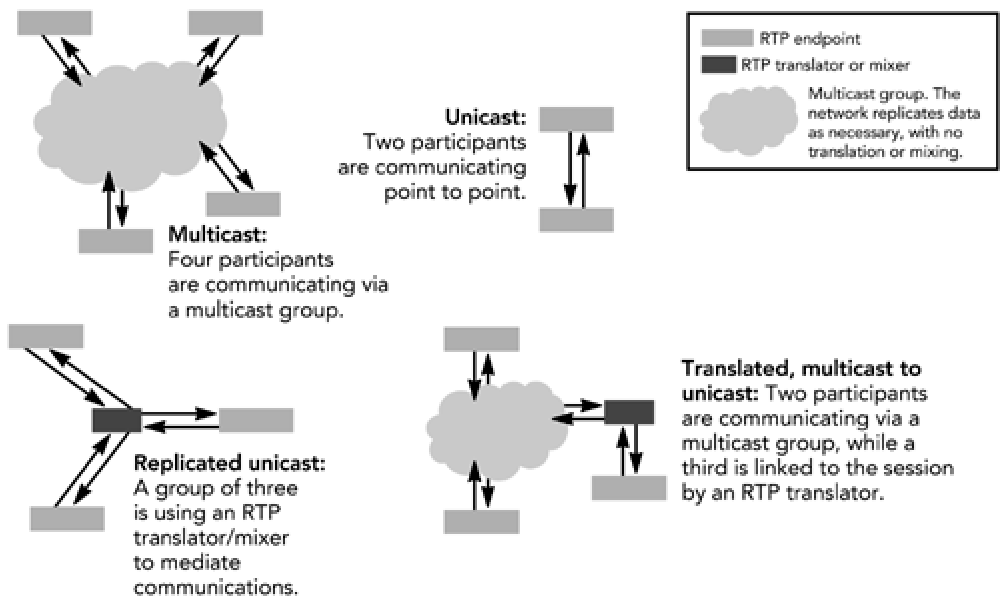
RTP Sessions

A *session* consists of a group of participants who are communicating using RTP. A participant may be active in multiple RTP sessions—for instance, one session for exchanging audio data and another session for exchanging video data. For each participant, the session is identified by a network address and port pair to which data should be sent, and a port pair on which data is received. The send and receive ports may be the same. Each port pair comprises two adjacent ports: an even-numbered port for RTP data packets, and the next higher (odd-numbered) port for RTCP control packets. The default port pair is 5004 and 5005 for UDP/IP, but many applications dynamically allocate ports during session setup and ignore the default. RTP sessions are designed to transport a single type of media; in a multimedia communication, each media type should be carried in a separate RTP session.

The latest revision to the RTP specification relaxes the requirement that the RTP data port be even-numbered, and allows non-adjacent RTP and RTCP ports. This change makes it possible to use RTP in environments where certain types of Network Address Translation (NAT) devices are present. If possible, for compatibility with older implementations, it is wise to use adjacent ports, even though this is not strictly required.

A session can be *unicast*, either directly between two participants (a point-to-point session) or to a central server that redistributes the data. Or it can be *multicast* to a group of participants. A session also need not be restricted to a single transport address space. For example, RTP translators can be used to bridge a session between unicast and multicast, or between IP and another transport, such as IPv6 or ATM. Translators are discussed in more detail later in this chapter, in the section titled Translators and Mixers. Some examples of session topologies are shown in Figure 4.1.

Figure 4.1. Types of RTP Sessions


RTP endpoint
RTP translator or mixer
Multicast group. The network replicates data as necessary, with no translation or mixing.
Unicast: Two participants are communicating point to point.
Multicast: Four participants are communicating via a multicast group.
Replicated unicast: A group of three is using an RTP translator/mixer to mediate communications.
Translated, multicast to unicast: Two participants are communicating via a multicast group, while a third is linked to the session by an RTP translator.

The range of possible sessions means that an RTP end system should be written to be essentially agnostic about the underlying transport. It is good design to restrict knowledge of the transport address and ports to your low-level networking code only, and to use RTP-level mechanisms for participant identification. RTP provides a "synchronization source" for this purpose, described in more detail later in this chapter.

In particular, note these tips:

- You should not use a transport address as a participant identifier because the data may have passed through a translator or mixer that may hide the original source address. Instead, use the synchronization source identifiers.
- You should not assume that a session has only two participants, even if it is using unicast. The other end of the unicast connection may be an RTP translator or mixer acting as a gateway for a potentially unlimited number of other participants.

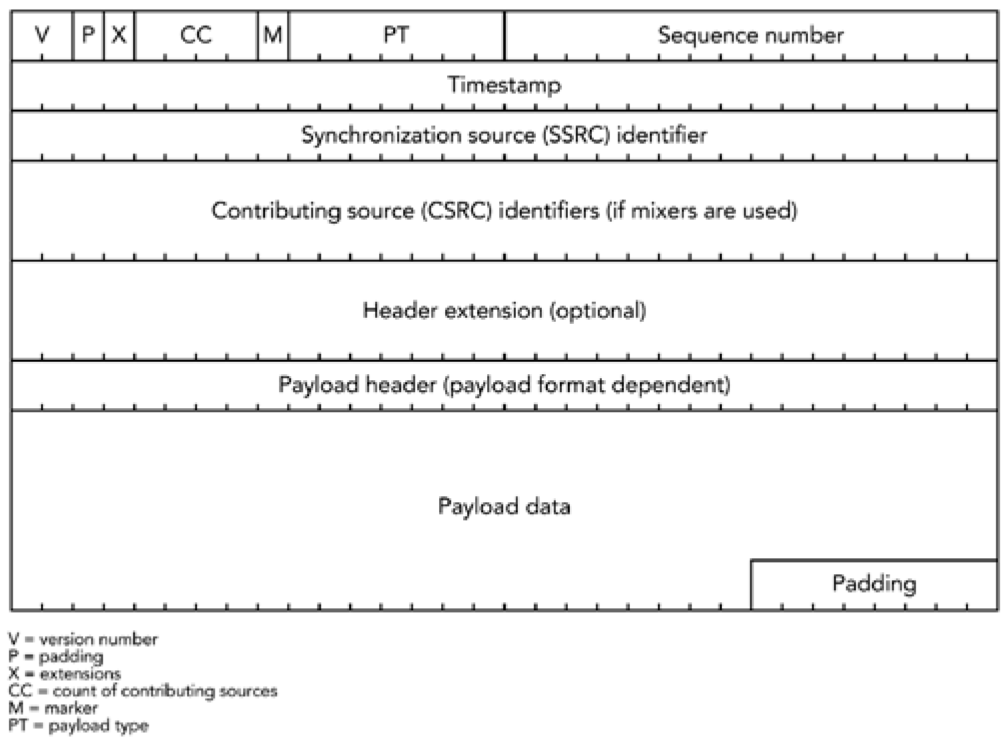
A good design makes the actual means of communication all but invisible to the participants.

The RTP Data Transfer Packet

The format of an RTP data transfer packet is illustrated in Figure 4.2. There are four parts to the packet:

1. The mandatory RTP header
2. An optional header extension
3. An optional payload header (depending on the payload format used)
4. The payload data itself

Figure 4.2. An RTP Data Transfer Packet

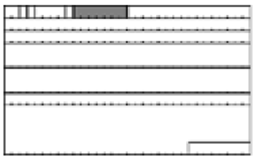

V
P
X
CC
M
PT
Sequence number
Timestamp
Synchronization source (SSRC) identifier
Contributing source (CSRC) identifiers (if mixers are used)
Header extension (optional)
Payload header (payload format dependent)
Payload data
Padding
V = version number
P = padding
X = extensions
CC = count of contributing sources
M = marker
PT = payload type

The entire RTP packet is contained within a lower-layer payload, typically UDP/IP.

Header Elements

The mandatory RTP data packet header is typically 12 octets in length, although it may contain a contributing source list, which can expand the length by 4 to 60 additional octets. The fields in the mandatory header are the payload type, sequence number, time-stamp, and synchronization source identifier. In addition, there is a count of contributing sources, a marker for interesting events, support for padding and a header extension, and a version number.

PAYLOAD TYPE



The *payload type*, or *PT*, field of the RTP header identifies the media transported by an RTP packet. The receiving application examines the payload type to determine how to treat the data—for example, passing it to a particular decompressor. The exact interpretation of the payload field is defined by an RTP profile, which binds the payload type numbers to payload format specifications, or by a non-RTP means.

Many applications operate under the RTP profile for audio and video conferences with minimal control (RFC 1890).[7] This profile (commonly called the *audio/video profile*) defines a table of default mappings between the payload type number and payload format specifications. Examples of these static assignments are shown in Table 4.1 (this is not a complete list; the profile defines additional assignments). In addition to the static assignments, out-of-band signaling—for example, using SIP, RTSP, SAP, or H.323—may be used to define the mapping. Payload types in the range 96 to 127 are reserved for dynamic assignment in this manner when the audio/video profile is being used; other profiles may specify different ranges.

Payload formats are named in terms of the MIME namespace. This namespace was originally defined for e-mail, to identify the content of attachments, but it has since become a general namespace for media formats and is used in many applications. The use of MIME types with RTP is relatively new—payload type names originally occupied a separate namespace—but it is a powerful feature, providing a central repository of transport and encoding options for each type of media.

Table 4.1. Examples of Static Payload Type Assignments

Payload Type Number	Payload Format	Specification	Description
0	AUDIO/PCMU	RFC 1890	ITU G.711 µ-law audio
3	AUDIO/GSM	RFC 1890	GSM full-rate audio
8	AUDIO/PCMA	RFC 1890	ITU G.711 A-law audio
12	AUDIO/QCELP	RFC 2658	PureVoice QCELP audio

Table 4.1. Examples of Static Payload Type Assignments

Payload Type Number	Payload Format	Specification	Description
14	AUDIO/MPA	RFC 2250	MPEG audio (e.g., MP3)
26	VIDEO/JPEG	RFC 2435	Motion JPEG video
31	VIDEO/H261	RFC 2032	ITU H.261 video
32	VIDEO/MPV	RFC 2250	MPEG I/II video

All payload formats should now have a MIME type registration. Newer payload formats include it in their specification; a group registration for the older ones is in progress.[51] The complete list of MIME types is maintained online at http://www.iana.org/assignments/media-types.

Whether the payload type assignment is static or dynamic, it is necessary to describe the session to the application so that the application knows which payload types are to be used. A common means of describing sessions is the Session Description Protocol (SDP).[15] A sample session description might be as follows:

```
v=0
o=bloggs 2890844526 2890842807 IN IP4 10.45.1.82
s=-
e=j.bloggs@example.com(Joe Bloggs)
c=IN IP4 224.2.17.12/127
t=2873397496 2873404696
m=audio 49170 RTP/AVP 0
m=video 51372 RTP/AVP 98
a=rtpmap:98 H263-1998/90000
```

Of interest in our discussion of RTP are the `c=` and `m=` lines, which communicate addresses and ports for the RTP session and define the profile and payload types in use, and the `a=rtpmap:` line, which makes a dynamic payload type assignment.

The example describes two RTP sessions: Audio is being sent to the IPv4 multicast group 224.2.17.12 on port 49170 with time-to-live 127, and video is being sent to the same multicast group on port 51372. Both audio and video use RTP/AVP as their transport; this is RTP transport using the RTP profile for audio and video conferences with minimal control.[7]

The payload type used for audio is 0. This is a static assignment in the profile, the payload format for AUDIO/PCMU. The payload type for video is 98, which is mapped to the payload format for VIDEO/H263-1998 by the `a=rtpmap:` line. By referencing the table of MIME type assignments, we find that the definition of VIDEO/H263-1998 is in RFC 2429.[22]

Although SDP is a common solution for describing RTP sessions, nothing in RTP requires SDP to be used. For example, applications based on ITU recommendation H.323 use RTP for their media transport but use a different mechanism (H.245) for describing sessions.

There has been some debate on the merits of static versus dynamic assignment of payload type numbers to payload formats, encouraged perhaps by the long list of static assignments in the audio/video profile and the perceived complexity of the signaling needed for dynamic assignments.

When RTP was new and its use was the subject of experimentation with simple payload formats, static payload type assignments made sense. A receiver could decode the RTP payload on the basis of the payload type number only, because the codecs required no additional configuration, and the absence of signaling simplified development of these new applications.

However, as designers have gained experience with RTP and applied it to more complex payload formats, it has become clear that the practice of making static assignments is flawed.

Most of the payload formats in use today require some configuration in addition to payload type assignment, requiring the use of signaling; and emerging applications—such as voice-over-IP and video-on-demand—require signaling for user location, authentication, and payment. Because signaling is required anyway, the incentive for static payload type assignment is lost.

Requiring dynamic assignment also avoids problems due to depletion of the payload type space; there are only 127 possible static assignments, and the number of possible payload formats far exceeds that. Dynamic assignment allows for only those formats needed for the duration of a session to be bound to payload type numbers.

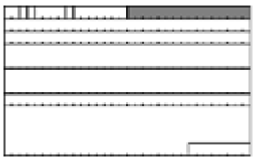
Accordingly, the policy of the IETF Audio/Video Transport working group is that no more static assignments will be made, and that applications should signal their payload type usage out of band.

the RTP media clock, and the format of any payload header and the payload itself. For static assignments, the clock rate is specified in the profile; dynamic assignments must indicate the clock rate along with the mapping between payload type and payload format. For example, in the previous session description the `a=rtpmap:` line specifies a 90,000-Hz clock for the VIDEO/H263-1998 payload format. Most payload formats operate with a limited set of clock rates, with the payload format specification defining which rates are valid.

An RTP session is not required to use only a single payload format; multiple payload formats can be used within a session, with the different formats being identified by different payload types. The format can change at any time within a session, and as long as the mapping from payload type to payload format has been communicated in advance, there is no requirement for signaling before the change occurs. An example might be encoding of DTMF tones within a voice-over-IP session, to support the "Press 0 to speak to an operator" style of automated service, in which one format is used for speech and another for the tones.

Even though multiple payload formats may be used within a session, the payload type is not intended to be used to multiplex different classes of media. For example, if both audio and video are being sent by an application, they should be sent as two different RTP sessions, on different addresses/ports, rather than being sent as a single RTP session and demultiplexed by the payload type. This separation of media allows applications to request different network quality of service for the different media, and it is also required for correct operation of the RTP control protocol.

SEQUENCE NUMBER



The RTP *sequence number* is used to identify packets, and to provide an indication to the receiver if packets are being lost or delivered out of order. It is not used to schedule playout of the packets—that is the purpose of the timestamp—although it does allow the receiver to reconstruct the order in which packets were sent.

The sequence number is an unsigned 16-bit integer, which increases by one with each data packet sent and wraps around to zero when the maximum value is reached. An important consequence of the 16-bit space is that sequence number wrap-around happens relatively often: A typical voice-over-IP application sending audio in 20-millisecond packets will wrap the sequence number approximately every 20 minutes.

This means that applications should not rely on sequence numbers as unique packet identifiers. Instead, it is recommended that they use an extended sequence number, 32 bits or wider, to identify packets internally, with the lower 16 bits being the sequence number from the RTP packet and the upper 16 being a count of the number of times the sequence number has wrapped around:

```
extended_seq_num = seq_num + (65536 * wrap_around_count)
```

Because of possible packet loss or reordering, maintaining the wrap-around counter (`wrap-around-count`) is not a simple matter of incrementing a counter when the sequence number wraps to zero. The RTP specification has an algorithm for maintaining the wrap-around counter:

```
uint16_t   udelta = seq – max_seq
if (udelta < max_dropout) {
   if (seq < max_seq) {
      wrap_around_count++
   }
   max_seq = seq;
} else if (udelta <= 65535 – max_misorder) {
   // The sequence number made a very large jump
   if (seq == bad_seq) {
      // Two sequential packets received; assume the
      // other side has restarted without telling us
      ...
   } else {
      bad_seq = seq + 1;
   }
} else {
   // Duplicate or misordered packet
   ...
}
```

Note that all calculations are done with modulo arithmetic and 16-bit unsigned quantities. Both `seq` and `max_seq` are the unextended sequence numbers from the RTP packets. The RTP specification recommends that `max_misorder = 100` and `max_dropout = 3000`.

If the extended sequence number is calculated immediately on reception of a packet and used thereafter, most of the application can be made unaware of sequence number wrap-around. The ability to hide the wrap-around greatly simplifies loss detection and concealment, packet reordering, and maintenance of statistics. Unless the packet rate is very high, the wrap-around time for a 32-bit sequence number is such that most applications can ignore the possibility. For example, the voice-over-IP example given earlier will take over two years to wrap around the extended sequence number.

If the packet rate is very high, a 32-bit extended sequence number may wrap around while the application is running. When designing applications for such environments, you must either use a larger extended sequence number (for example, 64 bits) to avoid the problem, or build the application to handle wrap-around by performing all calculations on sequence numbers using 32-bit modulo arithmetic. Incorrect operation during wrap-around of the sequence number is a common problem, especially when packets are lost or reordered around the time of wrap-around.

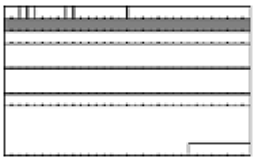
The initial value of the sequence number should be chosen randomly, rather than starting from zero. This precaution is intended to make known plain-text attacks on an encrypted RTP stream more difficult. Use of a random initial sequence number is important even if the source does not encrypt, because the stream may pass through an encrypting translator that is not known to the source, and adding the random offset in the translator is not trivial (because sequence numbers are reported in RTCP reception report packets; see Chapter 5, RTP Control Protocol). A common implementation problem is to assume that the sequence numbers start from zero; receivers should be able to play out a stream irrespective of the initial sequence number (this capability is also needed to handle late joins).

The sequence number should always follow a continuous sequence, increasing by one for each packet sent, and never jumping forward or backward (except for wrap-around, of course). This requirement should apply across changes in payload format regardless of how the media is generated. For example, when you're splicing together video clips—perhaps to insert advertisements—the RTP sequence number space must be continuous, and it must not be reset at the start of each clip. This has implications for the design of streaming media servers because they cannot rely on sequence numbers stored with a media file and must generate sequence numbers on the fly.

The primary use of the sequence number is loss detection. A gap in the sequence number space indicates to the receiver that it must take action to recover or conceal the missing data. This is discussed in more detail in Chapters 8, Error Concealment, and 9, Error Correction.

A secondary use of the sequence number is to allow reconstruction of the order in which packets were sent. A receiver does not necessarily care about this—because many payload formats allow for decoding of packets in any order—but sorting the packets into order as they are received may make loss detection easier. The design of playout buffer algorithms is discussed in more detail in Chapter 6, Media Capture, Playout, and Timing.

TIMESTAMP



The RTP *timestamp* denotes the sampling instant for the first octet of media data in a packet, and it is used to schedule playout of the media data. The timestamp is a 32-bit unsigned integer that increases at a media-dependent rate and wraps around to zero when the maximum value is exceeded. With typical video codecs, a clock rate of 90kHz is used, corresponding to a wrap-around of approximately 13 hours; with 8kHz audio the interval is approximately 6 days.

The initial value of the timestamp is randomly chosen, rather than starting from zero. As with the sequence number, this precaution is intended to make known plain-text attacks on an encrypted RTP stream more difficult. Use of a random initial timestamp is important even if the source does not encrypt, because the stream may pass through an encrypting translator that is not known to the source. A common implementation problem is to assume that the timestamp starts from zero. Receivers should be able to play out a stream irrespective of the initial timestamp and be prepared to handle wrap-around; because the timestamp does not start at zero, a wrap-around could occur at any time.

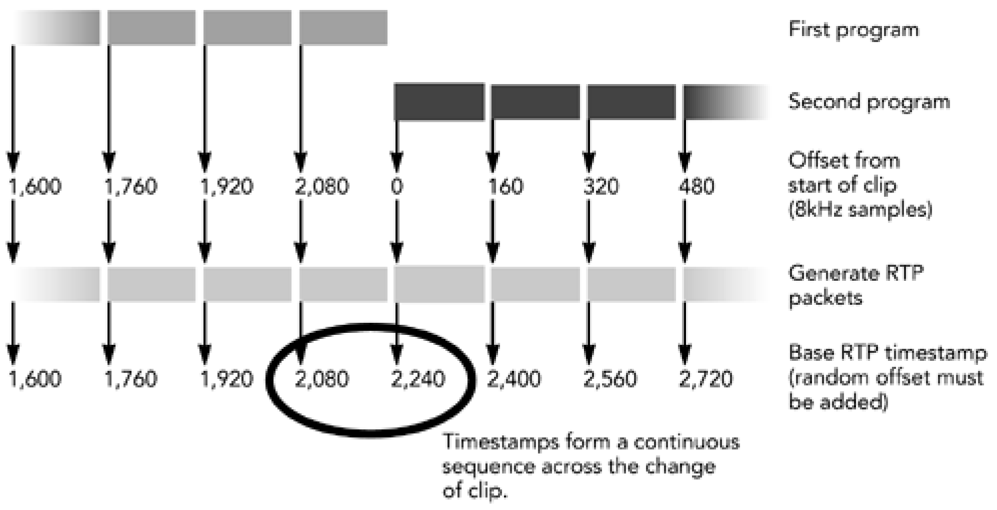
Timestamp wrap-around is a normal part of RTP operation and should be handled by all applications. The use of extended timestamps, perhaps 64-bit values, can make most of the application unaware of the wrap-around. Extended timestamps are not recommended, though, because 64-bit arithmetic is often inefficient on today's processors.

A better design performs all timestamp calculations using 32-bit modulo arithmetic. This approach allows differences between timestamps to be calculated, provided that the packets compared are within half the timestamp space of each other.

The timestamp is derived from a media clock that must increase in a linear and monotonic fashion (except for wrap-around, of course), producing a single timeline for each RTP session. This is true irrespective of the means by which the media stream is generated.

An example is useful to clarify the implications of the way in which the timestamp increases: When audio clips are spliced together within a single RTP session, the RTP timestamps must form a continuous sequence and must not be reset at the start of each clip. These requirements are illustrated in Figure 4.3, which shows that an RTP receiver cannot tell from the RTP headers that a change has occurred.

Figure 4.3. Formation of a Continuous Timestamp Sequence across Two Clips That Have Been Spliced Together



First program
Second program
1,600
1,760
1,920
2,080
0
160
320
480
Offset from start of clip (8kHz samples)
Generate RTP packets
1,600
1,760
1,920
2,080
2,240
2,400
2,560
2,720
Base RTP timestamp (random offset must be added)
Timestamps form a continuous sequence across the change of clip.

The same is true when a fast-forward or rewind operation occurs: The timestamps must form a continuous sequence and not jump around. This requirement is evident in the design of the Real-Time Streaming Protocol (RTSP),[14] which includes the concept of "normal play time" representing the time index into the stream. Because the continuity of RTP timestamps must be maintained, an RTSP server has to send an updated mapping between RTP timestamps and the normal play time during a seek operation.

The continuity of RTP timestamps has implications for the design of streaming media servers. The servers cannot rely on timestamps (or sequence numbers) stored with a media file but must generate them on the fly, taking into account seek operations within the media and the duration of any previous data that has been played out within the RTP session.

The requirement for a media clock that increases in a linear and monotonic fashion does not necessarily imply that the order in which media data is sampled is the order in which it is sent. After media frames have been generated—and hence have obtained their timestamps—they may be reordered before packetization. As a result, packets may be transmitted out of timestamp order, even though the sequence number order is maintained. The receiver has to reconstruct the timestamp order to play out the media.

An example is MPEG video, which contains both key frames and delta-encoded frames predicted from them forward (P-frames) and backward (B-frames). When B-frames are used, they are predicted from a later packet and hence must be delayed and sent out of order. The result is that the RTP stream will have non-monotonically increasing timestamps.[12] Another example is the use of interleaving to reduce the effects of burst loss (see the section titled Interleaving in Chapter 8, Error Concealment). In all cases, a single timeline, which the receiver must reconstruct to play out the media, is retained.

Timestamps on RTP packets are not necessarily unique within each wrap-around cycle. If two packets contain data from the same sampling instant, they will have the same timestamp. Duplication of timestamps typically occurs when a large video frame is split into multiple RTP packets for transmission (the packets will have different sequence numbers but the same timestamp).

The nominal rate of the media clock used to generate timestamps is defined by the profile and/or payload format in use. For payload formats with static payload type assignments, the clock rate is implicit when the static payload type is used (it is specified as part of the payload type assignment). The dynamic assignment process must specify the rate along with the payload type (see the section titled Payload Type earlier in this chapter). The chosen rate must be sufficient to perform lip synchronization with the desired accuracy, and to measure variation in network transit time. The clock rate may not be chosen arbitrarily; most payload formats define one or more acceptable rates.

Audio payload formats typically use the sampling rate as their media clock, so the clock increases by one for each full sample read. There are two exceptions: MPEG audio uses a 90kHz clock for compatibility with non-RTP MPEG transport; and G.722, a 16kHz speech codec, uses an 8kHz media clock for backward compatibility with RFC 1890, which mistakenly specified 8kHz instead of 16kHz.

Video payload formats typically use a 90kHz clock, for compatibility with MPEG and because doing so yields integer timestamp increments for the typical 24Hz, 25Hz, 29.97Hz, and 30Hz frame rates and the 50Hz, 59.94Hz, and 60Hz field rates in widespread use today. Examples include PAL (Phase Alternating Line) and NTSC (National Television Standards Committee) television, plus HDTV (High-Definition Television) formats.

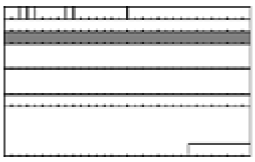
It is important to remember that RTP makes no guarantee as to the resolution, accuracy, or stability of the media clock—those properties are considered application dependent, and outside the scope of RTP—and in general, all that is known is its nominal rate. Applications should be able to cope with variability in the media clock, both at the sender and at the receiver, unless they have specific knowledge to the contrary.

In some cases it may be possible to define the resolution, accuracy, and stability of the media clock, and to use this knowledge to simplify application design. This is typically possible only when a single entity controls both sender and receiver, or when both are designed to a profile with a strict clock specification

.

The process by which a receiver reconstructs the correct timing of a media stream based on the timestamps is described in Chapter 6, Media Capture, Playout, and Timing.

SYNCHRONIZATION SOURCE



The *synchronization source* (SSRC) identifies participants within an RTP session. It is an ephemeral, per-session identifier that is mapped to a long-lived canonical name, CNAME, through the RTP control protocol (see the section titled RTCP SDES: Source Description, in Chapter 5, RTP Control Protocol).

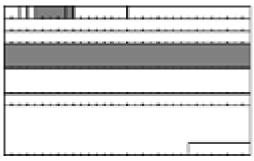
The SSRC is a 32-bit integer, chosen randomly by participants when they join the session. Having chosen an SSRC identifier, the participant uses it in the packets it sends out. Because SSRC values are chosen locally, two participants can select the same value. Such collisions may be detected when one application receives a packet from another that contains the SSRC identifier chosen for itself.

If a participant detects a collision between the SSRC it is using and that chosen by another participant, it must send an RTCP BYE for the original SSRC (see the section titled RTCP BYE: Membership Control, in Chapter 5, RTP Control Protocol) and select another SSRC for itself. This collision detection mechanism ensures that the SSRC is unique for each participant within a session.

It is important that a high-quality source of randomness is used to generate the SSRC, and that collision detection is implemented. In particular, the seed for the random number generator should not be based on the time at which the session is joined or on the transport addresses of the session, because collisions can result if multiple participants join at once.

All packets with the same SSRC form part of a single timing and sequence number space, so a receiver must group packets by SSRC for playback. If a participant generates multiple streams in one RTP session—for example, from separate video cameras—each must be identified as a different SSRC so that the receivers can distinguish which packets belong to each stream.

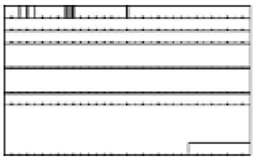
CONTRIBUTING SOURCES



Under normal circumstances, RTP data is generated by a single source, but when multiple RTP streams pass through a mixer or translator, multiple data sources may have contributed to an RTP data packet. The list of *contributing sources* (CSRCs) identifies participants who have contributed to an RTP packet but were not responsible for its timing and synchronization. Each contributing source identifier is a 32-bit integer, corresponding to the SSRC of the participant who contributed to this packet. The length of the CSRC list is indicated by the CC field in the RTP header.

Packets containing a CSRC list are produced by the operation of an RTP mixer, as described later in this chapter, in the section titled Mixers. When receiving a packet containing a CSRC list, the SSRC is used to group packets for playout in the usual manner, and each CSRC is added to the list of known participants. Each participant identified by a CSRC will have a corresponding stream of RTP control protocol packets, providing fuller identification of the participant.

MARKER



The *marker* (M) bit in the RTP header is used to mark events of interest within a media stream; its precise meaning is defined by the RTP profile and media type in use.

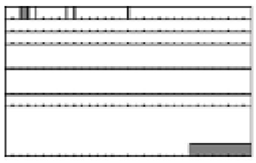
For audio streams operating under the RTP profile for audio and video conferences with minimal control, the marker bit is set to one to indicate the first packet sent after a period of silence, and otherwise set to zero. A marker bit set to one serves as a hint to the application that this may be a good time to adjust its playout point, because a small variation in the length of a silence period is not usually noticeable to listeners (whereas a change in the playout point while audio is being played is audible).

For video streams operating under the RTP profile for audio and video conferences with minimal control, the marker bit is set to one to indicate the last packet of a video frame, and otherwise set to zero. If set to one, the marker serves as a hint that the application can begin decoding the frame, rather than waiting for the following packet—which will have a different timestamp—to detect that the frame should be displayed.

In all cases, the marker bit provides only a hint to the application, which should be designed to operate even if packets with the marker set are lost. For audio streams, it is usually possible to intuit the end of a silent period because the relationship between sequence number and timestamp changes. The start of a video frame can be detected by a change in the timestamp. An application can use these observations to operate with reduced performance if the packets containing the marker bit are lost.

It is possible for an RTP profile to specify that additional marker bits exist, at the expense of a smaller payload type field. For example, a profile could mandate two marker bits and a six-bit payload type. No current profiles use this feature.

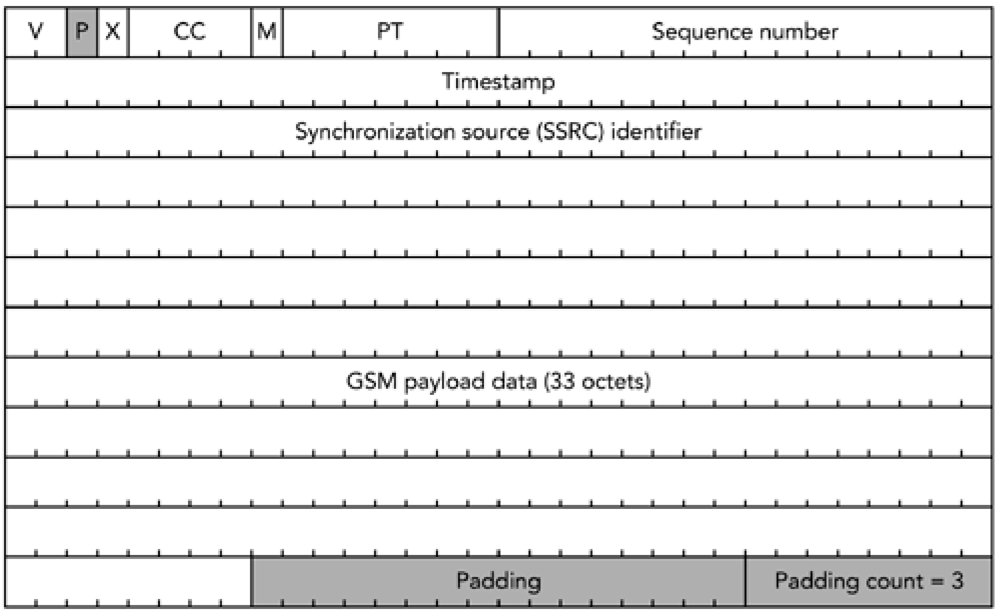
PADDING



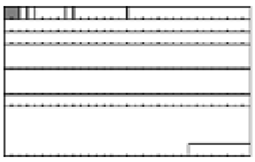
The *padding* (P) bit in the RTP header is used to indicate that the payload has been padded out past its natural length. If padding is added to an RTP packet, the P bit is set and the last octet of the payload is filled with a count of the number of padding octets. Padding is rarely used, but it is needed for some encryption schemes that work with particular block sizes, and to adapt a payload format to a fixed-capacity channel.

As an example of the use of padding, Figure 4.4 shows a GSM audio frame packetized in RTP that has been padded out to 48 octets from its natural length of 45 octets (33 for the GSM frame, 12 for the RTP header). This padding might be needed if the packet were encrypted with the Data Encryption Standard (DES),[56] which requires 8-octet (64-bit) blocks.

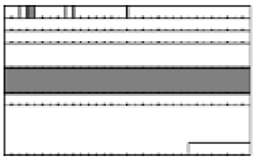
Figure 4.4. An RTP Packet Carrying a GSM Audio Frame, to Which Three Padding Bits Have Been Added


V
P
X
CC
M
PT
Sequence number
Timestamp
Synchronization source (SSRC) identifier
GSM payload data (33 octets)
Padding
Padding count = 3

VERSION NUMBER

Each RTP packet contains a *version number*, indicated by the V field. The current version of RTP to define additional versions, and the previous versions of RTP are not in widespread use. The only meaningful use of the version number field is as part of a packet validity check.

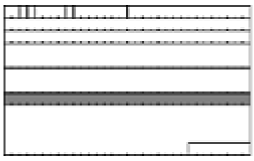
Header Extensions

RTP allows for the possibility that extension headers, signaled by the X bit being set to one, are present after the fixed RTP header, but before any payload header and the payload itself. The extension headers are of variable length, but they start with a 16-bit type field followed by a 16-bit length field (which counts the length of the extension in octets, excluding the initial 32 bits), allowing the extension to be ignored by receivers that do not understand it.

Extension headers provide for experiments that require more header information than that provided by the fixed RTP header. They are rarely used; extensions that require additional, payload format–independent header information are best written as a new RTP profile. If additional headers are required for a particular payload format, they should not use a header extension and instead should be carried in the payload section of the packet as a payload header.

Although header extensions are extremely rare, robust implementations should be prepared to process packets containing an unrecognized header extension by ignoring the extension.

Payload Headers


The mandatory RTP header provides information that is common to all payload formats. In many cases a payload format will need more information for optimal operation; this information forms an additional header that is defined as part of the payload format specification. The payload header is included in an RTP packet following the fixed header and any CSRC list and header extension. Often the definition of the payload header constitutes the majority of a payload format specification.

The information contained in a payload header can be either static—the same for every session using a particular payload format—or dynamic. The payload format specification will indicate which parts of the payload header are static and which are dynamic, and it must be configured on a per-session basis. Those parts that are dynamic are usually configured through SDP,[15] with the `a=fmtp:` attribute used to define "format parameters," although other means are sometimes used. The parameters that can be specified fall into three categories:

1. Those that affect the format of the payload header, signaling the presence or absence of header fields, their size, and their format. For example, some payload formats have several modes of operation, which may require different header fields for their use.
2. Those that do not affect the format of the payload header but do define the use of various header fields. For example, some payload formats define the use of interleaving and require header fields to indicate the position within the interleaving sequence.
3. Those that affect the payload format in lieu of a payload header. For example, parameters may specify the frame size for audio codecs, or the video frame rate.

Features of the payload format that do not change during a session usually are signaled out of band, rather than being included in the payload header. This reduces overheads during the session, at the expense of additional signaling complexity. The syntax and use of format parameters are usually specified as part of the payload format specification.

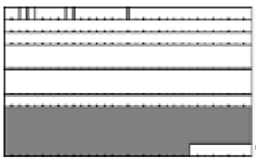
The primary reason for specifying payload headers is to provide error resilience for those formats that were not designed for use over lossy packet networks. The first example of this was the payload format for H.261 video, as discussed in RFC 2032 and RFC 2736.[9, 33] More recent examples are the more loss-tolerant payload formats for MP3 and AMR (Adaptive Multi-Rate) audio.[38, 41] The issue of error resilience is discussed further in Chapters 8, Error Concealment, and 9, Error Correction.

The RTP payload format for H.261 video offers an interesting lesson in design for error resilience. The H.261 codec allows groups of video blocks to be up to 3 kilobytes long. The original version of the payload format specified that each group of blocks should be inserted directly into an RTP packet, or if too large, should be arbitrarily split across packets. However, this approach leaves circumstances in which packets arrive at the receiver and must be discarded because the previous packet was lost, and the partial group of blocks is not independently decodable. This is a loss multiplier effect that we want to avoid.

In fact, the group of blocks is not the smallest unit in H.261 video. There are smaller units called *macro-blocks*, but they are not identifiable without parsing from the start of the group of blocks. However, including additional information at the start of each packet makes it possible to reinstate information that would normally be found by parsing from the start of the group of blocks. This technique is used to define a payload format that splits the H.261 stream on macro-block boundaries if it exceeds the network MTU.

This is a less obvious packetization for H.261, but it does mean that a smart decoder can reconstruct valid H.261 video from a stream of RTP packets that has experienced loss, without having to discard any of the data that arrived. It shows the advantages of error-resilient design of payload formats.

Payload Data

One or more frames of media payload data, directly following any payload header, make up the final part of an RTP packet (other than padding, if

needed). The size and format of the payload data depend on the payload format and format parameters chosen during session setup.

Many payload formats allow for multiple frames of data to be included in each packet. There are two ways in which a receiver can determine how many frames are present:

1. In many cases frames are of a fixed size, and it is possible to determine the number present by inspecting the size of the packet.
2. Other payload formats include an identifier in each encapsulated frame that indicates the size of the frame. An application needs to parse the encapsulated frames to determine the number of frames and their start points. This is usually the case when the frames can be variable sizes.

Usually no limit on the number of frames that may be included is specified. Receivers are expected to handle reception of packets with a range of sizes: The guidelines in the audio/video profile suggest up to 200 milliseconds worth of audio, in multiples of the frame size, and video codecs should handle both fragmented and complete frames.

There are two key issues to consider when you're choosing the amount of payload data to include in each packet: the maximum transmission unit (MTU) of the network path that will be traversed, and the latency induced by waiting for more data to be produced to fill a longer packet.

Packets that exceed the MTU will be either fragmented or dropped. It is clearly undesirable if oversize packets are dropped; less obvious are the problems due to fragmentation. A fragmented packet will be reassembled at the receiver, provided that all fragments arrive. If any fragment is lost, the entire packet must be discarded even though some parts of it were correctly received. The result is a loss multiplier effect, which can be avoided if the packets are sized appropriately, and if the payload format is designed such that each packet can be independently decoded (as discussed in relation to payload headers).

Latency is another concern because a packet cannot be sent until the last octet of data it will contain is produced. The data at the start of the packet is delayed until the complete packet is ready. In many applications, the latency concern provides a tighter constraint on the application than the MTU does.

Packet Validation

Because RTP sessions typically use a dynamically negotiated port pair, it is especially important to validate that packets received really are RTP, and not misdirected other data. At first glance, confirming this fact is nontrivial because RTP packets do not contain an explicit protocol identifier; however, by observing the progression of header fields over several packets, we can quickly obtain strong confidence in the validity of an RTP stream.

Possible validity checks that can be performed on a stream of RTP packets are outlined in Appendix A of the RTP specification. There are two types of tests:

1. **Per-packet checking**, based on fixed known values of the header fields. For example, packets in which the version number is not equal to 2 are invalid, as are those with an unexpected payload type.
2. **Per-flow checking**, based on patterns in the header fields. For example, if the SSRC is constant, and the sequence number increments by one with each packet received, and the timestamp intervals are appropriate for the payload type, this is almost certainly an RTP flow and not a misdirected stream.

The per-flow checks are more likely to detect invalid packets, but they require additional state to be kept in the receiver. This state is required for a valid source, but care must be taken because holding too much state to detect invalid sources can lead to a denial-of-service attack, in which a malicious source floods a receiver with a stream of bogus packets designed to use up resources.

A robust implementation will employ strong per-packet validity checks to weed out as many invalid packets as possible before committing resources to the per-flow checks to catch the others. It should also be prepared to aggressively discard state for sources that appear to be bogus, to mitigate the effects of denial-of-service attacks.

It is also possible to validate the contents of an RTP data stream against the corresponding RTCP control packets. To do this, the application discards RTP packets until an RTCP source description packet with the same SSRC is received. This is a very strong validity check, but it can result in significant validation delay, particularly in large sessions (because the RTCP reporting interval can be many seconds). For this reason we recommend that applications validate the RTP data stream directly, using RTCP as confirmation rather than the primary means of validation.

Translators and Mixers

In addition to normal end systems, RTP supports middle boxes that can operate on a media stream within a session. Two classes of middle boxes are defined: translators and mixers.

Translators

A *translator* is an intermediate system that operates on RTP data while maintaining the synchronization source and timeline of a stream. Examples include systems that convert between media-encoding formats without mixing, that bridge between different transport protocols, that add or remove encryption, or that filter media streams. A translator is invisible to the RTP end systems unless those systems have prior knowledge of the untranslated media. There are a few classes of translators:

- **Bridges**. Bridges are one-to-one translators that don't change the media encoding—for example, gateways between different transport protocols, like RTP/UDP/IP and RTP/ATM, or RTP/UDP/IPv4 and RTP/UDP/IPv6. Bridges make up the simplest class of translator, and typically they cause no changes to the RTP or RTCP data.
- **Transcoders**. Transcoders are one-to-one translators that change the media encoding—for example, decoding the compressed data and reencoding it with a different payload format—to better suit the characteristics of the output network. The payload type usually changes, as may the padding, but other RTP header fields generally remain unchanged. These translations require state to be maintained so that the RTCP sender reports can be adjusted to match, because they contain counts of source bit rate.
- **Exploders**. Exploders are one-to-many translators, which take in a single packet and produce multiple packets. For example, they receive a stream in which multiple frames of codec output are included within each RTP packet, and they produce output with a single frame per packet. The generated packets have the same SSRC, but the other RTP header fields may have to be changed, depending on the translation. These translations require maintenance of bidirectional state: The translator must adjust both outgoing RTCP sender reports and returning receiver reports to match.
- **Mergers**. Mergers are many-to-one translators, combining multiple packets into one. This is the inverse of the previous category, and the same issues apply.

The defining characteristic of a translator is that each input stream produces a single output stream, with the same SSRC. The translator itself is not a participant in the RTP session—it does not have an SSRC and does not generate RTCP itself—and is invisible to the other participants.

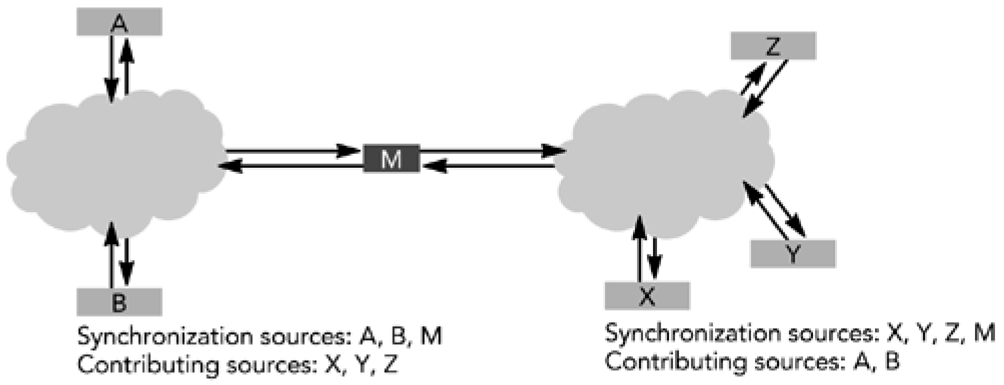
Mixers

A *mixer* is an intermediate system that receives RTP packets from a group of sources and combines them into a single output, possibly changing the encoding, before forwarding the result. Examples include the networked equivalent of an audio mixing deck, or a video picture-in-picture device.

Because the timing of the input streams generally will not be synchronized, the mixer will have to make its own adjustments to synchronize the media before combining them, and hence it becomes the synchronization source of the output media stream. A mixer may use playout buffers for each arriving media stream to help maintain the timing relationships between streams. A mixer has its own SSRC, which is inserted into the data packets it generates. The SSRC identifiers from the input data packets are copied into the CSRC list of the output packet.

A mixer has a unique view of the session: It sees all sources as synchronization sources, whereas the other participants see some synchronization sources and some contributing sources. In Figure 4.5, for example, participant X receives data from three synchronization sources—Y, Z, and M—with A and B contributing sources in the mixed packets coming from M. Participant A sees B and M as synchronization sources with X, Y, and Z contributing to M. The mixer generates RTCP sender and receiver reports separately for each half of the session, and it does not forward them between the two halves. It forwards RTCP source description and BYE packets so that all participants can be identified (RTCP is discussed in Chapter 5, RTP Control Protocol).

Figure 4.5. Mixer M Sees All Sources as Synchronization Sources; Other Participants (A, B, X, Y, and Z) See a Combination of Synchronization and Contributing Sources.


A
M
Z
B
X
Y
Synchronization sources: A, B, M
Contributing sources: X, Y, Z
Synchronization sources: X, Y, Z, M
Contributing sources: A, B

A mixer is not required to use the same SSRC for each half of the session, but it must send RTCP source description and BYE packets into *both* sessions for all SSRC identifiers it uses. Otherwise, participants in one half will not know that the SSRC is in use in the other half, and they may collide with it.

It is important to track which sources are present on each side of the translator or mixer, to detect when incorrect configuration has produced a loop (for example, if two translators or mixers are connected in parallel, forwarding packets in a circle). A translator or mixer should cease operation if a loop is detected, logging as much diagnostic information about the cause as possible. The source IP address of the looped packets is most helpful because it identifies the host that caused the loop.

Summary

This chapter has described the on-the-wire aspects of the RTP data transfer protocol in some detail. We considered the format of the RTP header and its use, including the payload type for identification of the format of the data, sequence number to detect loss, timestamp to show when to play out data, and synchronization source as a participant identifier. We also discussed the minor header fields: marker, padding, and version number.

The concept of the payload format, and its mapping onto payload type identifier and payload header, should now be apparent, showing how RTP is tailored to different types of media. This is an important topic, to which we will return in later chapters.

Finally, we discussed RTP translators and mixers: intermediate systems that extend the reach of RTP in a controlled manner, allowing sessions to bridge heterogeneity of the network.

Associated with the RTP data transfer protocol is a control channel, RTCP, which has been mentioned several times in this chapter. The next chapter focuses on this control channel in some depth, completing our discussion of the network aspects of RTP.

Chapter 5. RTP Control Protocol

- Components of RTCP
- Transport of RTCP Packets
- RTCP Packet Formats
- Security and Privacy
- Packet Validation
- Participant Database
- Timing Rules

There are two parts to RTP: the data transfer protocol, which was described in Chapter 4, and an associated control protocol, which is described in this chapter. The control protocol, RTCP, provides for periodic reporting of reception quality, participant identification and other source description information, notification on changes in session membership, and the information needed to synchronize media streams.

This chapter describes the uses of RTCP, the format of RTCP packets, and the timing rules used to scale RTCP over the full range of session sizes. It also discusses the issues in building a participant database, using the information contained in RTCP packets.

Components of RTCP

An RTCP implementation has three parts: the packet formats, the timing rules, and the participant database.

There are several types of RTCP packets. The five standard packet types are described in the section titled RTCP Packet Formats later in this chapter, along with the rules by which they must be aggregated into compound packets for transmission. Algorithms by which implementations can check RTCP packets for correctness are described in the section titled Packet Validation.

The compound packets are sent periodically, according to the rules described in the section titled Timing Rules later in this chapter. The interval between packets is known as the *reporting interval*. All RTCP activity happens in multiples of the reporting interval. In addition to being the time between packets, it is the time over which reception quality statistics are calculated, and the time between updates of source description and lip synchronization information. The interval varies according to the media format in use and the size of the session; typically it is on the order of 5 seconds for small sessions, but it can increase to several minutes for very large groups. Senders are given special consideration in the calculation of the reporting interval, so their source description and lip synchronization information is sent frequently; receivers report less often.

Each implementation is expected to maintain a participant database, based on the information collected from the RTCP packets it receives. This database is used to fill out the reception report packets that have to be sent periodically, but also for lip synchronization between received audio and video streams and to maintain source description information. The privacy concerns inherent in the participant database are mentioned in the section titled Security and Privacy later in this chapter. The Participant Database section, also in this chapter, describes the maintenance of the participant database.

Transport of RTCP Packets

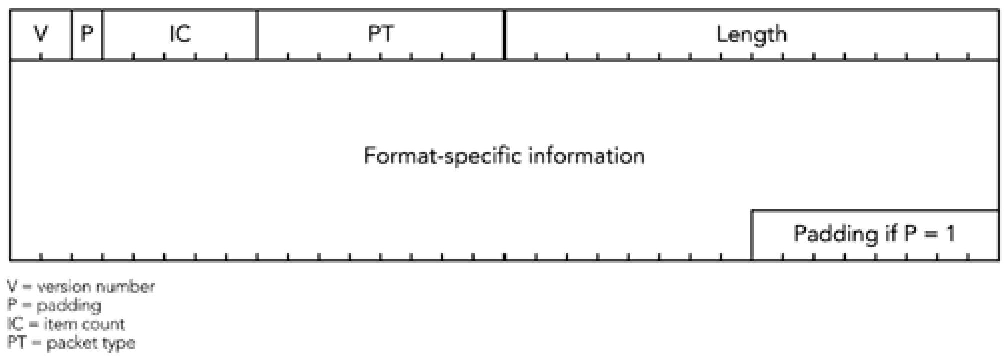
Each RTP session is identified by a network address and a pair of ports: one for RTP data and one for RTCP data. The RTP data port should be even, and the RTCP port should be one above the RTP port. For example, if media data is being sent on UDP port 5004, the control channel will be sent to the same address on UDP port 5005.

All participants in a session should send compound RTCP packets and, in turn, will receive the compound RTCP packets sent by all other participants. Note that feedback is sent to *all* participants in a multiparty session: either unicast to a translator, which then redistributes the data, or directly via multicast. The peer-to-peer nature of RTCP gives each participant in a session knowledge of all other participants: their presence, reception quality, and—optionally—personal details such as name, e-mail address, location, and phone number.

RTCP Packet Formats

Five types of RTCP packets are defined in the RTP specification: receiver report (RR), sender report (SR), source description (SDES), membership management (BYE), and application-defined (APP). They all follow a common structure—illustrated in Figure 5.1—although the format-specific information changes depending on the type of packet.

Figure 5.1. The Basic RTCP Packet Format



The header that all five packet types have in common is four octets in length, comprising five fields:

1. **Version number (V)**. The version number is always 2 for the current version of RTP. There are no plans to introduce new versions, and previous versions are not in widespread use.
2. **Padding (P)**. The padding bit indicates that the packet has been padded out beyond its natural size. If this bit is set, one or more octets of padding have been added to the end of this packet, and the last octet contains a count of the number of padding octets added. Its use is much the same as the

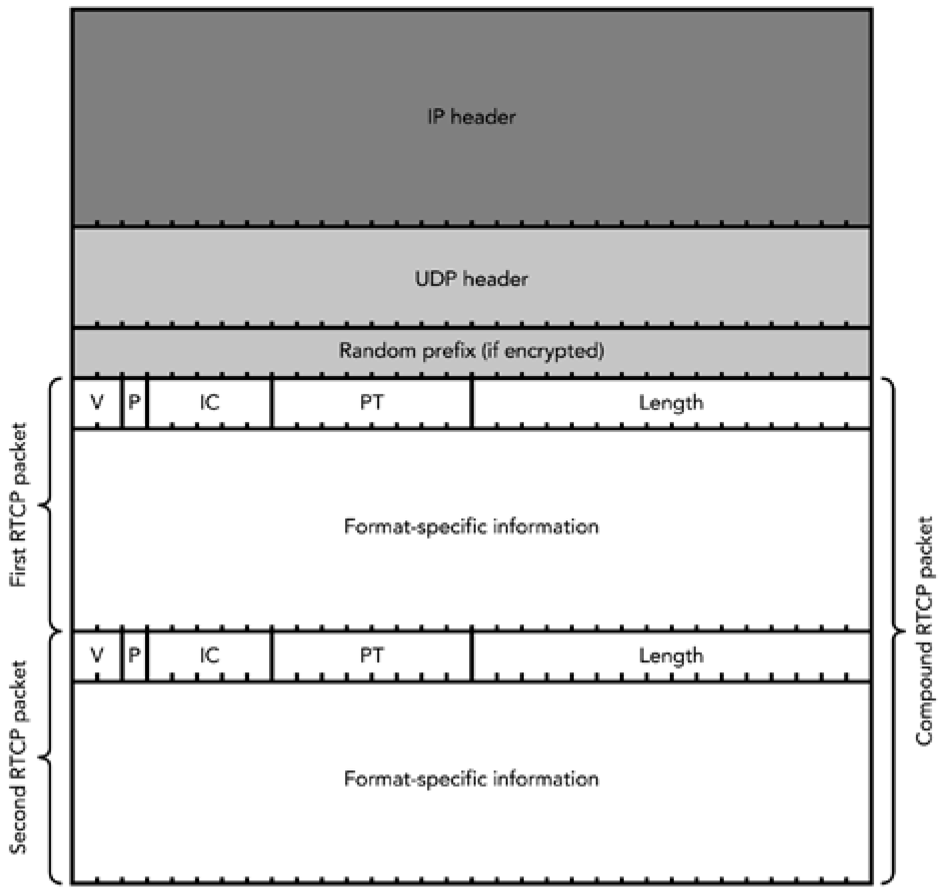
padding bit in RTP data packets, which was discussed in Chapter 4, RTP Data Transfer Protocol, in the section titled Padding. Incorrect use of the padding bit has been a common problem with RTCP implementations; the correct usage is described in the sections titled Packing Issues and Packet Validation later in this chapter.

3. **Item count (IC)**. Some packet types contain a list of items, perhaps in addition to some fixed, type-specific information. The item count field is used by these packet types to indicate the number of items included in the packet (the field has different names in different packet types depending on its use). Up to 31 items may be included in each RTCP packet, limited also by the maximum transmission unit of the network. If more than 31 items are needed, the application must generate multiple RTCP packets. An item count of zero indicates that the list of items is empty (this does not necessarily mean that the packet is empty). Packet types that don't need an item count may use this field for other purposes.
4. **Packet type (PT)**. The packet type identifies the type of information carried in the packet. Five standard packet types are defined in the RTP specification; other types may be defined in the future (for example, to report additional statistics or to convey other source-specific information).
5. **Length**. The length field denotes the length of the packet contents following the common header. It is measured in units of 32-bit words because all RTCP packets are multiples of 32 bits in length, so counting octets would only allow the possibility of inconsistency. Zero is a valid length, indicating that the packet consists of only the four-octet header (the IC header field will also be zero in this case).

Following the RTCP header is the packet data (the format of which depends on the packet type) and optional padding. The combination of header and data is an RTCP packet. The five standard types of RTCP packets are described in the sections that follow.

RTCP packets are *never* transported individually; instead they are always grouped together for transmission, forming compound packets. Each compound packet is encapsulated in a single lower-layer packet—often a UDP/IP packet—for transport. If the compound packet is to be encrypted, the group of RTCP packets is prefixed by a 32-bit random value. The structure of a compound packet is illustrated in Figure 5.2.

Figure 5.2. Format of a Compound RTCP Packet



A set of rules governs the order in which RTCP packets are grouped to form a compound packet. These rules are described later in the chapter, in the section titled Packing Issues, after the five types of RTCP packets have been described in more detail.

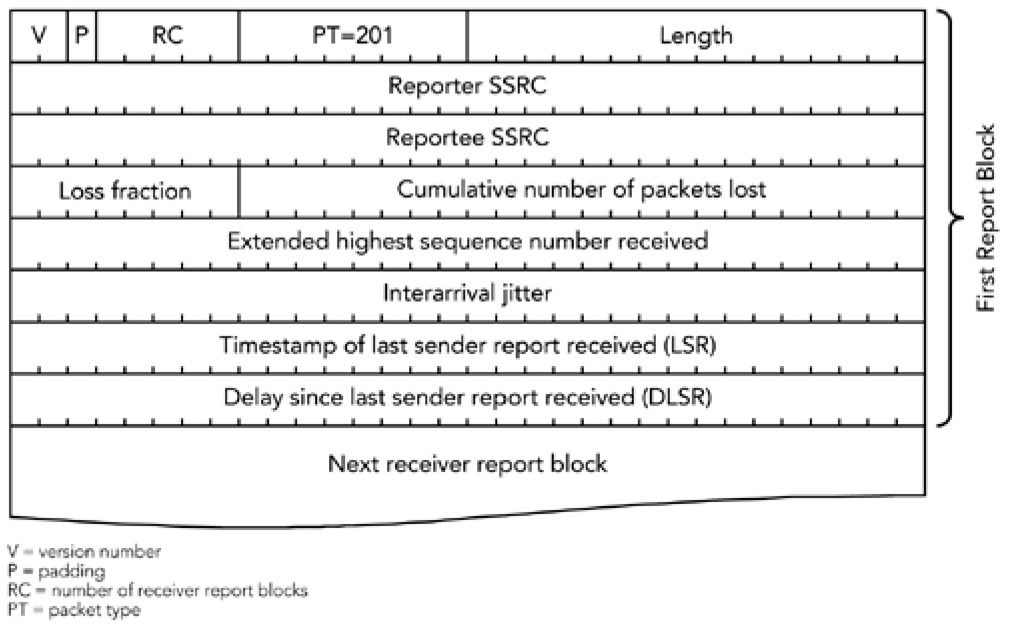
RTCP RR: Receiver Reports

One of the primary uses of RTCP is reception quality reporting, which is accomplished through RTCP receiver report (RR) packets, which are sent by all participants who receive data.

THE RTCP RR PACKET FORMAT

A receiver report packet is identified by a packet type of 201 and has the format illustrated in Figure 5.3. A receiver report packet contains the SSRC (synchronization source) of the participant who is sending the report (the *reporter SSRC*) followed by zero or more report blocks, denoted by the RC field.

Figure 5.3. Format of an RTCP RR Packet



V
P
RC
PT=201
Length
Reporter SSRC
Reportee SSRC
Loss fraction
Cumulative number of packets lost
Extended highest sequence number received
Interarrival jitter
Timestamp of last sender report received (LSR)
Delay since last sender report received (DLSR)
Next receiver report block
First Report Block
V = version number
P = padding
RC = number of receiver report blocks
PT = packet type

Many RTCP packet types have a structure similar to that of receiver reports, with a list of items following the fixed part of the packet. Note that the fixed part of the packet remains even if the list of items is empty, implying that a receiver report with no report blocks will have `RC=0`, but `Length=1`, corresponding to the four-octet fixed RTCP header plus the four-octet reporter SSRC.

Each report block describes the reception quality of a single synchronization source from which the reporter has received RTP packets during the current reporting interval. A total of 31 report blocks can be in each RTCP RR packet. If there are more than 31 active senders, the receiver should send multiple RR packets in a compound packet. Each report block has seven fields, for a total of 24 octets.

The *reportee SSRC* identifies the participant to whom this report block pertains. The statistics in the report block denote the quality of reception for the reportee synchronization source, as received at the participant generating the RR packet.

The *cumulative number of packets lost* is a 24-bit *signed* integer denoting the number of packets expected, less the number of packets actually received. The number of packets expected is defined to be the extended last sequence number received, less the initial sequence number received. The number of packets received includes any that are late or duplicated, and hence may be greater than the number expected, so the cumulative number of packets lost may be negative. The cumulative number of packets lost is calculated for the entire duration of the session, not per interval. This field saturates at the maximum positive value of `0x7FFFFF` if more packets than that are lost during the session.

Many of the RTCP statistics are maintained for the duration of the session, rather than per reporting interval. If an SSRC collision occurs, however, or if there is a very large gap in the sequence number space, such that the receiver cannot tell whether the fields may have wrapped, then the statistics are reset to zero.

The *extended highest sequence number* received in the RTP data packets from this synchronization source is calculated as discussed in Chapter 4, RTP Data Transfer Protocol, in the section titled Sequence Number. Because of possible packet reordering, this is not necessarily the extended sequence number of the last RTP packet received. The extended highest sequence number is calculated per session, not per interval.

The *loss fraction* is defined as the number of packets lost in this reporting interval, divided by the number expected. The loss fraction is expressed as a fixed-point number with the binary point at the left edge of the field, which is equivalent to the integer part after multiplying the loss fraction by 256 (that is, if 1/4 of the packets were lost, the loss fraction would be 1/4 x 256 = 64). If the number of packets received is greater than the number expected, because of the presence of duplicates, making the number of packets lost negative, then the loss fraction is set to zero.

The *interarrival jitter* is an estimate of the statistical variance in network transit time for the data packets sent by the reportee synchronization source. Interarrival jitter is measured in timestamp units, so it is expressed as a 32-bit unsigned integer, like the RTP timestamp.

To calculate the variance in network transit time, it is necessary to measure the transit time. Because sender and receiver typically do not have synchronized clocks, however, it is not possible to measure the absolute transit time. Instead the relative transit time is calculated as the difference between a packet's RTP timestamp and the receiver's RTP clock at the time of arrival, measured in the same units. This calculation requires the receiver to maintain a clock for each source, running at the same nominal rate as the media clock for that source, from which to derive these relative timestamps. (This clock may be the receiver's local playout clock, if that runs at the same rate as the source clocks.) Because of the lack of synchronization between the clocks of sender and receiver, the relative transit time includes an unknown constant offset. This is not a problem, because we are interested only in the variation in transit time: the difference in spacing between two packets at the receiver versus the spacing when they left the sender. In the following computation the constant offset due to unsynchronized clocks is accounted for by the subtraction.

If S_i is the RTP timestamp from packet *i*, and R_i is the time of arrival in RTP timestamp units for packet *i*, then the relative transit time is $(R_i - S_i)$, and for two packets, *i* and *j*, the difference in relative transit time may be expressed as

$$D(i,j) = (R_j - S_j) - (R_i - S_i)$$

Note that the timestamps, R_x and S_x, are 32-bit unsigned integers, whereas $D(i,j)$ is a signed quantity. The calculation is performed with modulo arithmetic (in C, this means that the timestamps are of type `unsigned int`, provided that `sizeof(unsigned int) == 4`).

The interarrival jitter is calculated as each data packet is received, using the difference in relative transit times $D(i,j)$ for that packet and the previous packet received (which is not necessarily the previous packet in sequence number order). The jitter is maintained as a moving average, according to the following formula:

$$J_i = J_{i-1} + \frac{(|D(i-1,i)| - J_{i-1})}{16}$$

Whenever a reception report is generated, the current value of J_i for the reportee SSRC is included as the interarrival jitter.

The *last sender report* (*LSR*) timestamp is the middle 32 bits out of the 64-bit NTP (Network Time Protocol) format timestamp included in the most recent RTCP SR packet received from the reportee SSRC. If no SR has been received yet, the field is set to zero.

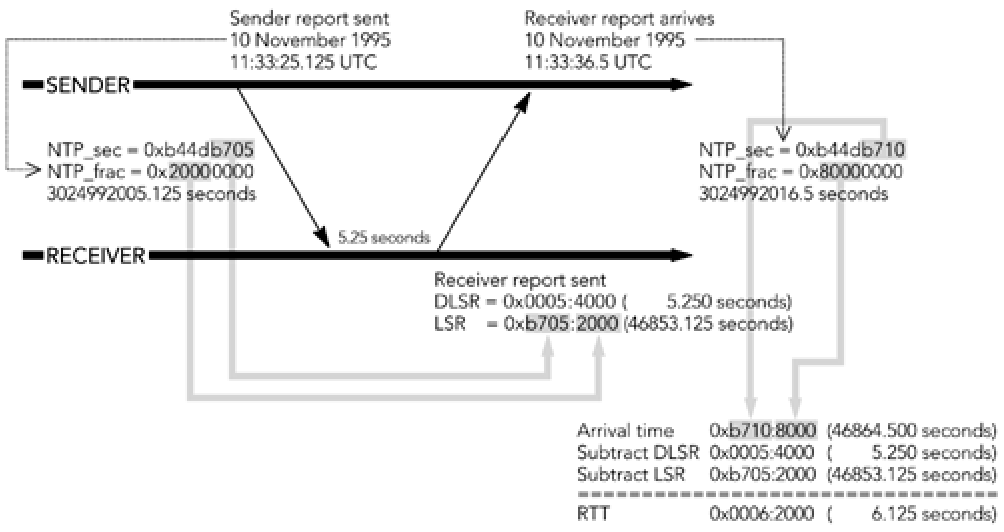
The *delay since last sender report* (*DLSR*) is the delay, expressed in units of 1/65,536 seconds, between receiving the last SR packet from the reportee SSRC and sending this reception report block. If no SR packet has been received from the reportee SSRC, the DLSR field is set to zero.

INTERPRETING RR DATA

The reception quality feedback in RR packets is useful not only for the sender, but also for other participants and third-party monitoring tools. The feedback provided in RR packets can allow the sender to adapt its transmissions according to the feedback. In addition, other participants can determine whether problems are local or common to several receivers, and network managers may use monitors that receive only the RTCP packets to evaluate the performance of their networks.

A sender can use the LSR and DLSR fields to calculate the round-trip time between it and each receiver. On receiving an RR packet pertaining to it, the sender subtracts the LSR field from the current time, to give the delay between sending the SR and receiving this RR. The sender then subtracts the DLSR field to remove the offset introduced by the delay in the receiver, to get the network round-trip time. The process is shown in Figure 5.4, an example taken from the RTP specification. (Note that RFC 1889 contains an error in this example, which has been corrected in the new version of the RTP specification.)

Figure 5.4. Sample Round-Trip Time (RTT) Computation.



Note that the calculated value is the *network* round-trip time, and it excludes any processing at the endpoints. For example, the receiver must buffer the data to smooth the effects of jitter before it can play the media (see Chapter 6, Media Capture, Playout, and Timing).

The round-trip time is important in interactive applications because delay hinders interactivity. Studies have shown that it is difficult to conduct conversations when the total round-trip time exceeds about 300 milliseconds[61] (this number is approximate and depends on the listener and the task being performed). A sender may use knowledge of the round-trip time to optimize the media encoding—for example, by generating packets that contain less data to reduce packetization delays—or to drive the use of error correction codes (see Chapter 9, Error Correction).

The fraction lost gives an indication of the short-term packet loss rates to a receiver. By watching trends in the reported statistics, a sender can judge whether the loss is a transient or a long-term effect. Many of the statistics in RR packets are cumulative values, to allow long-term averaging. Differences can be calculated between any two RR packets, making measurements over both short and long periods possible and giving resilience to the loss of reports.

For example, the packet loss rate over the interval between RR packets can be derived from the cumulative statistics, as well as being directly reported. The difference in the cumulative number of packets lost gives the number lost during that interval, and the difference in the extended last sequence numbers gives the number of packets expected during the interval. The ratio of these values is the fraction of packets lost. This number should be equal to the fraction lost field in the RR packet if the calculation is done with consecutive RR packets, but the ratio also gives an estimate of the loss fraction if one or more RR packets have been lost, and it can show negative loss when there are duplicate packets. The advantage of the fraction lost field is that it provides loss information from a single RR packet. This is useful in very large sessions, in which the reporting interval is long enough that two RR packets may not have been received.

Loss rates can be used to influence the choice of media format and error protection coding used (see Chapter 9, Error Correction). In particular, a higher loss rate indicates that a more loss-tolerant format should be used, and that, if possible, the data rate should be reduced (because most loss is caused by congestion; see Chapter 2, Voice and Video Communication over Packet Networks, and Chapter 10, Congestion Control).

The jitter field may also be used to detect the onset of congestion: A sudden increase in jitter will often precede the onset of packet loss. This effect depends on the network topology and the number of flows, with high degrees of statistical multiplexing reducing the correlation between increased jitter and the onset of packet congestion.

Senders should be aware that the jitter estimate depends on packets being sent with spacing that matches their timestamp. If the sender delays some packets, that delay will be counted as part of the network jitter. This can be an issue with video, where multiple packets are often generated with the same timestamp but are spaced for transmission rather than being sent as a burst. This is not necessarily a problem, because the jitter measure still gives an indication of the amount of buffer space that the receiver will require (because the buffer space needs to accommodate both the jitter and the spacing delay).

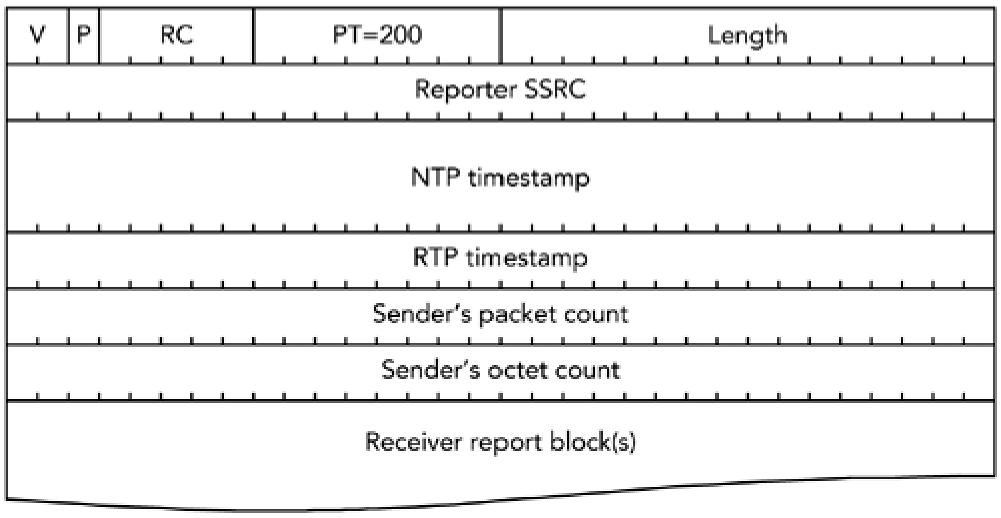
RTCP SR: Sender Reports

In addition to reception quality reports from receivers, RTCP conveys sender report (SR) packets sent by participants that have recently sent data. These provide information on the media being sent, primarily so that receivers can synchronize multiple media streams (for example, to lipsync audio and video).

THE RTCP SR PACKET FORMAT

A sender report packet is identified by a packet type of 200 and has the format illustrated in Figure 5.5. The payload contains a 24-octet sender information block followed by zero or more receiver report blocks, denoted by the RC field, exactly as if this were a receiver report packet. Receiver report blocks are present when the sender is also a receiver.

Figure 5.5. Format of an RTCP SR Packet

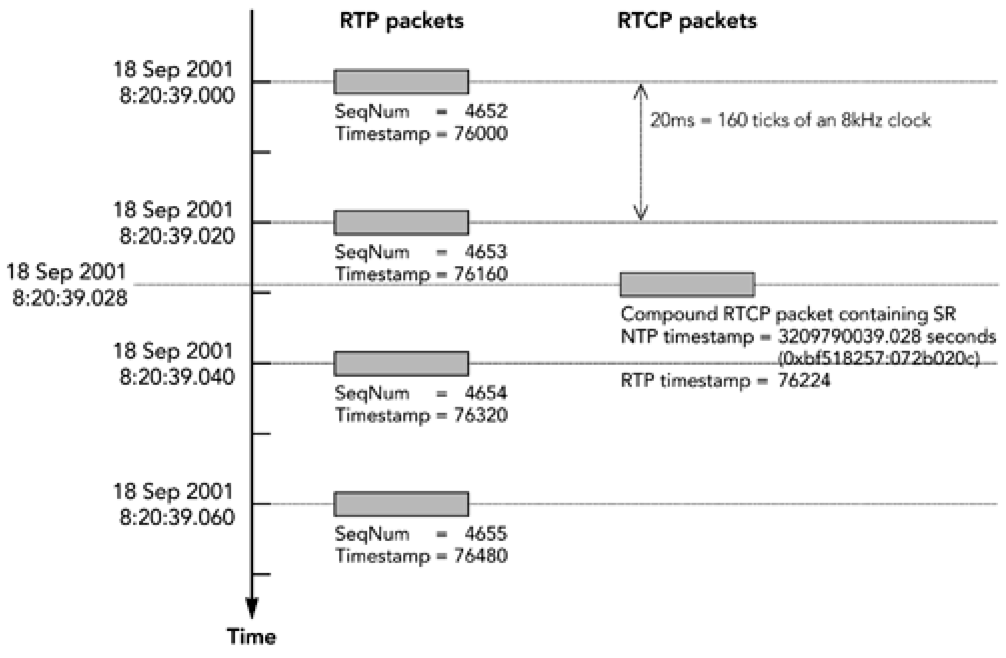


The NTP *timestamp* is a 64-bit unsigned value that indicates the time at which this RTCP SR packet was sent. It is in the format of an NTP timestamp, counting seconds since January 1, 1900, in the upper 32 bits, with the lower 32 bits representing fractions of a second (that is, a 64-bit fixed-point value, with the binary point after 32 bits). To convert a UNIX timestamp (seconds since 1970) to NTP time, add 2,208,988,800 seconds.

Although the NTP timestamp in RTCP SR packets uses the format of an NTP timestamp, the clock does not have to be synchronized with the Network Time Protocol or have any particular accuracy, resolution, or stability. For a receiver to synchronize two media streams, however, those streams must be related to the *same clock*. The Network Time Protocol[5] is occasionally useful for synchronizing the sending clocks, although it is needed only if the media streams to be synchronized are generated by different systems. These issues are discussed further in Chapter 7, Lip Synchronization.

The *RTP timestamp* corresponds to the same instant as the NTP timestamp, but it is expressed in the units of the RTP media clock. The value is generally *not* the same as the RTP timestamp of the previous data packet, because some time will have elapsed since the data in that packet was sampled. Figure 5.6 shows an example of the SR packet timestamps. The SR packet has the RTP timestamp corresponding to the time at which it is sent, which does *not* correspond to either of the surrounding RTP data packets.

Figure 5.6. Use of Timestamps in RTCP SR Packets



The *sender's packet count* is the number of data packets that this synchronization source has generated since the beginning of the session. The *sender's octet count* is the number of octets contained in the payload of those data packets (not including the headers or any padding).

The packet count and octet count fields are reset if a sender changes its SSRC (for example, because of a collision). They will eventually wrap around if the source continues to transmit for a long time, but generally this is not a problem. Subtraction of an older value from a newer value will give the correct result if 32-bit modulo arithmetic is used and no more than 2^{32} counts occurred in between, even if there was a wrap-around (in C, this means that the counters are of type `unsigned int`, as long as `sizeof(unsigned int) == 4`). The packet and octet counts enable receivers to calculate the average data rate of the source.

INTERPRETING SR DATA

From the SR information, an application can calculate the average payload data rate and the average packet rate over an interval without receiving the data. The ratio of the two is the average payload size. If it can be assumed that packet loss is independent of packet size, the number of packets received by a particular receiver, multiplied by the average payload size (or the corresponding packet size), gives the apparent throughput available to that receiver.

The timestamps are used to generate a correspondence between media clocks and a known external reference (the NTP format clock). This makes lip synchronization possible, as explained in Chapter 7.

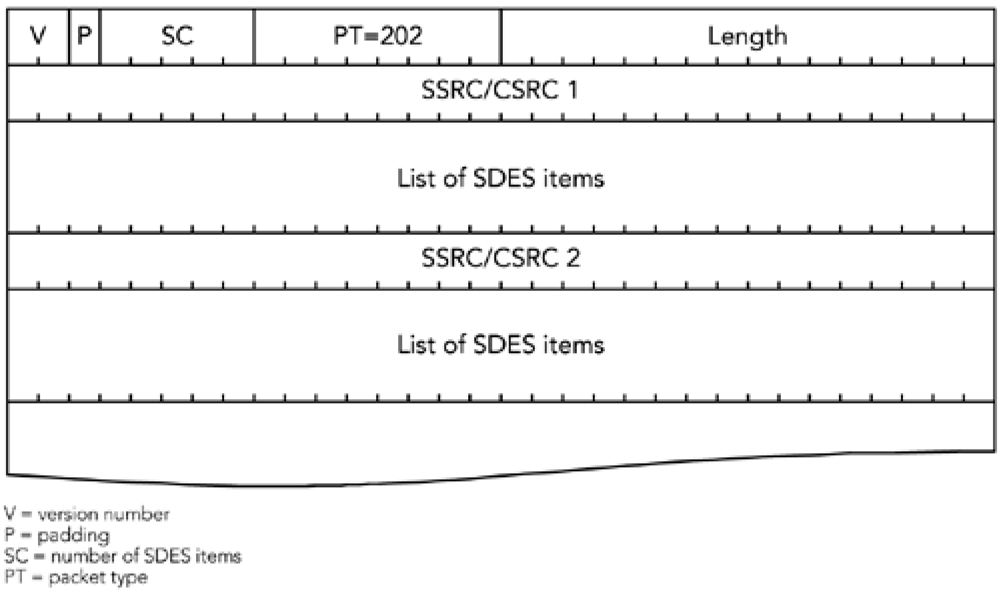
RTCP SDES: Source Description

RTCP can also be used to convey source description (SDES) packets that provide participant identification and supplementary details, such as location, e-mail address, and telephone number. The information in SDES packets is typically entered by the user and is often displayed in the graphical user interface of an application, although this depends on the nature of the application (for example, a system providing a gateway from the telephone system into RTP might use the SDES packets to convey caller ID).

THE RTCP SDES PACKET FORMAT

Each source description packet has the format illustrated in Figure 5.7 and uses RTCP packet type 202. SDES packets comprise zero or more lists of SDES items, the exact number denoted by the SC header field, each of which contains information on a single source.

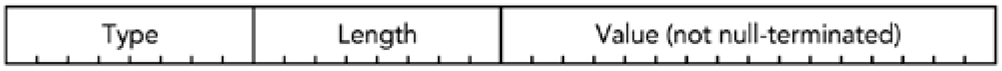
Figure 5.7. Format of an RTCP SDES Packet



It is possible for an application to generate packets with empty lists of SDES items, in which case the SC and length fields in the RTCP common header will both be zero. In normal use, SC is equal to one (mixers and translators that are aggregating forwarded information will generate packets with larger lists of SDES items).

Each list of SDES items starts with the SSRC of the source being described, followed by one or more entries with the format shown in Figure 5.8. Each entry starts with a type and a length field, then the item text itself in UTF-8 format.[13] The length field indicates how many octets of text are present; the text is *not* null-terminated.

Figure 5.8. Format of an SDES Item



The entries in each SDES item are packed into the packet in a continuous manner, with no separation or padding. The list of items is terminated by one or more null octets, the first of which is interpreted as an item of type zero to denote the end of the list. No length octet follows the null item type octet, but additional null octets must be included if needed to pad until a 32-bit boundary is reached. Note that this padding is separate from that indicated by the P bit in the RTCP header. A list with zero items (four null octets) is valid but useless.

Several types of SDES items are defined in the RTP specification, and others may be defined by future profiles. Item type zero is reserved and indicates the end of the list of items. The other standard item types are CNAME, NAME, EMAIL, PHONE, LOC, TOOL, NOTE, and PRIV.

STANDARD SDES ITEMS

The CNAME item (type = 1) provides a canonical name (CNAME) for each participant. It provides a stable and persistent identifier, independent of the synchronization source (because the SSRC will change if an application restarts or if an SSRC collision occurs). The CNAME can be used to associate multiple media streams from a participant across different RTP sessions (for example, to associate voice and video that need to be synchronized), and to name a participant across restarts of a media tool. It is the only mandatory SDES item; all implementations are required to send SDES CNAME items.

The CNAME is allocated algorithmically from the user name and host IP address of the participant. For example, if the author were using an IPv4-based application, the CNAME might be `csp@10.7.42.16`. IPv6 applications use the colon-separated numeric form of the address.[16] If the application is running on a system with no notion of user names, the host IP address only is used (with no user name or @ symbol).

As long as each participant joins only a single RTP session—or a related set of sessions that are intended to be synchronized—the use of user name and host IP address is sufficient to generate a consistent unique identifier. If media streams from multiple hosts, or from multiple users, are to be synchronized, then the senders of those streams must collude to generate a consistent CNAME (which typically is the one chosen algorithmically by one of the participants).

> The use of private addresses and Network Address Translation (NAT) means that IP addresses are no longer globally unique. For lip synchronization and other uses of associated RTP sessions to operate correctly through Network Address Translation, the translator must also translate the RTCP CNAME to a unique form when crossing domain boundaries. This translation must be consistent across multiple RTP streams.

The NAME item (type = 2) conveys the participant's name and is intended primarily to be displayed in lists of participants as part of the user interface. This value is typically entered by the user, so applications should not assume anything about its value; in particular, it should not be assumed to be unique.

The EMAIL item (type = 3) conveys the e-mail address of a participant formatted as in RFC 822[2]—for example, jdoe@example.com. Sending applications should attempt to validate that the EMAIL value is a syntactically correct e-mail address before including it in an SDES item; receivers cannot assume that it is a valid address.

The PHONE item (type = 4) conveys the telephone number of a participant. The RTP specification recommends that this be a complete international number, with a plus sign replacing the international access code (for example, +1 918 555 1212 for a number in the United States), but many implementations allow users to enter this value with no check on format.

The LOC item (type = 5) conveys the location of the participant. Many implementations allow the user to enter the value directly, but it is possible to convey location in any format. For example, an implementation could be linked to the Global Positioning System and include GPS coordinates as its location.

The TOOL item (type = 6) indicates the RTP implementation—the tool—in use by the participant. This field is intended for debugging and marketing purposes. It should include the name and version number of the implementation. Typically the user is not able to edit the contents of this field.

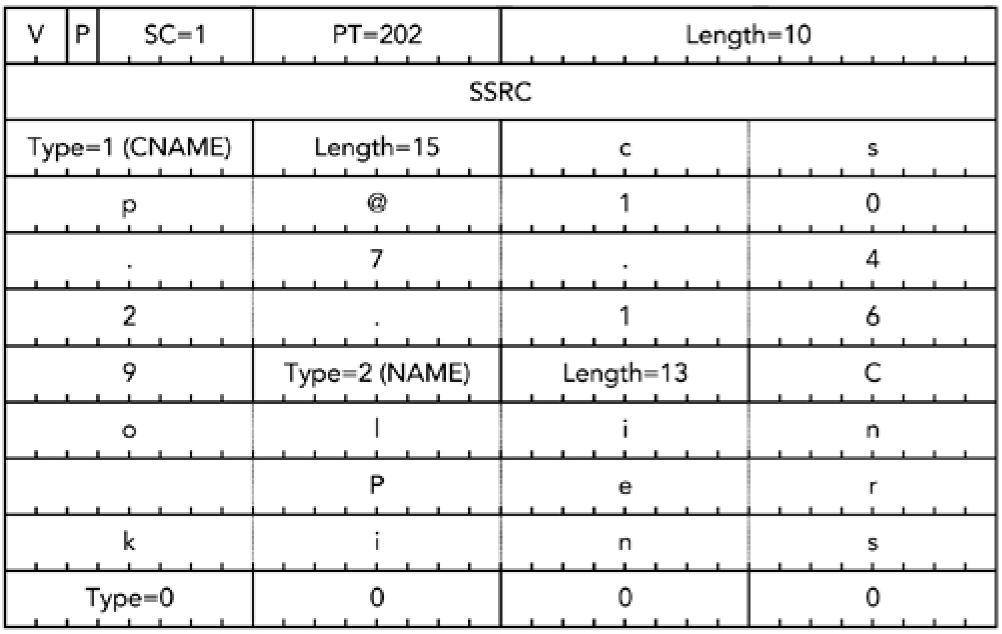
The NOTE item (type = 7) allows the participant to make a brief statement about anything. It works well for a "back in five minutes" type of note, but it is not really suitable for instant messaging, because of the potentially long delay between RTCP packets.

PRIV items (type = 8) are a private extension mechanism, used to define experimental or application-specific SDES extensions. The text of the item begins with an additional single-octet length field and prefix string, followed by a value string that fills the remainder of the item. The intention is that the initial prefix names the extension and is followed by the value of that extension. PRIV items are rarely used; extensions can more efficiently be managed if new SDES item types are defined.

The CNAME is the only SDES item that applications are required to transmit. An implementation should be prepared to receive any of the SDES items, even if it ignores them. There are various privacy issues with SDES (see the section titled Security and Privacy later in this chapter), which means that an implementation should not send any information in addition to the CNAME unless the user has explicitly authorized it to do so.

Figure 5.9 shows an example of a complete RTCP source description packet containing CNAME and NAME items. Note the use of padding at the end of the list of SDES items, to ensure that the packet fits into a multiple of 32 bits.

Figure 5.9. A Sample SDES Packet



V | P | SC=1	PT=202	Length=10	
SSRC			
Type=1 (CNAME)	Length=15	c	s
p	@	1	0
.	7	.	4
2	.	1	6
9	Type=2 (NAME)	Length=13	C
o	l	i	n
	P	e	r
k	i	n	s
Type=0	0	0	0

PARSER ISSUES

When implementing a parser for SDES packets, you should remember three important points:

1. The text of SDES items is not null-terminated, implying that manipulating SDES items in languages that assume null-terminated strings requires care. In C, for example, SDES items should be manipulated with `strncpy()`, which allow strings up to a specified length to be copied (use of `strcpy()` is inappropriate because the text is not null-terminated). Care-less implementations may be susceptible to buffer overflow attacks, which are a serious security risk.
2. The text of SDES items is in UTF-8 format; local character sets require conversion before use. It is often necessary to query the locale in use on the system, and to convert between the system character set and UTF-8. Some applications inadvertently generate SDES packets with the wrong character set; an implementation should be robust to this mistake (for example, if the use of an incorrect character set causes the UTF-8 parser to produce an invalid Unicode character).
3. The text of SDES items may be entered by the user and cannot be trusted to have safe values. In particular, it may contain metacharacters that have undesirable side effects. For example, some user interface scripting languages allow command substitution to be triggered by metacharacters, potentially giving an attacker the means to execute arbitrary code. Implementers should take steps to ensure safe handling of SDES data in their environment.

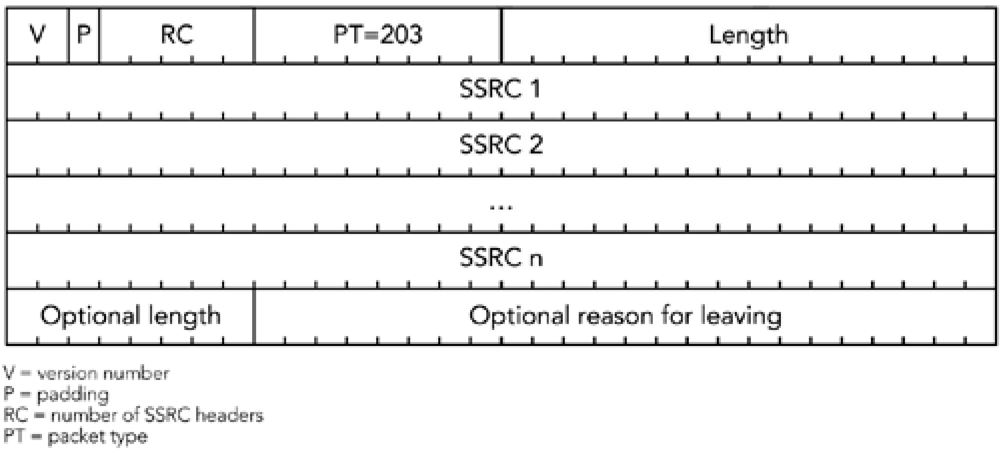
RTCP BYE: Membership Control

RTCP provides for loose membership control through RTCP BYE packets, which indicate that some participants have left the session. A BYE packet is generated when a participant leaves the session, or when it changes its SSRC—for example, because of a collision. BYE packets may be lost in transit, and some applications do not generate them; so a receiver must be prepared to time out participants who have not been heard from for some time, even if no BYE has been received from them.

The significance of a BYE packet depends, to some extent, on the application. It always indicates that a participant is leaving the RTP session, but there may also be a signaling relationship between the participants (for example, SIP, RTSP, or H.323). An RTCP BYE packet does not terminate any other relationship between the participants.

BYE packets are identified by packet type 203 and have the format shown in Figure 5.10. The RC field in the common RTCP header indicates the number of SSRC identifiers in the packet. A value of zero is valid but useless. On receiving a BYE packet, an implementation should assume that the listed sources have left the session and ignore any further RTP and RTCP packets from that source. It is important to keep state for departing participants for some time after a BYE has been received, to allow for delayed data packets.

Figure 5.10. Format of an RTCP BYE Packet



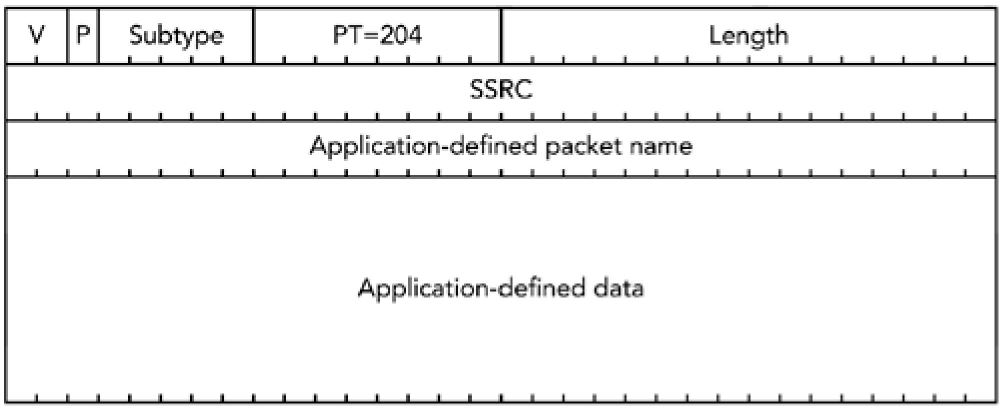
The section titled Participant Database later in this chapter further describes the state maintenance issues relating to timeout of participants and RTCP BYE packets.

A BYE packet may also contain text indicating the reason for leaving a session, suitable for display in the user interface. This text is optional, but an implementation must be prepared to receive it (even though the text may be ignored).

RTCP APP: Application-Defined RTCP Packets

The final class of RTCP packet (APP) allows for application-defined extensions. It has packet type 204, and the format shown in Figure 5.11. The *application-defined packet name* is a four-character prefix intended to uniquely identify this extension, with each character being chosen from the ASCII character set, and uppercase and lowercase characters being treated as distinct. It is recommended that the packet name be chosen to match the application it represents, with the choice of subtype values being coordinated by the application. The remainder of the packet is application-specific.

Figure 5.11. Format of an RTCP APP Packet



Application-defined packets are used for nonstandard extensions to RTCP, and for experimentation with new features. The intent is that experimenters use APP as a first place to try new features, and then register new packet types if the features have wider use. Several applications generate APP packets, and implementations should be prepared to ignore unrecognized APP packets.

Packing Issues

As noted earlier, RTCP packets are never sent individually, but rather are packed into a compound packet for transmission. Various rules govern the structure of compound packets, as detailed next.

If the participant generating the compound RTCP packet is an active data sender, the compound must start with an RTCP SR packet. Otherwise it must start with an RTCP RR packet. This is true even if no data has been sent or received, in which case the SR/RR packet contains no receiver report blocks (the RC header field is zero). On the other hand, if data is received from many sources and there are too many reports to fit into a single SR/RR packet, the compound should begin with an SR/RR packet followed by several RR packets.

Following the SR/RR packet is an SDES packet. This packet must include a CNAME item, and it may include other items. The frequency of inclusion of the other (non-CNAME) SDES items is determined by the RTP profile in use. For example, the audio/video profile[7] specifies that other items may be included with every third compound RTCP packet sent, with a NAME item being sent seven out of eight times within that slot and the remaining SDES cyclically taking up the eighth slot. Other profiles may specify different choices.

BYE packets, when ready for transmission, must be placed as the last packet in a compound. Other RTCP packets to be sent may be included in any order. These strict ordering rules are intended to make packet validation easier because it is highly unlikely that a misdirected packet will meet these constraints.

A potentially difficult issue in the generation of compound RTCP packets is how to handle sessions with larger numbers of active senders. If there are more than 31 active senders, it is necessary to include additional RR packets within the compound. This may be repeated as often as is required, up to the maximum transmission unit (MTU) of the network. If there are so many senders that the receiver reports cannot all fit within the MTU, the receiver reports for some senders must be omitted. In that case, reports that are omitted should be included in the next compound packet generated (requiring a receiver to keep track of the sources reported on in each interval).

A similar issue arises when the SDES items to be included within the packet exceed the maximum packet size. The trade-off between including additional receiver reports and including source description information is left to the implementation. There is no single correct solution.

Sometimes it is necessary to pad a compound RTCP packet out beyond its natural size. In such cases the padding is added to the last RTCP packet in the compound only, and the P bit is set in that last packet. Padding is an area where some implementations are incorrect; the section titled Packet Validation later in this chapter discusses common problems.

Security and Privacy

Various privacy issues are inherent in the use of RTCP—in particular, source description packets. Although these packets are optional, their use can expose significant personal details, so applications should not send SDES information without first informing the user that the information is being made available.

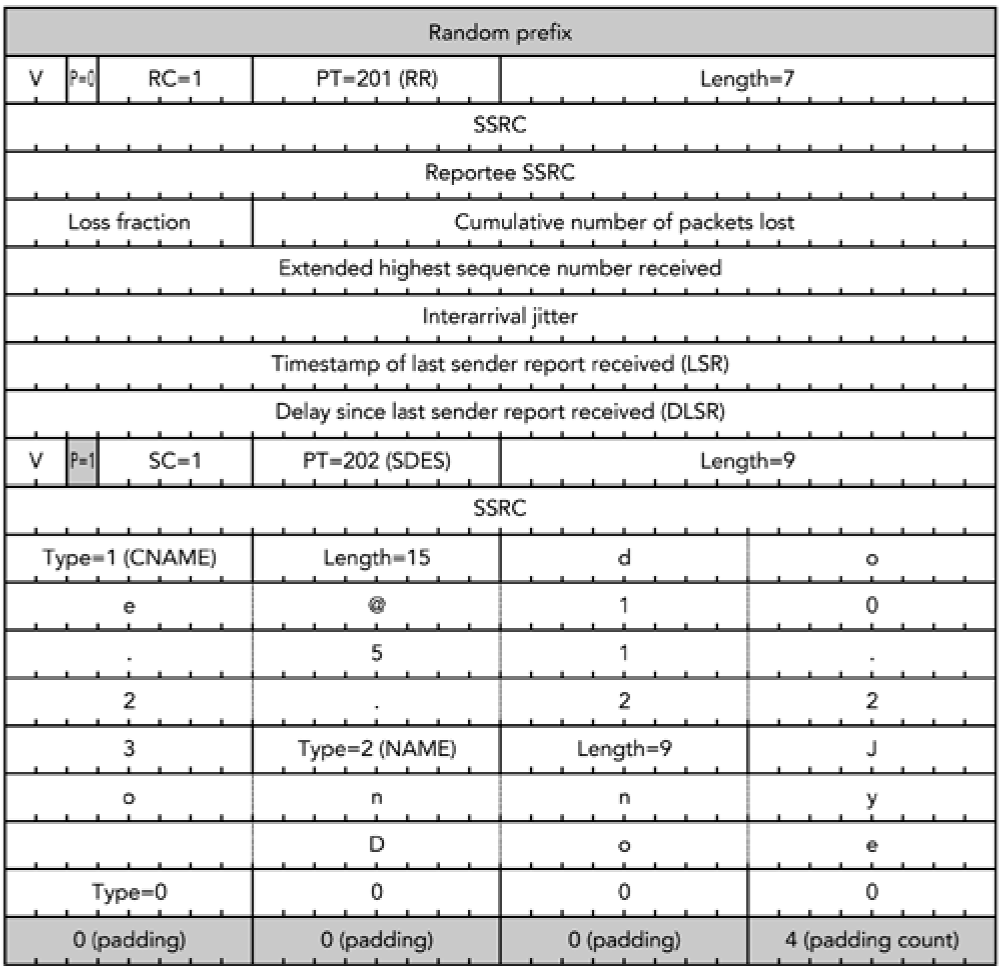
The use of SDES CNAME packets is an exception because these packets are mandatory. The inclusion of an IP address within CNAME packets is a potential issue. However, the same information is available from the IP header of the packet. If the RTP packets pass through Network Address Translation (NAT), the translation of the address in the IP header that is performed should also be performed on the address in the CNAME. In practice, many NAT implementations are unaware of RTP, so there is a potential for leakage of the internal IP address.

The exposure of user names may be a greater concern—in which case applications may omit or rewrite the user name, provided that this is done consistently among the set of applications using CNAME for association.

Some receivers may not want their presence to be visible. It is acceptable if those receivers do not send RTCP at all, although doing so prevents senders from using the reception quality information to adapt their transmission to match the receivers.

To achieve confidentiality of the media stream, RTCP packets may be encrypted. When encrypted, each compound packet contains an additional 32-bit random prefix, as illustrated in Figure 5.12, to help avoid plain-text attacks.

Figure 5.12. Example of an Encrypted RTCP Packet, Showing the Correct Use of Padding



Security and privacy are discussed in more detail in Chapter 13, Security Considerations.

Packet Validation

It is important to validate whether received packets really are RTP or RTCP. The packing rules, mentioned earlier, allow RTCP packets to be rigorously validated.

Successful validation of an RTCP stream gives high assurance that the corresponding RTP stream is also valid, although it does not negate the need for validation of the RTP packets.

Listing 5.1 shows the pseudocode for the validation process. These are the key points:

- All packets must be compound RTCP packets.
- The version field of all packets must equal 2.
- The packet type field of the first RTCP packet in a compound packet must be equal to SR or RR.
- If padding is needed, it is added to only the last packet in the compound. The padding bit should be zero for all other packets in the compound RTCP packet.
- The length fields of the individual RTCP packets must total the overall length of the compound RTCP packet as received.

Because new RTCP packet types may be defined by future profiles, the validation procedure should *not* require each packet type to be one of the five defined in the RTP specification.

Listing 5.1 Pseudocode for Packet Validation

```
validate_rtcp(rtcp_t *packet, int length)

    rtcp_t    *end  = (rtcp_t *) (((char *) packet) + length);
    rtcp_t    *r    = packet;
    int        l    = 0;
    int        p    = 0;
    // All RTCP packets must be compound packets
    if ((packet->length+ 1) * 4) == length) {
        ... error: not a compound packet
    }
    // Check the RTCP version, packet type, and padding of the first
    // in the compound RTCP packet...
    if (packet->version != 2) {
        ...error: version number != 2 in the first subpacket
    }
    if (packet-> p != 0) {
        ...error: padding bit is set on first packet in compound
    }
    if ((packet->pt != RTCP_SR) && (packet->pt != RTCP_RR)) {
        ...error: compound packet does not start with SR or RR
    }
    // Check all following parts of the compound RTCP packet. The RTP
    // version number must be 2, and the padding bit must be zero on
    // all except the last packet.
    do {
       if (p == 1) {
          ...error: padding before last packet in compound
       }
       if (r-> p) {
          p = 1;
       }
       if (r-> version != 2) {
          ...error: version number != 2 in subpacket
       }
       l += (r->length + 1) * 4;
       r  = (rtcp_t *) (((uint32_t *) r) + r->length + 1);
    } while (r < end);

    // Check that the length of the packets matches the length of the
    // UDP packet in which they were received...
    if ((l != length) || (r != end)) {
       ...error: length does not match UDP packet length
    }
      ...packet is valid
}
```

One common implementation problem causes packets to fail their validity test: When you're padding compound RTCP packets beyond their natural length, you need to ensure that the padding is added to only the last packet in the compound. A common mistake has been to add the padding to the last packet, but to set the P bit in the header of the first packet in the compound. The P bit must be set only in the *last* packet.

It is possible to detect RTCP misdirected onto the RTP port via the packet type field. The standard RTCP packets have packet type values with the high bit set; if they are misdirected onto the RTP port, the high bit of the packet type field will fall into the place of the M bit in the RTP header. With the top bit stripped, the standard RTCP packet types correspond to an RTP payload type in the range 72 to 76. This range is reserved in the RTP specification and will not be used for valid RTP data packets, so detection of packets in this range implies that the stream is misdirected. Similarly, RTP packets sent to the RTCP port may clearly be distinguished by their packet type, which will be outside the valid range for RTCP packet types.

Participant Database

Each application in an RTP session will maintain a database of information about the participants and about the session itself. The session information, from which the RTCP timing is derived, can be stored as a set of variables:

- The RTP bandwidth—that is, the typical session bandwidth, configured when the application starts.
- The RTCP bandwidth fraction—that is, the percentage of the RTP bandwidth devoted to RTCP reports. This is usually 5%, but profiles may define a means of changing this (0% also may be used, meaning that RTCP is not sent).
- The average size of all RTCP packets sent and received by this participant.
- The number of members in the session, the number of members when this participant last sent an RTCP packet, and the fraction of those who have sent RTP data packets during the preceding reporting interval.
- The time at which the implementation last sent an RTCP packet, and the next scheduled transmission time.
- A flag indicating whether the implementation has sent any RTP data packets since sending the last two RTCP packets.
- A flag indicating whether the implementation has sent any RTCP packets at all.

In addition, the implementation needs to maintain variables to include in RTCP SR packets:

- The number of packets and octets of RTP data it has sent.
- The last sequence number it used.
- The correspondence between the RTP clock it is using and an NTP-format timestamp.

A session data structure containing these variables is also a good place to store the SSRC being used, the SDES information for the implementation, and the file descriptors for the RTP and RTCP sockets. Finally, the session data structure should contain a database for information held on each participant.

In terms of implementation, the session data can be stored simply: a single structure in a C-based implementation, a class in an object-oriented system. With the exception of the participant-specific data, each variable in the structure or class is a simple type: integer, text string, and so on. The format of the participant-specific data is described next.

To generate RTCP packets properly, each participant also needs to maintain state for the other members in the session. A good design makes the participant database an integral part of the operation of the system, holding not just RTCP-related information, but all state for each participant. The per-participant data structure may include the following:

- SSRC identifier.
- Source description information: the CNAME is required; other information may be included (note that these values are not null-terminated, and care must be taken in their handling).
- Reception quality statistics (packet loss and jitter), to allow generation of RTCP RR packets.
- Information received from sender reports, to allow lip synchronization (see Chapter 7).
- The last time this participant was heard from so that inactive participants can be timed out.
- A flag indicating whether this participant has sent data within the current RTCP reporting interval.
- The media playout buffer, and any codec state needed (see Chapter 6, Media Capture, Playout, and Timing).
- Any information needed for channel coding and error recovery—for example, data awaiting reception of repair packets before it can be decoded (see Chapters 8, Error Concealment, and 9, Error Correction).

Within an RTP session, members are identified by their synchronization source identifier. Because there may be many participants and they may need to be accessed in any order, the appropriate data structure for the participant database is a hash table, indexed by SSRC identifier. In applications that deal with only a single media format, this is sufficient. However, lip synchronization also requires the capability to look up sources by their CNAME. As a result, the participant database should be indexed by a double hash table: once by SSRC and once by CNAME.

Some implementations use less-than-perfect random number generators when choosing their SSRC identifier. This means that a simple hashing function—for example, using the lowest few bits of the SSRC as an index into a table—can lead to unbalanced and inefficient operation. Even though SSRC values are supposed to be random, they should be used with an efficient hashing function. Some have suggested using the MD5 hash of the SSRC as the basis for the index, although that may be considered overkill.

Participants should be added to the database after a validated packet has been received from them. The validation step is important: An implementation does not want to create a state for a participant unless it is certain that the participant is valid. Here are some guidelines:

- If an RTCP packet is received and validated, the participant should be entered into the database. The validity checks on RTCP are strong, and it is difficult for bogus packets to satisfy them.
- An entry should not be made on the basis of RTP packets only, unless multiple packets are received with consecutive sequence numbers. The validity checks possible for a single RTP packet are weak, and it is possible for a bogus packet to satisfy the tests yet be invalid.

This implies that the implementation should maintain an additional, lightweight table of probationary sources (sources in which only a single RTP packet has been received). To prevent bogus sources of RTP and RTCP data from using too much memory, this table should be aggressively timed out and should have a fixed maximum size. It is difficult to protect against an attacker who purposely generates many different sources to use up all memory of the receivers, but these precautions will prevent accidental exhaustion of memory if a misdirected non-RTP stream is received.

Each CSRC (contributing source) in a valid RTP packet also counts as a participant and should be added to the database. You should expect to receive SDES information for participants identified only by CSRC.

When a participant is added to the database, an application should also update the session-level count of the members and the sender fraction. Addition of a participant may also cause RTCP forward reconsideration, which will be discussed shortly.

Participants are removed from the database after a BYE packet is received or after a specified period of inactivity. This sounds simple, but there are several subtle points.

There is no guarantee that packets are received in order, so an RTCP BYE may be received before the last data packet from a source. To prevent state from being torn down and then immediately reestablished, a participant should be marked as having left after a BYE is received, and its state should be held over for a few seconds (my implementation uses a fixed two-second delay). The important point is that the delay is longer than both the maximum expected reordering and the media playout delay, thereby allowing for late packets and for any data in the playout buffer for that participant to be used.

Sources may be timed out if they haven't been heard from for more than five times the reporting interval. If the reporting interval is less than 5 seconds, the 5-second minimum is used here (even if a smaller interval is used when RTCP packets are being sent).

When a BYE packet is received or when a member times out, RTCP reverse reconsideration takes place, as described in the section titled BYE Reconsideration later in this chapter.

Timing Rules

The rate at which each participant sends RTCP packets is not fixed but varies according to the size of the session and the format of the media stream. The aim is to restrict the total amount of RTCP traffic to a fixed fraction—usually 5%—of the session bandwidth. This goal is achieved by a reduction in the rate at which each participant sends RTCP packets as the size of the session increases. In a two-party telephone call using RTP, each participant will send an RTCP report every few seconds; in a session with thousands of participants—for example, an Internet radio station—the interval between RTCP reports from each listener may be many minutes.

Each participant decides when to send RTCP packets on the basis of the set of rules described later in this section. It is important to follow these rules closely, especially for implementations that may be used in large sessions. If implemented correctly, RTCP will scale to sessions with many thousands of members. If not, the amount of control traffic will grow linearly with the number of members and will cause significant network congestion.

Reporting Interval

Compound RTCP packets are sent periodically, according to a randomized timer. The average time each participant waits between sending RTCP packets is known as the reporting interval. It is calculated on the basis of several factors:

- **The bandwidth allocated to RTCP**. This is a fixed fraction—usually 5%—of the session bandwidth. The session bandwidth is the expected data rate for the session; typically this is the bit rate of a single stream of audio or video data, multiplied by the typical number of simultaneous senders. The session bandwidth is fixed for the duration of a session, and supplied as a configuration parameter to the RTP application when it starts.

  The fraction of the session bandwidth allocated to RTCP can be varied by the RTP profile in use. It is important that all members of a session use the same fraction; otherwise state for some members may be prematurely timed out.

- **The average size of RTCP packets sent and received**. The average size includes not just the RTCP data, but also the UDP and IP header sizes (that is, add 28 octets per packet for a typical IPv4 implementation).

- **The total number of participants and the fraction of those participants who are senders**. This requires an implementation to maintain a database of all participants, noting whether they are senders (that is, if RTP data packets or RTCP SR packets have been received from them) or receivers (if only RTCP RR, SDES, or APP packets have been received). The earlier section titled Participant Database explained this in detail.

  To guard against buggy implementations that might send SR packets when they have not sent data, a participant that does listen for data should consider another participant to be a sender only if data packets have been received. An implementation that only sends data and does not listen for others' data (such as a media server) may use RTCP SR packets as an indication of a sender, but it should verify that the packet and byte count fields are nonzero and changing from one SR to the next.

If the number of senders is greater than zero but less than one-quarter of the total number of participants, the reporting interval depends on whether we are sending. If we are sending, the reporting interval is set to the number of senders multiplied by the average size of RTCP packets, divided by 25% of the desired RTCP bandwidth. If we are not sending, the reporting interval is set to the number of receivers multiplied by the average size of RTCP packets, divided by 75% of the desired RTCP bandwidth:

```
If ((senders > 0) and (senders < (25% of total number of participants))
{
    If (we are sending) {
        Interval = average RTCP size * senders / (25% of RTCP bandwidth)
    } else {
        Interval = average RTCP size * receivers / (75% of RTCP bandwidth)
    }
}
```

If there are no senders, or if more than one-quarter of the members are senders, the reporting interval is calculated as the average size of the RTCP packets multiplied by the total number of members, divided by the desired RTCP bandwidth:

```
if ((senders = 0) or (senders > (25% of total number of participants))
{
  Interval = average RTCP size * total number of members / RTCP bandwidth
}
```

These rules ensure that senders have a significant fraction of the RTCP bandwidth, sharing at least one-quarter of the total RTCP bandwidth. The RTCP packets required for lip synchronization and identification of senders can therefore be sent comparatively quickly, while still allowing reports from receivers.

The resulting interval is always compared to an absolute minimum value, which by default is chosen to be 5 seconds. If the interval is less than the minimum interval, it is set to the minimum:

```
If (Interval < minimum interval) {
   Interval = minimum interval
}
```

In some cases it is desirable to send RTCP more often than the default minimum interval. For example, if the data rate is high and the application demands more timely reception quality statistics, a short default interval will be required. The latest revision of the RTP specification allows for a reduced minimum interval in these cases:

```
Minimum interval = 360 / (session bandwidth in Kbps)
```

This reduced minimum is smaller than 5 seconds for session bandwidths greater than 72 Kbps. When the reduced minimum is being used, it is important to remember that some participants may still be using the default value of 5 seconds, and to take this into account when determining whether to time out a participant because of inactivity.

The resulting interval is the average time between RTCP packets. The transmission rules described next are then used to convert this value into the actual send time for each packet. The reporting interval should be recalculated whenever the number of participants in a session changes, or when the fraction of senders changes.

Basic Transmission Rules

When an application starts, the first RTCP packet is scheduled for transmission on the basis of an initial estimate of the reporting interval. When the first packet is sent, the second packet is scheduled, and so on. The actual time between packets is randomized, between one-half and one and a half times the reporting interval, to avoid synchronization of the participants' reports, which could cause them to arrive all at once, every time. Finally, if this is the first RTCP packet sent, the interval is halved to provide faster feedback that a new member has joined, thereby allowing the next send time to be calculated as shown here:

```
I = (Interval * random[0.5, 1.5])

if (this is the first RTCP packet we are sending) {
    I *= 0.5
}
next_rtcp_send_time = current_time + I
```

The routine random[0.5, 1.5] generates a random number in the interval 0.5 to 1.5. On some platforms it may be implemented by the rand() system call; on others, a call such as drand48() may be a better source of randomness.

As an example of the basic transmission rules, consider an Internet radio station sending 128-Kbps MP3 audio using RTP-over-IP multicast, with an audience of 1,000 members. The default values for the minimum reporting interval (5 seconds) and RTCP bandwidth fraction (5%) are used, and the average size of RTCP packets is assumed to be 90 octets (including UDP/IP headers). When a new audience member starts up, it will not be aware of the other listeners, because it has not yet received any RTCP data. It must assume that the only other member is the sender and calculate its initial reporting interval accordingly. The fraction of members who are senders (the single source) is more than 25% of the known membership (the source and this one receiver), so the reporting interval is calculated like this:

```
Interval = average RTCP size * total number of members / RTCP bandwidth
       =   90 octets      *        2             / (5% of 128 Kbps)
       =   180 octets     /  800 octets per second
       =   0.225 seconds
```

Because 0.225 seconds is less than the minimum, the minimum interval of 5 seconds is used as the interval. This value is then randomized and halved because this is the first RTCP packet to be sent. Thus the first RTCP packet is sent between 1.25 and 3.75 seconds after the application is started.

During the time between starting the application and sending the first RTCP packet, several receiver reports will have been received from the other members of the session, allowing the implementation to update its estimate of the number of members. This updated estimate is used to schedule the second RTCP packet.

As we will see later, 1,000 listeners is enough that the average interval will be greater than the minimum, so the rate at which RTCP packets are received in aggregate from all listeners is 75% x 800 bytes per second ÷ 90 bytes per packet = 6.66 packets per second. If the application sends its first RTCP packet after, say, 2.86 seconds, the known audience size will be approximately 2.86 seconds x 6.66 per second = 19.

Because the fraction of senders is now less than 25% of the known membership, the reporting interval for the second packet is then calculated in this way:

```
Interval = receivers * average RTCP size / (75% of RTCP bandwidth)
```

```
         =    19    *       90        / (75% of (5% of 128 Kbps))
         =        1710        / (0.75 * (0.05 * 16000 octets/second))
         =        1710        / 600
         =          2.85 seconds
```

Again, this value is increased to the minimum interval and randomized. The second RTCP packet is sent between 2.5 and 7.5 seconds after the first.

The process repeats, with an average of 33 new receivers being heard from between sending the first and second RTCP packets, for a total known membership of 52. The result will be an average interval of 7.8 seconds, which, because it is greater than the minimum, is used directly. Consequently the third packet is sent between 3.9 and 11.7 seconds after the second. The average interval between packets increases as the other receivers become known, until the complete audience has been heard from. The interval is then calculated in this way:

```
Interval = receivers * average RTCP size / (75% of RTCP bandwidth)
         =    1000   *     90          / (75% of (5% of 128 Kbps))
         =        90000       / (0.75 * (0.05 * 16000 octets/second))
         =        90000       / 600
         =          150 seconds
```

An interval of 150 seconds is equivalent to 1/150 = 0.0066 packets per second, which with 1,000 listeners gives the average RTCP reception rate of 6.66 packets per second.

The proposed standard version of RTP[6] uses only these basic transmission rules. Although these are sufficient for many applications, they have some limitations that cause problems in sessions with rapid changes in membership. The concept of reconsideration was introduced to avoid these problems.

Forward Reconsideration

As the preceding section suggested, when the session is large, it takes a certain number of reporting intervals before a new member knows the total size of the session. During this learning period, the new member is sending packets faster than the "correct" rate, because of incomplete knowledge. This issue becomes acute when many members join at once, a situation known as a *step join*. A typical scenario in which a step join may occur is at the start of an event, when an application starts automatically for many participants at once.

In the case of a step join, if only the basic transmission rules are used, each participant will join and schedule its first RTCP packet on the basis of an initial estimate of zero participants. It will send that packet after an average of half of the minimum interval, and it will schedule the next RTCP packet on the basis of the observed number of participants at that time, which can now be several hundreds or even thousands. Because of the low initial estimate for the size of the group, there is a burst of RTCP traffic when all participants join the session, and this can congest the network.

Rosenberg has studied this phenomenon[100] and reports on the case in which 10,000 members join a session at once. His simulations show that in such a step join, all 10,000 members try to send an RTCP packet within the first 2.5 seconds, which is almost 3,000 times the desired rate. Such a burst of packets will cause extreme network congestion—not the desired outcome for a low-rate control protocol.

Continually updating the estimate of the number of participants and the fraction who are senders, and then using these numbers to reconsider the send time of each RTCP packet, can solve this problem. When the scheduled transmission time arrives, the interval is recalculated on the basis of the updated estimate of the group size, and this value is used to calculate a new send time. If the new send time is in the future, the packet is not sent but is rescheduled for that time.

This procedure may sound complex, but it is actually simple to implement. Consider the pseudocode for the basic transmission rules, which can be written like this:

```
if (current_time >= next_rtcp_send_time) {
    send RTCP packet
    next_rtcp_send_time = rtcp_interval() + current_time
}
```

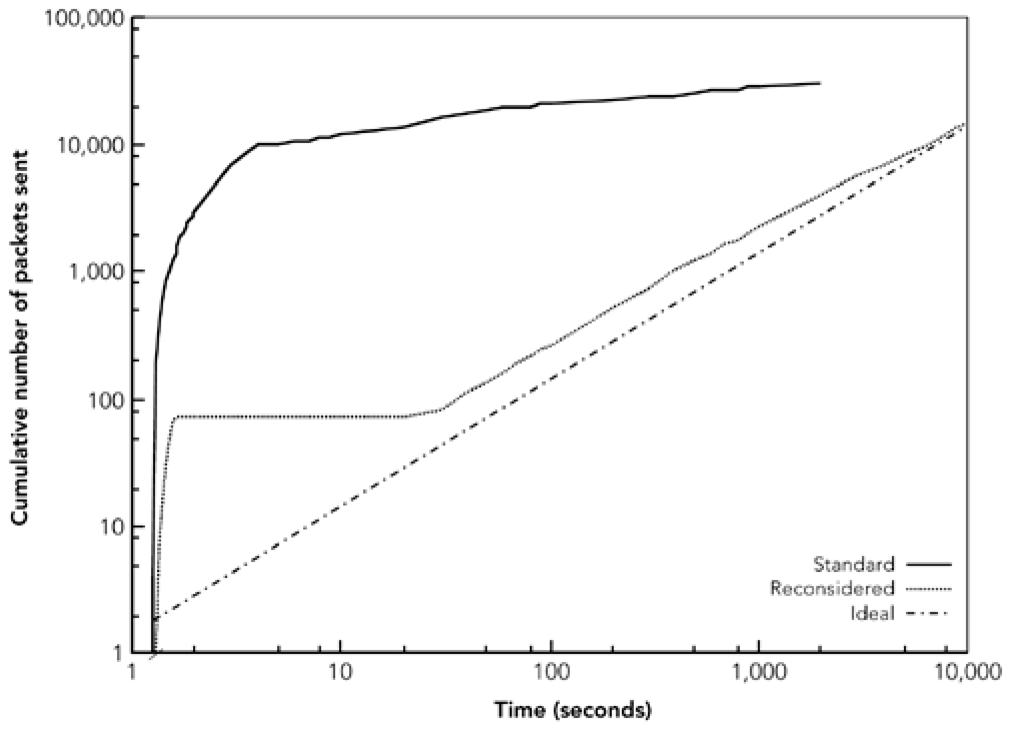
With forward reconsideration, this changes to the following:

```
if (current_time >= next_rtcp_check_time) {
  new_rtcp_send_time = (rtcp_interval() / 1.21828) + last_rtcp_send_time
  if (current_time >= new_rtcp_send_time) {
    send RTCP packet
    next_rtcp_check_time = (rtcp_interval() /1.21828) + current_time
  } else {
    next_rtcp_check_time = new_send_time
  }
}
```

Here the function `rtcp_interval()` returns a randomized sampling of the reporting interval, based on the current estimate of the session size. Note the division of `rtcp_interval()` by a factor of 1.21828 (Euler's constant `e` minus 1.5). This is a compensating factor for the effects of the reconsideration algorithm, which converges to a value below the desired 5% bandwidth fraction.

The effect of reconsideration is to delay RTCP packets when the estimate of the group size is increasing. This effect is shown in Figure 5.13, which illustrates that the initial burst of packets is greatly reduced when reconsideration is used, comprising only 75 packets—rather than 10,000—before the other participants learn to scale back their reporting interval.

Figure 5.13. The Effect of Forward Reconsideration on RTCP Send Rates (Adapted from J. Rosenberg and H. Schulzrinne, "Timer Reconsideration for Enhanced RTP Scalability," Proceedings of IEEE Infocom '98, San Francisco, CA, March 1998. © 1998 IEEE.)



As another example, consider the scenario discussed in the previous section, Basic Transmission Rules, in which a new listener is joining an established Internet radio station using multicast RTP. When the listener is joining the session, the first RTCP packet is scheduled as before, between 1.25 and 3.75 seconds after the application is started. The difference comes when the scheduled transmission time arrives: Rather than sending the packet, the application reconsiders the schedule on the basis of the current estimate of the number of members. As was calculated before, assuming a random initial interval of 2.86 seconds, the application will have received about 19 RTCP packets from the other members, and a new average interval of 2.85 seconds will be calculated:

```
Interval = number of receivers * average RTCP size / (75% of RTCP bandwidth)
         =   19  *  90 / (0.75  * (0.05 * 16000 octets/second))
        = 1710/ 600
        = 2.85 seconds
```

The result is less than the minimum, so the minimum of 5 seconds is used, randomized and divided by the scaling factor. If the resulting value is less than the current time (in this example 2.85 seconds after the application started), then the packet is sent. If not—for example, if the new randomized value is 5.97 seconds—the packet is rescheduled for the later time.

After the new timer expires (in this example 5.97 seconds after the application started), the reconsideration process takes place again. At this time the receiver will have received RTCP packets from approximately 5.97 seconds x 6.66 per second = 40 other members, and the recalculated RTCP interval will be 6 seconds before randomization and scaling. The process repeats until the reconsidered send time comes out before the current time. At that point the first RTCP packet is sent, and the second is scheduled.

Reconsideration is simple to implement, and it is recommended that all implementations include it, even though it has significant effects only after the number of participants reaches several hundred. An implementation that includes forward reconsideration will be safe no matter what size the session, or how many participants join simultaneously. One that uses only the basic transmission rules may send RTCP too often, causing network congestion in large sessions with synchronized joins.

Reverse Reconsideration

If there are problems with step joins, one might reasonably expect there to be problems due to the rapid departure of many participants (a *step leave*). This is indeed the case with the basic transmission rules, although the problem is not with RTCP being sent too often and causing congestion, but with it not being sent often enough, causing premature timeout of participants.

The problem occurs when most, but not all, of the members leave a large session. As a result the reporting interval decreases rapidly, perhaps from several minutes to several seconds. With the basic transmission rules, however, packets are not rescheduled after the change, although the timeout interval is updated. The result is that those members who did not leave are marked as having timed out; their packets do not arrive within the new timeout period.

The problem is solved in a similar way to that of step joins: When each BYE packet is received, the estimate of the number of participants is updated, and the send time of the next RTCP packet is reconsidered. The difference from forward reconsideration is that the estimate will be getting smaller, so the next packet is sent earlier than it would otherwise have been.

When a BYE packet is received, the new transmission time is calculated on the basis of the fraction of members still present after the BYE, and the amount of time left before the original scheduled transmission time. The procedure is as follows:

```
if (BYE packet received) {
  member_fraction = num_members_after_BYE / num_members_before_BYE
  time_remaining  = next_rtcp_send_time – current_time
  next_rtcp_send_time = current_time + member_fraction * time_remaining
}
```

The result is a new transmission time that is earlier than the original value, but later than the current time. Packets are therefore scheduled early enough that the remaining members do not time each other out, preventing the estimate of the number of participants from erroneously falling to zero.

In practice, reverse reconsideration significantly reduces the problems due to premature timeout but does not completely solve them. In some cases the estimate of the group membership may drop to zero for a brief period after a step leave, but will rapidly return to the correct value. During the development of reconsideration, the additional complexity to completely fix this problem outweighed the gain.

Implementation of reverse reconsideration is a secondary concern: It's an issue only in sessions with several hundred participants and rapid changes in membership, and failing to implement it may result in false timeouts but no networkwide problems.

BYE Reconsideration

In the proposed standard version of RTP,[6] a member desiring to leave a session sends a BYE packet immediately, then exits. If many members decide to leave at once, this can cause a flood of BYE packets and can result in network congestion (much as happens with RTCP packets during a step join, if forward reconsideration is not employed).

To avoid this problem, the current version of RTP allows BYE packets to be sent immediately only if there are fewer than 50 members when a participant decides to leave. If there are more than 50 members, the leaving member should delay sending a BYE if other BYE packets are received while it is preparing to leave, a process called *BYE reconsideration*.

BYE reconsideration is analogous to forward reconsideration, but based on a count of the number of BYE packets received, rather than the number of other members. When a participant wants to leave a session, it suspends normal processing of RTP/RTCP packets and schedules a BYE packet according to the forward reconsideration rules, calculated as if there were no other members and as if this were the first RTCP packet to be sent. While waiting for the scheduled transmission time, the participant ignores all RTP and RTCP packets except for BYE packets. The BYE packets received are counted, and when the scheduled BYE transmission time arrives, it is reconsidered on the basis of this count. The process continues until the BYE is sent, and then the participant leaves the session.

As this description suggests, the delay before a BYE can be sent depends on the number of members leaving. If only a single member decides to leave, the BYE will be delayed between 1.026 and 3.078 seconds (based on a 5-second minimum reporting interval, halved because BYE packets are treated as if they're the initial RTCP packet). If many participants decide to leave at once, there may be a considerable delay between deciding to leave a session and being able to send the BYE packet. If a fast exit is needed, it is safe to leave the session without sending a BYE; other participants will time out their state eventually.

The use of BYE reconsideration is a relatively minor decision: It is useful only when many participants leave a session at once, and when the others care about receiving notification that a participant has left. It is safe to leave large sessions without sending a BYE, rather than implementing the BYE reconsideration algorithm.

Comments on Reconsideration

The reconsideration rules were introduced to allow RTCP to scale to very large sessions in which the membership changes rapidly. I recommend that all implementations include reconsideration, even if they are initially intended only for use in small sessions; this will prevent future problems if the tool is used in a way the designer did not foresee.

On first reading, the reconsideration rules appear complex and difficult to implement. In practice, they add a small amount of additional code. My implementation of RTP and RTCP consists of about 2,500 lines of C code (excluding sockets and encryption code). Forward and reverse reconsideration together add only 15 lines of code. BYE reconsideration is more complex, at 33 lines of code, but still not a major source of difficulty.

Correct operation of the reconsideration rules depends to a large extent on the statistical average of the behavior of many individual participants. A single incorrect implementation in a large session will cause little noticeable difference to the behavior, but many incorrect implementations in a single session can lead to significant congestion problems. For small sessions, this is largely a theoretical problem, but as the session size increases, the effects of bad RTCP implementations are magnified and can cause network congestion that will affect the quality of the audio and/or video.

Common Implementation Problems

The most common problems observed with RTCP implementations relate to the basic transmission rules, and to the bandwidth calculation:

- Incorrect scaling with the number of participants. A fixed reporting interval will cause traffic to grow linearly with the number of members, eventually far exceeding the amount of audio/video data sent and causing network congestion.
- Lack of randomization of the reporting interval. Implementations that use a nonrandom reporting interval have the potential to unintentionally synchronize their reports, causing bursts of RTCP packets that can overwhelm receivers.
- Forgetting to include lower-layer overheads in the bandwidth calculations. All packet sizes, when calculating the reporting interval, should include the IP and UDP headers (28 octets, for a typical IPv4-based implementation).
- Incorrect use of padding. If padding is needed, it should be added to only the last packet in a compound RTCP packet.

When testing the behavior of an RTCP implementation, it is important to use a range of scenarios. Problems can be found in tests of both large and small sessions, sessions in which the membership changes rapidly, sessions in which a large fraction of the participants are senders and in which few are senders, and sessions in which step joins and leaves occur. Testing large-scale sessions is inherently difficult. If an implementation can be structured to be independent of the underlying network transport system, it will allow the simulation of large sessions on a single test machine.

The IETF audio/video transport working group has produced a document describing testing strategies for RTP implementations,[40] which may also be useful.

Summary

This chapter has described the RTP control protocol, RTCP, in some detail. There are three components:

1. The RTCP packet formats, and the means by which compound packets are generated
2. The participant database as the main data structure for an RTP-based application, and the information that needs to be stored for correct operation of RTCP
3. The rules governing timing of RTCP packets: periodic transmission, adaptation to the size of the session, and reconsideration

We have also briefly discussed security and privacy issues, which are discussed in depth in Chapter 13, Security Considerations, as well as the correct validation of RTCP packets.

The RTP control protocol is an integral part of RTP, used for reception quality reporting, source description, membership control, and lip synchronization. Correct implementation of RTCP can significantly enhance an RTP session: It permits the receiver to lipsync audio and video, identifies the other members of a session, and allows the sender to make an informed choice of error protection scheme to use to achieve optimum quality.

Chapter 6. Media Capture, Playout, and Timing

In this chapter we move on from our discussion of networks and protocols, and talk instead about the design of systems that use RTP. An RTP implementation has multiple aspects, some of which are required for all applications; others are optional depending on the needs of the application. This chapter discusses the most fundamental features necessary for media capture, playout, and timing recovery;

later chapters describe ways in which reception quality can be improved or overheads reduced.

We start with a discussion of the behavior of a sender: media capture and compression, generation of RTP packets, and the under-lying media timing model. Then the discussion focuses on the receiver, and the problems of media playout and timing recovery in the face of uncertain delivery conditions. Key to receiver design is the operation of the playout buffer, and much of this chapter is spent on this subject.

It is important to remember that many designs are possible for senders and receivers, with the RTP specification permitting a wide range of implementation choices. The design outlined here is one possible implementation, with a particular set of trade-offs; implementers should use alternative techniques if they are more appropriate to particular scenarios. (Many implementations are described in the literature; for example, McCanne and Jacobson[87] outline the structure of a video conferencing system influential in the design of RTP.)

Behavior of a Sender

As noted in Chapter 1, An Introduction to RTP, a sender is responsible for capturing audiovisual data, whether live or from a file, compressing it for transmission, and generating RTP packets. It may also participate in error correction and congestion control by adapting the transmitted media stream in response to receiver feedback. Figure 1.2 in Chapter 1 shows the process.

The sender starts by reading uncompressed media data—audio samples or video frames—into a buffer from which encoded frames are produced. Frames may be encoded in several ways depending on the compression algorithm used, and encoded frames may depend on both earlier and later data. The next section, Media Capture and Compression, describes this process.

Compressed frames are assigned a timestamp and a sequence number, and loaded into RTP packets ready for transmission. If a frame is too large to fit into a single packet, it may be fragmented into several packets for transmission. If a frame is small, several frames may be bundled into a single RTP packet. The section titled Generating RTP Packets later in this chapter describes these functions, both of which are possible only if supported by the payload format. Depending on the error correction scheme in use, a channel coder may be used to generate error correction packets or to reorder frames for interleaving before transmission (Chapters 8 and 9 discuss error concealment and error correction). The sender will generate periodic status reports, in the form of RTCP packets, for the media streams it is generating. It will also receive reception quality feedback from other participants, and it may use that information to adapt its transmission. RTCP was described in Chapter 5, RTP Control Protocol.

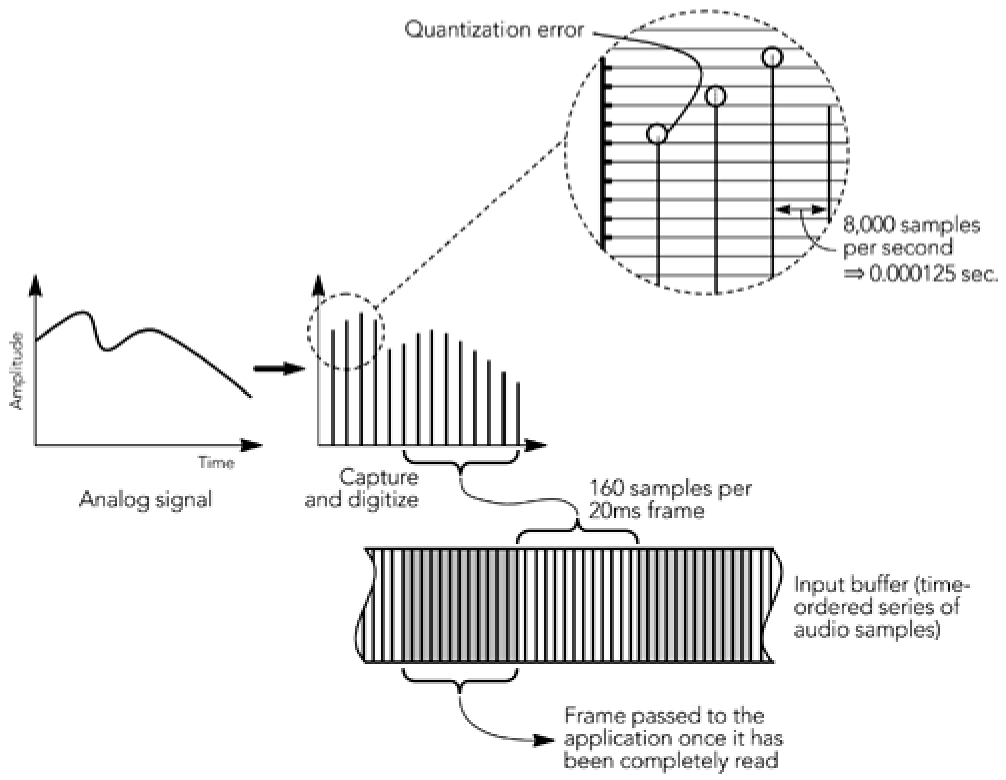
Media Capture and Compression

The media capture process is essentially the same whether audio or video is being transmitted: An uncompressed frame is captured, if necessary it is transformed into a suitable format for compression, and then the encoder is invoked to produce a compressed frame. The compressed frame is then passed to the packetization routine, and one or more RTP packets are generated. Factors specific to audio/video capture are discussed in the next two sections, followed by a description of issues raised by prerecorded content.

Audio Capture and Compression

Considering the specifics of audio capture, Figure 6.1 shows the sampling process on a general-purpose workstation, with sound being captured, digitized, and stored into an audio input buffer. This input buffer is commonly made available to the application after a fixed number of samples have been collected. Most audio capture APIs return data from the input buffer in fixed-duration frames, blocking until sufficient samples have been collected to form a complete frame. This imposes some delay because the first sample in a frame is not made available until the last sample has been collected. If given a choice, applications intended for interactive use should select the buffer size closest to that of the codec frame duration, commonly either 20 milliseconds or 30 milliseconds, to reduce the delay.

Figure 6.1. Audio Capture, Digitization, and Framing


Quantization error
8,000 samples per second ⇒ 0.000125 sec.
Amplitude
Time
Analog signal
Capture and digitize
160 samples per 20ms frame
Input buffer (time-ordered series of audio samples)
Frame passed to the application once it has been completely read

Uncompressed audio frames can be returned from the capture device with a range of sample types and at one of several sampling rates. Common audio capture devices can return samples with 8-, 16-, or 24-bit resolution, using linear, µ-law or A-law quantization, at rates between 8,000 and 96,000 samples per second, and in mono or stereo. Depending on the capabilities of the capture device and on the media codec, it may be necessary to convert the media to an alternative format before the media can be used—for example, changing the sample rate or converting from linear to µ-law quantization. Algorithms for audio format conversion are outside the scope of this book, but standard signal-processing texts give a range of possibilities.

One of the most common audio format conversions is from one sampling rate to another, when the audio capture device samples at one rate, yet the codec requires another rate. (For example, the device may operate at a fixed rate to 44.1kHz to enable high-quality CD playback, yet the desire is to transmit using an 8kHz voice codec.) Sample rate conversion between arbitrary rates is possible but is considerably more efficient and accurate for conversion between rates that are integer multiples of each other. The computational requirements of sample rate conversion should be taken into account when the capture mode for the audio hardware is being selected. Other audio format conversions, such as converting between linear and µ-law quantization are inexpensive and can readily be performed in software.

Captured audio frames are passed to the encoder for compression. Depending on the codec, state may be maintained between frames—the compression context—that must be made available to the encoder along with each new frame of data. Some codecs, particularly music codecs, base their compression on a series of uncompressed frames and not on uncompressed frames in isolation. In these cases the encoder may need to be passed several frames of audio, or it may buffer frames internally and produce output only after receiving several frames. Some codecs produce fixed-size frames as their output; others produce variable-size frames. Those with variable-size frames commonly select from a fixed set of output rates according to the desired quality or signal content; very few are truly variable-rate.

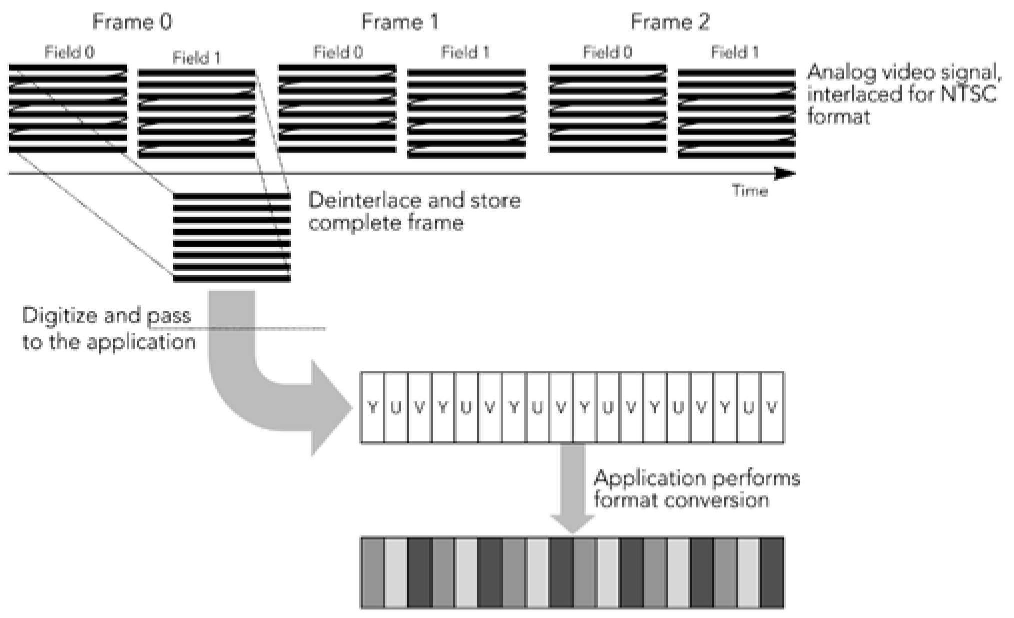
Many speech codecs perform voice activity detection with silence suppression, detecting and suppressing frames that contain only silence or background noise. Suppressed frames either are not transmitted or are replaced with occasional low-rate comfort noise packets. The result can be a significant savings in network capacity, especially if statistical multiplexing is used to make effective use of limited-capacity channels.

Video Capture and Compression

Video capture devices typically operate on complete frames of video, rather than returning individual scan lines or fields of an interlaced image. Many offer the ability to subsample and capture the frame at reduced resolution or to return a subset of the frames. Frames may have a range of sizes, and capture devices may return frames in a variety of formats, color spaces, depths, and subsampling.

Depending on the codec used, it may be necessary to convert from the device format before the frame can be used. Algorithms for such conversion are outside the scope of this book, but any standard video signal–processing text will give a range of possibilities, depending on the desired quality and the available resources. The most commonly implemented conversion is probably between RGB and YUV color spaces; in addition, color dithering and subsampling are often required. These conversions are well suited to acceleration in response to the single-instruction, multiple-data (SIMD) instructions present in many processor architectures (for example, Intel MMX instructions, SPARC VIS instructions). Figure 6.2 illustrates the video capture process, with the example of an NTSC signal captured in YUV format being converted into RGB format before use.

Figure 6.2. Video Capture


Frame 0
Frame 1
Frame 2
Field 0
Field 1
Field 0
Field 1
Field 0
Field 1
Analog video signal, interlaced for NTSC format
Time
Deinterlace and store complete frame
Digitize and pass to the application
Y U V Y U V Y U V Y U V Y U V Y U V
Application performs format conversion

Once video frames have been captured, they are buffered before being passed to the encoder for compression. The amount of buffering depends on the compression scheme being used; most video codecs perform interframe compression, in which each frame depends on the surrounding frames. Interframe compression may require the coder to delay compressing a particular frame until the frames on which it depends have been captured. The encoder will maintain state information between frames, and this information must be made available to the encoder along with the video frames.

For both audio and video, the capture device may directly produce compressed media, rather than having separate capture and compression stages. This is common on special-purpose hardware, but some workstation audiovisual interfaces also have built-in compression. Capture devices that work in this way simplify RTP implementations because they don't need to include a separate codec, but they may limit the scope of adaptation to clock skew and/or network jitter, as described later in this chapter.

No matter what the media type is and how compression is performed, the result of the capture and compression stages is a sequence of compressed frames, each with an associated capture time. These frames are passed to the RTP module, for packetization and transmission, as described in the next section, Generating RTP Packets.

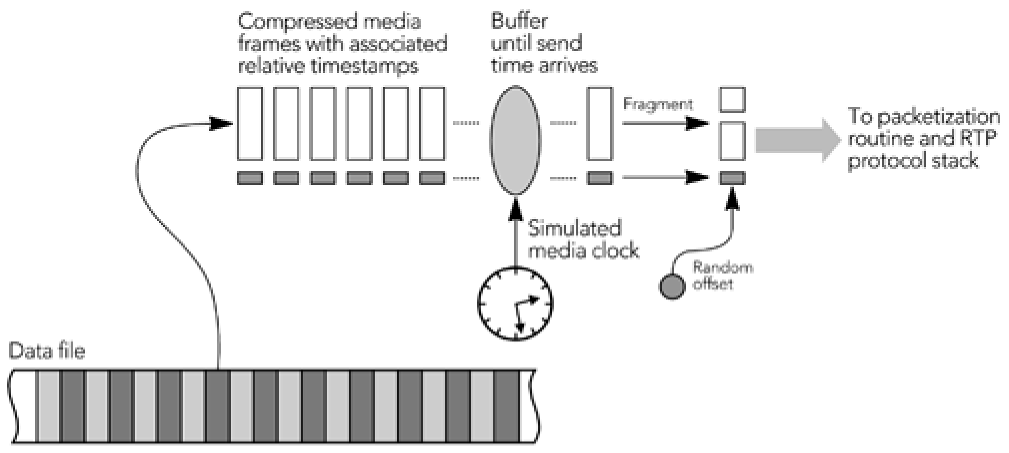
Use of Prerecorded Content

When streaming from a file of prerecorded and compressed content, media frames are passed to the packetization routines in much the same way as for live content. The RTP specification makes no distinction between live and prerecorded media, and senders generate data packets from compressed frames in the same way, no matter how the frames were generated.

In particular, when beginning to stream prerecorded content, the sender must generate a new SSRC and choose random initial values for the RTP timestamp and sequence number. During the streaming process, the sender must be prepared to handle SSRC collisions and should generate and respond to RTCP packets for the stream. Also, if the sender implements a control protocol, such as RTSP,[14] that allows the receiver to pause or seek within the media stream, the sender must keep track of such interactions so that it can insert the correct sequence number and timestamp into RTP data packets (these issues are also discussed in Chapter 4, RTP Data Transfer Protocol).

The need to implement RTCP, and to ensure that the sequence number and timestamp are correct, implies that a sender cannot simply store complete RTP packets in a file and stream directly from the file. Instead, as shown in Figure 6.3, frames of media data must be stored and packetized on the fly.

Figure 6.3. Use of Prerecorded Content


Compressed media frames with associated relative timestamps
Buffer until send time arrives
Fragment
To packetization routine and RTP protocol stack
Simulated media clock
Random offset
Data file

Generating RTP Packets

As compressed frames are generated, they are passed to the RTP packetization routine. Each frame has an associated timestamp, from which the RTP timestamp is derived. If the payload format supports fragmentation, large frames are fragmented to fit within the maximum transmission unit of the network (this is typically needed only for video). Finally, one or more RTP packets are generated for each frame, each including media data and any required payload header. The format of the media packet and payload header is defined according to the payload format specification for the codec used. The critical parts to the packet generation process are assigning timestamps to frames, fragmenting large frames, and generating the payload header. These issues are discussed in more detail in the sections that follow.

In addition to the RTP data packets that directly represent the media frames, the sender may generate error correction packets and may reorder frames before transmission. These processes are described in Chapters 8, Error Concealment, and 9, Error Correction. After the RTP packets have been sent, the buffered media data corresponding to those packets is eventually freed. The sender must not discard data that might be needed for error correction or in the encoding process. This requirement may mean that the sender must buffer data for some time after the corresponding packets have been sent, depending on the codec and error correction scheme used.

Timestamps and the RTP Timing Model

The RTP timestamp represents the sampling instant of the first octet of data in the frame. It starts from a random initial value and increments at a media-dependent rate.

During capture of a live media stream, the sampling instant is simply the time when the media is captured from the video frame grabber or audio sampling device. If the audio and video are to be synchronized, care must be taken to ensure that the processing delay in the different capture devices is accounted for, but otherwise the concept is straightforward. For most audio payload formats, the RTP timestamp increment for each frame is equal to the number of samples—not octets—read from the capture device. A common exception is MPEG audio, including MP3, which uses a 90kHz media clock, for compatibility with other MPEG content. For video, the RTP timestamp is incremented by a nominal per frame value for each frame captured, depending on the clock and frame rate. The majority of video formats use a 90kHz clock because that gives integer timestamp increments for common video formats and frame rates. For example, if sending at the NTSC standard rate of (approximately) 29.97 frames per second using a payload format with a 90kHz clock, the RTP timestamp is incremented by *exactly* 3,003 per packet.

For prerecorded content streamed from a file, the timestamp gives the time of the frame in the playout sequence, plus a constant random offset. As noted in Chapter 4, RTP Data Transfer Protocol, the clock from which the RTP timestamp is derived must increase in a continuous and monotonic fashion irrespective of seek operations or pauses in the presentation. This means that the timestamp does not always correspond to the time offset of the frame from the start of the file; rather it measures the timeline since the start of the playback.

Timestamps are assigned per frame. If a frame is fragmented into multiple RTP packets, each of the packets making up the frame will have the same timestamp.

The RTP specification makes no guarantee as to the resolution, accuracy, or stability of the media clock. The sender is responsible for choosing an appropriate clock, with sufficient accuracy and stability for the chosen application. The receiver knows the nominal clock rate but typically has no other knowledge regarding the precision of the clock. Applications should be robust to variability in the media clock, both at the sender and at the receiver, unless they have specific knowledge to the contrary.

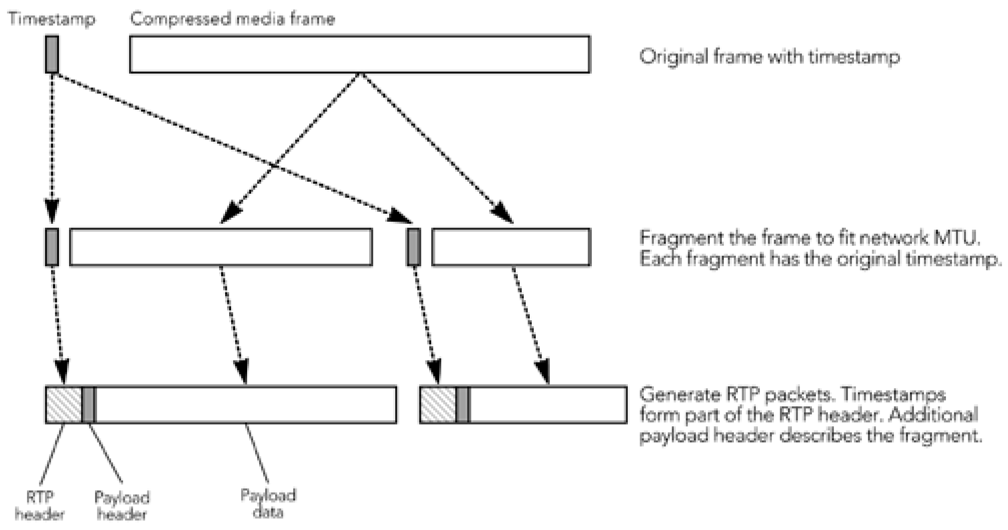
The timestamps in RTP data packets and in RTCP sender reports represent the timing of the media at the sender: the timing of the sampling process, and the relation between the sampling process and a reference clock. A receiver is expected to reconstruct the timing of the media from this information. Note that the RTP timing model says nothing about *when* the media data is to be played out. The timestamps in data packets give the relative timing, and RTCP sender reports provide a reference for interstream synchronization, but RTP says nothing about the amount of buffering that may be needed at the receiver, or about the decoding time of the packets.

Although the timing model is well defined by RTP, the specification makes no mention of the algorithms used to reconstruct the timing at a receiver. This is intentional: The design of playout algorithms depends on the needs of the application and is an area where vendors may differentiate their products.

Fragmentation

Frames that exceed the network maximum transmission unit (MTU) must be fragmented into several RTP packets before transmission, as shown in Figure 6.4. Each fragment has the timestamp of the frame and may have an additional payload header to describe the fragment.

Figure 6.4. Fragmentation of a Media Frame into Several RTP Packets


Timestamp
Compressed media frame
Original frame with timestamp
Fragment the frame to fit network MTU. Each fragment has the original timestamp.
Generate RTP packets. Timestamps form part of the RTP header. Additional payload header describes the fragment.
RTP header
Payload header
Payload data

The fragmentation process is critical to the quality of the media in the presence of packet loss. The ability to decode each fragment independently is desirable; otherwise loss of a single fragment will result in the entire frame being discarded—a loss multiplier effect we wish to avoid. Payload formats that may require fragmentation typically define rules by which the payload data may be split in appropriate places, along with payload headers to help the receiver use the data in the event of some fragments being lost. These rules require support from the encoder to generate fragments that both obey the packing rules of the payload format and fit within the network MTU.

If the encoder cannot produce appropriately sized fragments, the sender may have to use an arbitrary fragmentation. Fragmentation can be accomplished by the application at the RTP layer, or by the network using IP fragmentation. If some fragments of an arbitrarily fragmented frame are lost, it is likely that the entire frame will have to be discarded, significantly impairing quality (Handley and Perkins[33] describe these issues in more detail).

When multiple RTP packets are generated for each frame, the sender must choose between sending the packets in a single burst and spreading their transmission across the framing interval. Sending the packets in a single burst reduces the end-to-end delay but may overwhelm the limited buffering capacity of the network or receiving host. For this reason it is recommended that the sender spread the packets out in time across the framing interval. This issue is important mostly for high-rate senders, but it is good practice for other implementations as well.

Payload Format–Specific Headers

In addition to the RTP header and the media data, packets often contain an additional payload-specific header. This header is defined by the RTP payload format specification in use, and provides an adaptation layer between RTP and the codec output.

Typical use of the payload header is to adapt codecs that were not designed for use over lossy packet networks to work on IP, to otherwise provide error resilience, or to support fragmentation. Well-designed payload headers can greatly enhance the performance of a payload format, and implementers should pay attention in order to correctly generate these headers, and to use the data provided to repair the effects of packet loss at the receiver.

The section titled Payload Headers in Chapter 4, RTP Data Transfer Protocol, discusses the use of payload headers in detail.

Behavior of a Receiver

As highlighted in Chapter 1, An Introduction to RTP, a receiver is responsible for collecting RTP packets from the network, repairing and correcting for any lost packets, recovering the timing, decompressing the media, and presenting the result to the user. In addition, the receiver is expected to send reception quality reports so that the sender can adapt the transmission to match the network characteristics. The receiver will also typically maintain a database of participants in a session to be able to provide the user with information on the other participants. Figure 1.3 in Chapter 1 shows a block diagram of a receiver.

The first step of the reception process is to collect packets from the network, validate them for correctness, and insert them into a per-sender input queue. This is a straightforward operation, independent of the media format. The next section—Packet Reception—describes this process.

The rest of the receiver processing operates in a sender-specific manner and may be media-specific. Packets are removed from their input queue and passed to an optional channel-coding routine to correct for loss (Chapter 9 describes error correction). Following any channel coder, packets are inserted into a source-specific playout buffer, where they remain until complete frames have been received and any variation in interpacket timing caused by the network has been smoothed. The calculation of the amount of delay to add is one of the most critical aspects in the design of an RTP implementation and is explained in the section titled The Playout Buffer later in this chapter. The section titled Adapting the Playout Point describes a related operation: how to adjust the timing without disrupting playout of the media.

Sometime before their playout time is reached, packets are grouped to form complete frames, damaged or missing frames are repaired (Chapter 8, Error Concealment, describes repair algorithms), and frames are decoded. Finally, the media data is rendered for the user. Depending on the media format and output device, it may be possible to play each stream individually—for example, presenting several video streams, each in its own window. Alternatively, it may be necessary to mix the media from all sources into a single stream for playout—for example, combining several audio sources for playout via a single set of speakers. The final section of this chapter—Decoding, Mixing, and Playout—describes these operations.

The operation of an RTP receiver is a complex process, and more involved than the operation of a sender. This increased complexity is largely due to the variability inherent in IP networks: Most of the complexity comes from the need to compensate for lost packets and to recover the timing of a stream.

Packet Reception

An RTP session comprises both data and control flows, running on distinct ports (usually the data packets flow on an even-numbered port, with control packets on the next higher—odd-numbered—port). This means that a receiving application will open two sockets for each session: one for data, one for control. Because RTP runs above UDP/IP, the sockets used are standard `SOCK_DGRAM` sockets, as provided by the Berkeley sockets API on UNIX-like systems, and by Winsock on Microsoft platforms.

Once the receiving sockets have been created, the application should prepare to receive packets from the network and store them for further processing. Many applications implement this as a loop, calling `select()` repeatedly to receive packets—for example:

```
fd_data = create_socket(...);
fd_ctrl = create_socket(...);
while (not_done) {
   FD_ZERO(&rfd);
   FD_SET(fd_data, &rfd);
   FD_SET(fd_ctrl, &rfd);
   timeout = ...;
   if (select(max_fd, &rfd, NULL, NULL, timeout) > 0) {
      if (FD_ISSET(fd_data, &rfd)) {
         ...validate data packet
         ...process data packet
      }
      if (FD_ISSET(fd_ctrl, &rfd)) {
            ...validate control packet
            ...process control packet
        }
    }
    ...do other processing
}
```

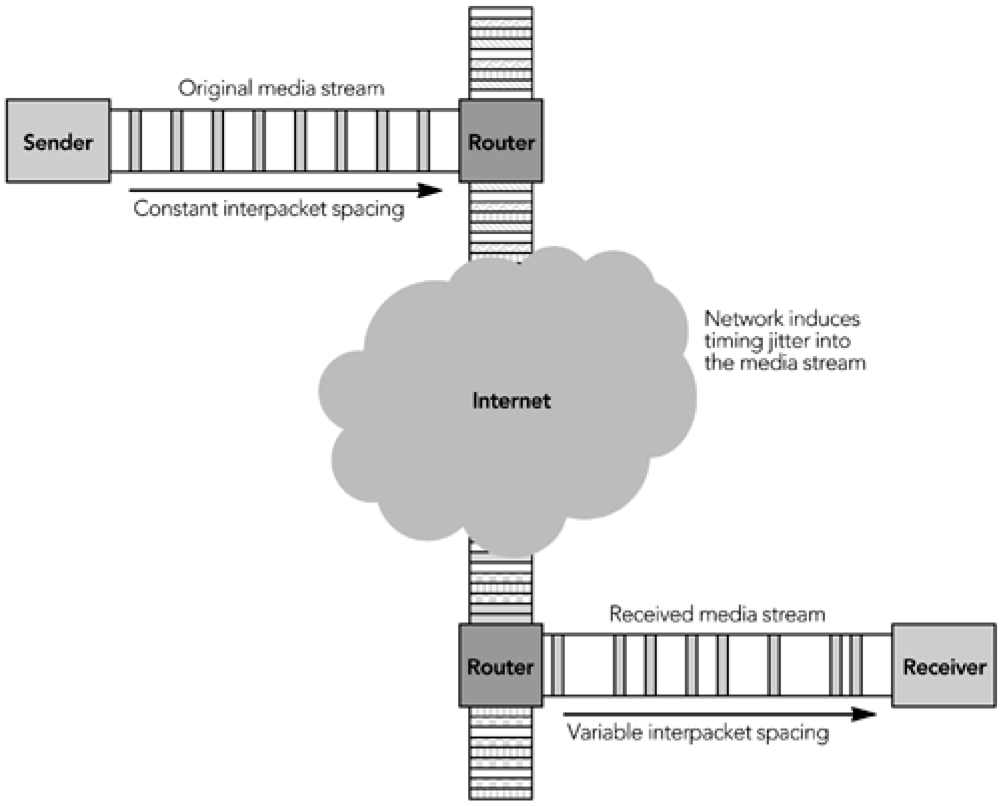
Data and control packets are validated for correctness as described in Chapters 4, RTP Data Transfer Protocol, and 5, RTP Control Protocol, and processed as described in the next two sections. The timeout of the `select()` operation is typically chosen according to the framing interval of the media. For example, a system receiving audio with 20-millisecond packet duration will implement a 20-millisecond timeout, allowing the other processing—such as decoding the received packets—to occur synchronously with arrival and playout, and resulting in an application that loops every 20 milliseconds.

Other implementations may be event driven rather than having an explicit loop, but the basic concept remains: Packets are continually validated and processed as they arrive from the network, and other application processing must be done in parallel to this (either explicitly time-sliced, as shown above, or as a separate thread), with the timing of the application driven by the media processing requirements. Real-time operation is essential to RTP receivers; packets must be processed at the rate they arrive, or reception quality will be impaired.

Receiving Data Packets

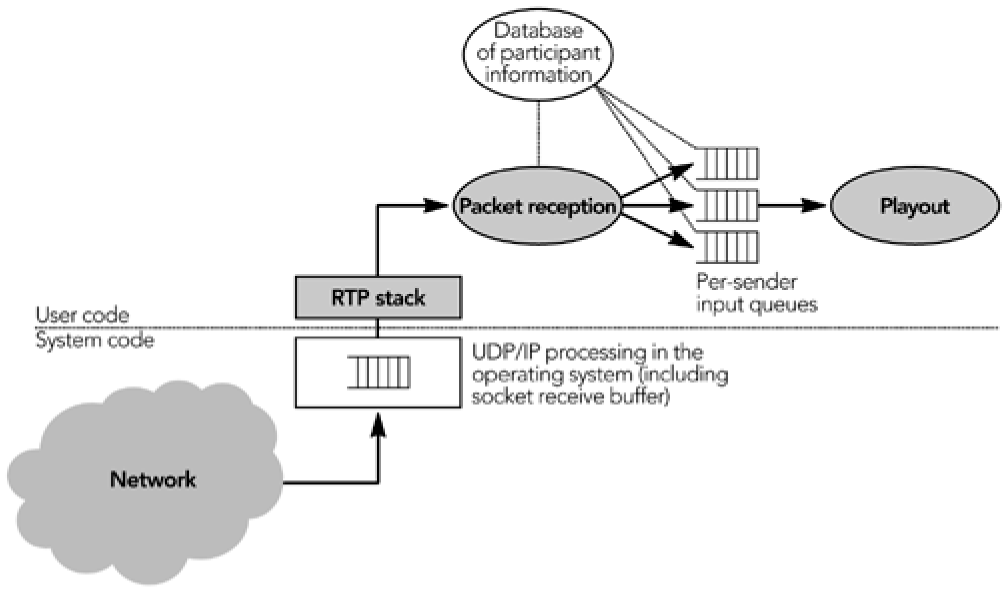
The first stage of the media playout process is to capture RTP data packets from the network, and to buffer those packets for further processing. Because the network is prone to disrupt the interpacket timing, as shown in Figure 6.5, there will be bursts when several packets arrive at once and/or gaps when no packets arrive, and packets may even arrive out of order. The receiver does not know when data packets are going to arrive, so it should be prepared to accept packets in bursts, and in any order.

Figure 6.5. Disruption of Interpacket Timing during Network Transit


Original media stream
Sender
Router
Constant interpacket spacing
Network induces timing jitter into the media stream
Internet
Received media stream
Router
Receiver
Variable interpacket spacing

As packets are received, they are validated for correctness, their arrival time is noted, and they are added to a per-sender input queue, sorted by RTP timestamp, for later processing. These steps decouple the arrival rate of packets from the rate at which they are processed and played to the user, allowing the application to cope with variation in the arrival rate. Figure 6.6 shows the separation between the packet reception and playout routines, which are linked only by the input queues.

Figure 6.6. Separation of Packet Reception from Playout, Using Input Queues


Database of participant information
Packet reception
Playout
Per-sender input queues
RTP stack
User code
System code
UDP/IP processing in the operating system (including socket receive buffer)
Network

It is important to store the exact arrival time, *M*, of RTP data packets so that the interarrival jitter can be calculated. Inaccurate arrival time measurements give the appearance of network jitter and cause the playout delay to increase. The arrival time should be measured according to a local reference wall clock, *T*, converted to the media clock rate, *R*. It is unlikely that the receiver has such a clock, so usually we calculate the arrival time by sampling the reference clock (typically the system wall clock time) and converting it to the local timeline:

$$M = T \times R + \texttt{offset}$$

where the `offset` is used to map from the reference clock to the media timeline, in the process correcting for skew between the media clock and the reference clock.

As noted earlier, processing of data packets may be time-sliced along with packet reception in a single-threaded application, or it may run in a separate thread in a multithreaded system. In a time-sliced design, a single thread handles both packet reception and playout. On each loop, all outstanding packets are read from the socket and inserted into the correct input queue. Packets are removed from the queues as needed and scheduled for playout. If packets arrive in bursts, some may remain in their input queue for multiple iterations of the loop, depending on the desired rate of playout and available processing capacity.

A multithreaded receiver typically has one thread waiting for data to arrive on the socket, sorting arriving packets onto the correct input queue. Other threads pull data from the input queues and arrange for the decoding and playout of the media. The asynchronous operation of the threads, along with the buffering in the input queues, effectively decouples the playout process from short-term variations in the input rate.

No matter what design is chosen, an application will usually not be able to receive and process packets continually. The input queues accommodate fluctuation in the playout process within the application, but what of delays in the packet reception routine? Fortunately, most general-purpose operating systems handle reception of UDP/IP packets on an interrupt-driven basis and can buffer packets at the socket level even when the application is busy. This capability provides limited buffering before packets reach the application. The default socket buffer is suitable for most implementations, but applications that receive high-rate streams or have significant periods of time when they are unable to handle reception may need to increase the size of the socket buffer beyond its default value (the `setsockopt(fd, SOL_SOCKET, SO_RCVBUF, ...)` function performs this operation on many systems). The larger socket buffer accommodates varying delays in packet reception processing, but the time packets spend in the socket buffer appears to the application as jitter in the network. The application might increase its playout delay to compensate for this perceived variation.

Receiving Control Packets

In parallel with the arrival of data packets, an application must be prepared to receive, validate, process, and send RTCP control packets. The information in the RTCP packets is used to maintain the database of the senders and receivers within a session, as discussed in Chapter 5, RTP Control Protocol, and for participant validation and identification, adaptation to network conditions, and lip synchronization. The participant database is also a good place from which to hang the participant-specific input queues, playout buffer, and other state needed by the receiver.

Single-threaded applications typically include both data and control sockets in their `select()` loop, interleaving reception of control packets along with all other processing. Multithreaded applications can devote a thread to RTCP reception and processing. Because RTCP packets are infrequent compared to data packets, the overhead of their processing is usually low and is not especially time-critical. It is, however, important to record the exact arrival time of sender report (SR) packets because this value is returned in receiver report (RR) packets and used in the round-trip time calculation.

When RTCP sender/receiver report packets arrive—describing the reception quality as seen at a particular receiver—the information they contain is stored. Parsing the report blocks in SR/RR packets is straightforward, provided you remember that the data is in network byte order and must be converted to host order before being used. The `count` field in the RTCP header indicates how many report blocks are present; remember that zero is a valid value, indicating that the sender of the RTCP packet is not receiving any RTP data packets.

The main use of RTCP sender/receiver reports is for an application to monitor reception of the streams it has sent: If the reports indicate poor reception, it is possible either to add error protection codes or to reduce the sending rate to compensate. In multisender sessions it is also possible to monitor the quality of other senders, as seen by other receivers; for example, a network operations center might monitor SR/RR packets as a check that the network is operating correctly. Applications typically store reception quality data as it is received, and periodically they use the stored data to adapt their transmission.

Sender reports also contain the mapping between the RTP media clock and the sender's reference clock, used for lip synchronization (see Chapter 7), and a count of the amount of data sent. Once again, this information is in network byte order and needs to be converted before use. This information needs to be stored if it is to be used for lip synchronization purposes.

When RTCP source description packets arrive, the information they contain is stored and may be displayed to the user. The RTP specification contains sample code to parse SDES packets (see Appendix A.5 of the specification[50]). The SDES CNAME (canonical name) provides the link between audio and video streams, indicating where lip synchronization should be performed. It is also used to group multiple streams coming from a single source—for example, if a participant has multiple cameras sending video to a single RTP session—and this may affect the way media is displayed to the user.

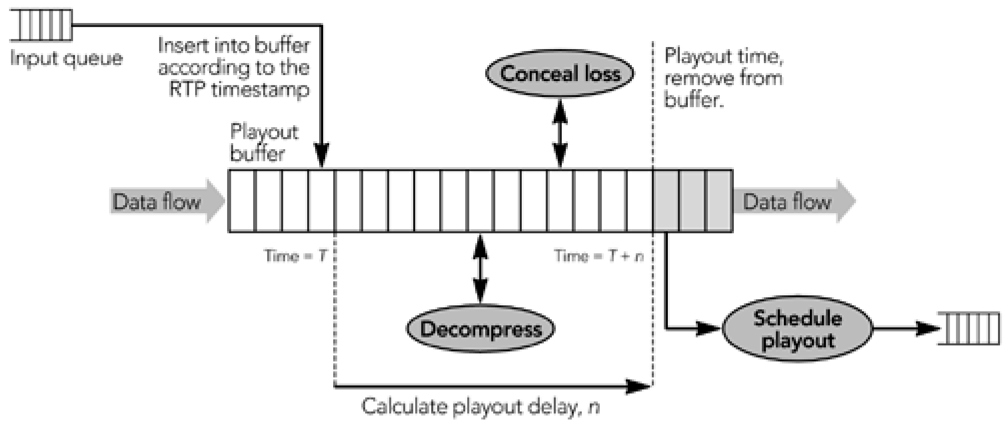
Once RTCP packets have been validated, the information they contain is added to the participant database. Because the validity checks for RTCP packets are strong, the presence of a participant in the database is a solid indication that the participant is valid. This is a useful check when RTP packets are being validated: If the SSRC in an RTP data packet was previously seen in an RTCP packet, it is highly likely to be a valid source.

When RTCP BYE packets are received, entries in the participant database are marked for later removal. As noted in Chapter 5, RTP Control Protocol, entries are not removed immediately but should be kept for some small time to allow any delayed packets to arrive. (My own implementation uses a fixed two-second timeout; the precise value is unimportant, provided that it is larger than the typical network timing jitter.) Receivers also perform periodic housekeeping to time out inactive participants. Performing this task with every packet is not necessary; once per RTCP report interval is sufficient.

The Playout Buffer

Data packets are extracted from their input queue and inserted into a source-specific playout buffer sorted by their RTP timestamps. Frames are held in the playout buffer for a period of time to smooth timing variations caused by the network. Holding the data in a playout buffer also allows the pieces of fragmented frames to be received and grouped, and it allows any error correction data to arrive. The frames are then decompressed, any remaining errors are concealed, and the media is rendered for the user. Figure 6.7 illustrates the process.

Figure 6.7. The Playout Buffer



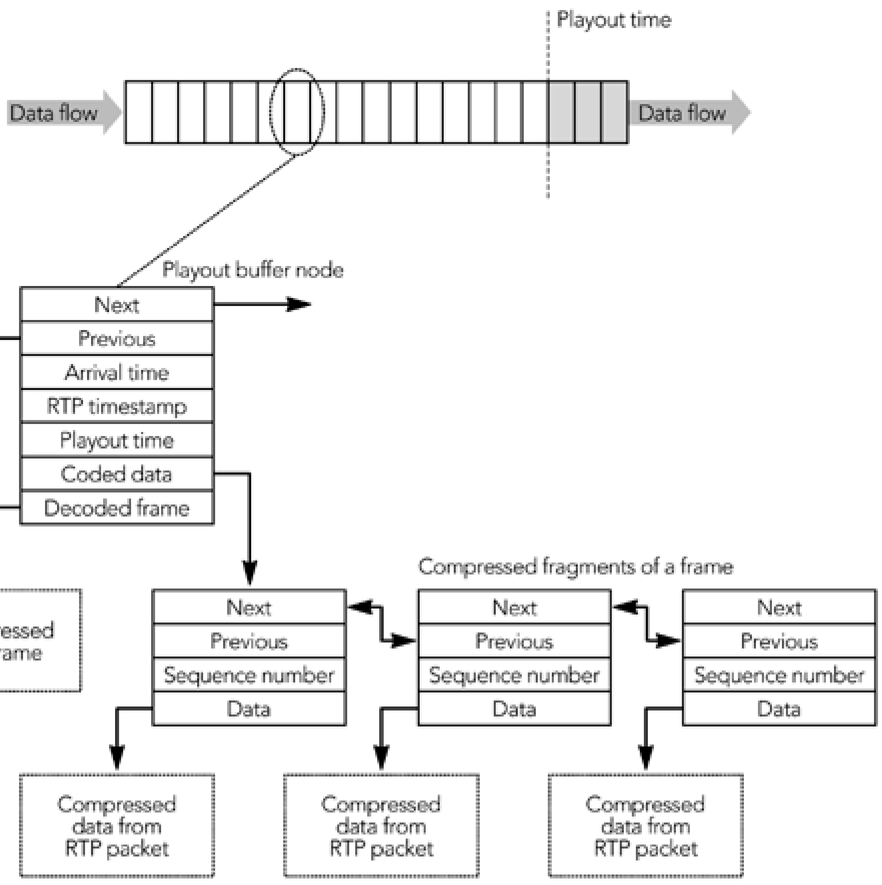
Input queue
Insert into buffer according to the RTP timestamp
Playout buffer
Conceal loss
Playout time, remove from buffer.
Data flow
Data flow
Time = T
Time = T + n
Decompress
Schedule playout
Calculate playout delay, n

A single buffer may be used to compensate for network timing variability and as a decode buffer for the media codec. It is also possible to separate these functions: using separate buffers for jitter removal and decoding. However, there is no strict layering requirement in RTP: Efficient implementations often mingle related functions across layer boundaries, a concept termed *integrated layer processing*.[65]

Basic Operation

The playout buffer comprises a time-ordered linked list of nodes. Each node represents a frame of media data, with associated timing information. The data structure for each node contains pointers to the adjacent nodes, the arrival time, RTP timestamp, and desired playout time for the frame, and pointers to both the compressed fragments of the frame (the data received in RTP packets) and the uncompressed media data. Figure 6.8 illustrates the data structures involved.

Figure 6.8. The Playout Buffer Data Structures



When the first RTP packet in a frame arrives, it is removed from the input queue and positioned in the playout buffer in order of its RTP timestamp. This involves creating a new playout buffer node, which is inserted into the linked list of the playout buffer. The compressed data from the recently arrived packet is linked from the playout buffer node, for later decoding. The frame's playout time is then calculated, as explained later in this chapter.

The newly created node resides in the playout buffer until its playout time is reached. During this waiting period, packets containing other fragments of the frame may arrive and are linked from the node. Once it has been determined that all the fragments of a frame have been received, the decoder is invoked and the resulting uncompressed frame linked from the playout buffer node. Determining that a complete frame has been received depends on the codec:

- Audio codecs typically do not fragment frames, and they have a single packet per frame (MPEG Audio Layer-3—MP3—is a common exception);
- Video codecs often generate multiple packets per video frame, with the RTP marker bit being set to indicate the RTP packet containing the last fragment.

Receiving a video packet with the marker bit set does not necessarily mean that the complete frame has been received, since packets may be lost or reordered in transit. Instead, it gives the highest RTP sequence number for a frame. Once all RTP packets with the same timestamp but lower sequence number have been received, the frame is complete. Whether the frame is complete can easily be determined if the packet with the marker bit for the previous frame was received. If that packet was lost, as revealed by a timestamp change that appears without the marker bit, and if only one packet is lost according to the sequence numbers, then the first packet after the loss is the first packet of the frame. If multiple packets are lost, typically it is not possible to tell whether those packets belonged to the new frame or the previous frame (knowledge of the media format may make it possible to determine the frame boundary in some cases, but that ability depends on the specific codec and payload format).

The decision of when to invoke the decoder depends on the receiver and is not specified by RTP. Frames can be decoded as soon as they arrive or kept compressed until the last possible moment. The choice depends on the relative availability of processing cycles and storage space for uncompressed frames, and perhaps on the receiver's estimate of future resource availability. For example, a receiver may wish to decode data early if it knows that an index frame is due and it will shortly be busy.

Eventually the playout time for a frame arrives, and the frame is queued for playout as discussed in the section Decoding, Mixing, and Playout later in this chapter. If the frame has not already been decoded, at this time the receiver must make its best effort to decode the frame, even if some fragments are missing, because this is the last chance before the frame is needed. This is also the time when error concealment (see Chapter 8) may be invoked to hide any uncorrected packet loss.

Once the frame has been played out, the corresponding playout buffer node and its linked data should be destroyed or recycled. If error concealment is used, however, it may be desirable to delay this process until the surrounding frames have also been played out because the linked media data may be useful for the concealment operation.

RTP packets arriving late and corresponding to frames that have missed their playout point should be discarded. The timeliness of a packet can be determined by comparison of its RTP timestamp with the timestamp of the oldest packet in the playout buffer (note that the comparison should be done with 32-bit modulo arithmetic, to allow for timestamp wrap-around). It is clearly desirable to choose the playout delay so that late packets are rare, and applications should monitor the number of late packets and be prepared to adapt their playout delay in response. Late packets indicate an inappropriate playout delay, typically caused by changing network delays or skew between clocks at the sending and receiving hosts.

The trade-off in playout buffer operation is between fidelity and delay: An application must decide the maximum playout delay it can accept, and this in turn determines the fraction of packets that arrive in time to be played out. A system designed for interactive use—for example, video conferencing or telephony—must try to keep the playout delay as small as possible because it cannot afford the latency incurred by the buffering. Studies of human perception point to a limit in round-trip time of about 300 milliseconds as the maximum tolerable for interactive use; this limit implies an end-to-end delay of only 150 milliseconds including network transit time and buffering delay if the system is symmetric. However, a noninteractive system, such as streaming video, television, or radio, may allow the playout buffer to grow up to several seconds, thereby enabling noninteractive systems to handle variation in packet arrival times better.

Playout Time Calculation

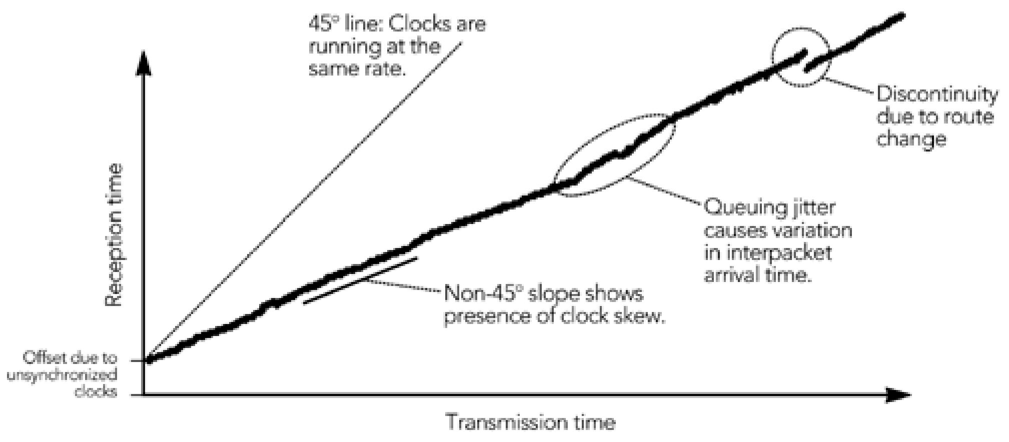
The main difficulty in designing an RTP playout buffer is determining the playout delay: How long should packets remain in the buffer before being scheduled for playout? The answer depends on various factors:

- The delay between receiving the first and last packets of a frame
- The delay before any error correction packets are received (see Chapter 9, Error Correction)
- The variation in interpacket timing caused by network queuing jitter and route changes
- The relative clock skew between sender and receiver
- The end-to-end delay budget of the application, and the relative importance of reception quality and latency

The factors under control of the application include the spacing of packets in a frame, and the delay between the media data and any error correction packets, both of which are controlled by the sender. The effects of these factors on the playout delay calculation are discussed in the section titled Compensation for Sender Behavior later in this chapter.

Outside the control of the application is the behavior of the network, and the accuracy and stability of the clocks at sender and receiver. As an example, consider Figure 6.9, which shows the relationship between packet transmission time and reception time for a trace of RTP audio packets. If the sender clock and receiver clock run at the same rate, as is desired, the slope of this plot should be exactly 45 degrees. In practice, sender and receiver clocks are often unsynchronized and run at slightly different rates. For the trace in Figure 6.9, the sender clock is running faster than the receiver clock, so the slope of the plot is less than 45 degrees (Figure 6.9 is an extreme example, to make it easy to see the effect; the slope is typically much closer to 45 degrees). The section titled Compensation for Clock Skew later in this chapter explains how to correct for unsynchronized clocks.

Figure 6.9. Packet Send Time versus Receive Time, Illustrating Clock Skew



If the packets have a constant network transit time, the plot in Figure 6.9 will produce an exactly straight line. However, typically the network induces some jitter in the interpacket spacing due to variation in queuing delays, and this is observable in the figure as deviations from the straight-line plot. The figure also shows a discontinuity, resulting from a step change in the network transit time, most likely due to a route change in the network. Chapter 2, Voice and Video Communication over Packet Networks, has a more detailed discussion of the effects, and the section titled Compensation for Jitter later in this chapter explains how to correct for these issues. Correcting for more extreme variations is discussed in the sections Compensation for Route Changes and Compensation for Packet Reordering.

The final point to consider is the end-to-end delay budget of the application. This is mainly a human-factors issue: What is the maximum acceptable end-to-end delay for the users of the application, and how long does this leave for smoothing in the playout buffer after the network transit time has been factored out? As might be expected, the amount of time available for buffering does affect the design of the playout buffer; the section titled Compensation for Jitter discusses this subject further.

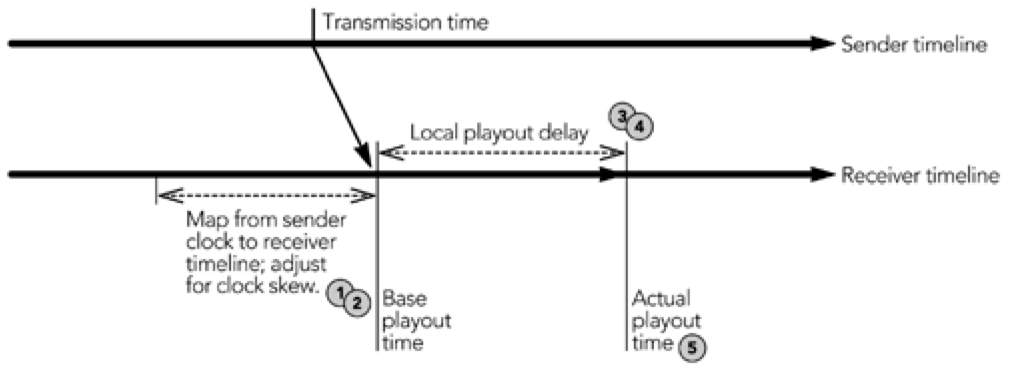
A receiver should take these factors into account when determining the playout time for each frame. The playout calculation follows several steps:

1. The sender timeline is mapped to the local playout timeline, compensating for the relative offset between sender and receiver clocks, to derive a base time for the playout calculation (see Mapping to the Local Timeline later in this chapter).

2. If necessary, the receiver compensates for clock skew relative to the sender, by adding a skew compensation offset that is periodically adjusted to the base time (see Compensation for Clock Skew).
3. The playout delay on the local timeline is calculated according to a sender-related component of the playout delay (see Compensation for Sender Behavior) and a jitter-related component (see Compensation for Jitter).
4. The playout delay is adjusted if the route has changed (see Compensation for Route Changes), if packets have been reordered (see Compensation for Packet Reordering), if the chosen playout delay causes frames to overlap, or in response to other changes in the media (see Adapting the Playout Point).
5. Finally, the playout delay is added to the base time to derive the actual playout time for the frame.

Figure 6.10 illustrates the playout calculation, noting the steps of the process. The following sections give details of each stage.

Figure 6.10. The Playout Calculation



MAPPING TO THE LOCAL TIMELINE

The first stage of the playout calculation is to map from the sender's timeline (as conveyed in the RTP timestamp) to a time-line meaningful to the receiver, by adding the relative offset between sender and receiver clocks to the RTP timestamp.

To calculate the relative offset, the receiver tracks the difference, $d(n)$, between the RTP timestamp of the nth packet, $T_{R(n)}$, and the arrival time of that packet, $T_{L(n)}$, measured in the same units:

$$d(n) = T_{L(n)} - T_{R(n)}$$

The difference, $d(n)$, includes a constant factor because the sender and receiver clocks were initialized at different times with different random values, a variable delay due to data preparation time at the sender, a constant factor due to the minimum network transit time, a variable delay due to network timing jitter, and a rate difference due to clock skew. The difference is a 32-bit unsigned integer, like the timestamps from which it is calculated; and because the sender and receiver clocks are unsynchronized, it can have any 32-bit value.

The difference is calculated as each packet arrives, and the receiver tracks its minimum observed value to obtain the relative offset:

$$\texttt{offset} = \min(d(n-w)\ldots d(n))$$

Because of the rate difference between $T_{L(n)}$ and $T_{R(n)}$ due to clock skew, the difference, $d(n)$, will tend to drift larger or smaller. To prevent this drift, the minimum offset is calculated over a window, w, of the differences since the last compensation for clock skew. Also note that an unsigned comparison is required because the values may wrap around:

```
a < b if (a - b) & 0x80000000 != 0
```

The `offset` value is used to calculate the base playout point, according to the timeline of the receiver:

$$\texttt{base\_playout\_time}(n) = \mathrm{T}_{R(n)} + \texttt{offset}$$

This is the initial estimate of the playout time, to which are applied additional factors compensating for clock skew, jitter, and so on.

COMPENSATION FOR CLOCK SKEW

RTP payload formats define the nominal clock rate for a media stream but place no requirements on the stability and accuracy of the clock. Sender and receiver clocks commonly run at slightly different rates, forcing the receiver to compensate for the variation. A plot of packet transmission time versus reception time, as in Figure 6.9, illustrates this. If the slope of the plot is exactly 45 degrees, the clocks have the same rate; deviations are caused by clock skew between sender and receiver.

Receivers must detect the presence of clock skew, estimate its magnitude, and adjust the playout point to compensate. There are two possible compensation strategies: tuning the receiver clock to match the sender clock, or periodically adjusting playout buffer occupancy to regain alignment.

The latter approach accepts the skew and periodically realigns the playout buffer by inserting or deleting data. If the sender is faster, the receiver will eventually have to discard some data to bring the clocks into alignment, otherwise its playout buffer will be over-run. If the sender is slower, the receiver will eventually run out of media to play, and must synthesize some data to fill the gap that is left. The magnitude of the clock skew determines the frequency of playout point adjustments, and hence the quality degradation experienced.

Alternatively, if the receiver clock rate is finely adjustable, it may be possible to tune its rate to exactly match that of the sender, avoiding the need for a playout buffer realignment. This approach can give higher quality because data is never discarded due to skew, but it may require hardware support that is not common (systems using audio may be able to resample to match the desired rate using software).

Estimating the amount of clock skew present initially appears to be a simple problem: Observe the rate of the sender clock—the RTP timestamp—and compare with the local clock. If $T_{R(n)}$ is the RTP timestamp of the *n*th packet received, and $T_{L(n)}$ is the value of the local clock at that time, then the clock skew can be estimated as follows:

$$\text{skew} = \frac{T_{R(n)} - T_{R(n-1)}}{T_{L(n)} - T_{L(n-1)}}$$

with a skew of less than unity meaning that the sender is slower than the receiver, and a skew of greater than unity meaning that the sender clock is fast compared to the receiver. Unfortunately, the presence of network timing jitter means that this simple estimate is not sufficient; it will be directly affected by variation in interpacket spacing due to jitter. Receivers must look at the long-term variation in the packet arrival rate to derive an estimate for the underlying clock skew, removing the effects of jitter.

There are many possible algorithms for managing clock skew, depending on the accuracy and sensitivity to jitter that is required. In the following discussion I describe a simple approach to estimating and compensating for clock skew that has proven suitable for voice-over-IP applications,[79] and I give pointers to algorithms for more demanding applications.

The simple approach to clock skew management continually monitors the average network transit delay and compares it with an active delay estimate. Increasing divergence between the active delay estimate and measured average delay denotes the presence of clock skew, eventually causing the receiver to adapt playout. As each packet arrives, the receiver calculates the instantaneous one-way delay for the *n*th packet, d_n, based on the reception time of the packet and its RTP timestamp:

$$d_n = T_{L(n)} - T_{R(n)}$$

On receipt of the first packet, the receiver sets the active delay, $E = d_0$, and the estimated average delay, $D_0 = d_0$. With each subsequent packet the average delay estimate, D_n, is updated by an exponentially weighted moving average:

$$\begin{aligned} D_n &= {}^{31}/_{32} D_{n-1} + (1 - {}^{31}/_{32}) d_n \\ &= (31\, D_{n-1} + d_n)/32 \end{aligned}$$

The factor ${}^{31}/_{32}$ controls the averaging process, with values closer to unity making the average less sensitive to short-term fluctuation in the transit time. Note that this calculation is similar to the calculation of the estimated jitter; but it retains the sign of the variation, and it uses a time constant chosen to capture the long-term variation and reduce the response to short-term jitter.

The average one-way delay, D_n, is compared with the active delay estimate, *E*, to estimate the divergence since the last estimate:

$$\text{divergence} = E - D_n$$

If the sender clock and receiver clock are synchronized, the divergence will be close to zero, with only minor variations due to network jitter. If the clocks are skewed, the divergence will increase or decrease until it exceeds a predetermined threshold, causing the receiver to take compensating action. The threshold depends on the jitter, the codec, and the set of possible adaptation points. It has to be large enough that false adjustments due to jitter are avoided, and it should be chosen such that the discontinuity caused by an adjustment can easily be concealed. Often a single framing interval is suitable, meaning that an entire codec frame is inserted or removed.

Compensation involves growing or shrinking the playout buffer as described in the section titled Adapting the Playout Point later in this chapter. The playout point can be changed up to the divergence as measured in RTP timestamp units (for audio, the divergence typically gives the number of samples to add or remove). After compensating for skew, the receiver resets the active delay estimate, *E*, to equal the current delay estimate, D_n, resetting the divergence to zero in the process (the estimate for `base_play-out_time`(*n*) is also reset at this time).

In C-like pseudocode, the algorithm performed as each packet is received becomes this:

```
adjustment_due_to_skew(rtp_packet p, uint32_t curr_time) {
   static int      first_time = 1;
   static uint32_t  delay_estimate;
   static uint32_t  active_delay;
   uint32_t        adjustment = 0;
   uint32_t        d_n = p->ts – curr_time;

   if (first_time) {
      first_time = 0;
      delay_estimate = d_n;
      active_delay = d_n;
   } else {
      delay_estimate = (31 * delay_estimate + d_n)/32;
   }
   if (active_delay – delay_estimate > SKEW_THRESHOLD) {
      // Sender is slow compared to receiver
      adjustment   = SKEW_THRESHOLD;
      active_delay = delay_estimate;
   }
   if (active_delay – delay_estimate < -SKEW_THRESHOLD) {
      // Sender is fast compared to receiver
      adjustment   = -SKEW_THRESHOLD;
      active_delay = delay_estimate;
   }
   // Adjustment will be 0, SKEW_THRESHOLD, or –SKEW_THRESHOLD. It is
   // appropriate that SKEW_THRESHOLD equals the framing interval.
   return adjustment;
}
```

The assumptions of this algorithm are that the jitter distribution is symmetric and that any systematic bias is due to clock skew. If the distribution of skew values is asymmetric for reasons other than clock skew, this algorithm will cause spurious skew adaptation. Significant short-term fluctuations in the network transit time also might confuse the algorithm, causing the receiver to perceive network jitter as clock skew and adapt its playout point. Neither of these issues should cause operational problems: The skew compensation algorithm will eventually correct itself, and any adaptation steps would likely be needed in any case to accommodate the fluctuations.

Another assumption of the skew compensation described here is that it is desirable to make step adjustments to the playout point—for example, adding or removing a complete frame at a time—while concealing the discontinuity as if a packet were lost. For many codecs—in particular, frame-based voice codecs—this is appropriate behavior because the codec is optimized to conceal lost frames and skew compensation can leverage this ability, provided that care is taken to add or remove unimportant, typically low-energy, frames. In some cases, however, it is desirable to adapt more smoothly, perhaps interpolating a single sample at a time.

If smoother adaptation is needed, the algorithm by Moon et al.[90] may be more suitable, although it is more complex and has correspondingly greater requirements for state maintenance. The basis of their approach is to use linear programming on a plot of observed one-way delay versus time, to fit a line that lies under all the data points, as closely to them as possible. An equivalent approach is to derive a best-fit line under the data points of a plot such as that in Figure 6.9, and use this to estimate the slope of the line, and hence the clock skew. Such algorithms are more accurate, provided the skew is constant, but they clearly have higher overheads because they require the receiver to keep a history of points, and perform an expensive line-fitting algorithm. They can, however, derive very accurate skew measurements, given a long enough measurement interval.

Long-running applications should take into account the possibility that the skew might be nonstationary, and vary according to outside effects. For example, temperature changes can affect the frequency of crystal oscillators and cause variation in the clock rate and skew between sender and receiver. Nonstationary clock skew may confuse some algorithms (for example, that of Moon et al.[90]) that use long-term measurements. Other algorithms, such as that of Hodson et al.,[79] described earlier, work on shorter timescales and periodically recalculate the skew, so they are robust to variations.

When choosing a clock skew estimation algorithm, it is important to consider how the playout point will be varied, and to choose an estimator with an appropriate degree of accuracy. For example, applications using frame-based audio codecs may adapt by adding or removing a single frame, so an estimator that measures skew to the nearest sample may be overkill. The section titled Adapting the Playout Point

Although they are outside the scope of this book, the algorithms of the Network Time Protocol may also be of interest to implementers. RFC 1305[5] is recommended reading for those with an interest in clock synchronization and skew compensation (the PDF version, available from the RFC Editor Web site, at http://www.rfc-editor.org, is considerably more readable than the text-only version).

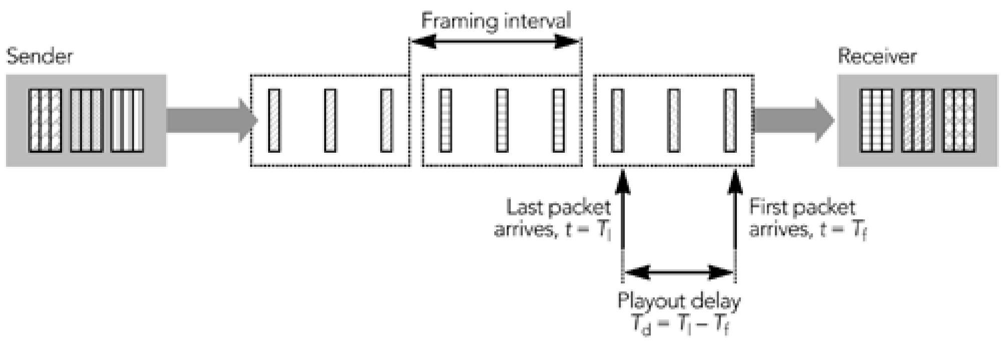
later in this chapter discusses this issue in more detail.

COMPENSATION FOR SENDER BEHAVIOR

The nature of the sender's packet generation process can influence the receiver's playout calculation in several ways, causing increased playout buffering delay.

If the sender spreads the packets that make up a frame in time across the framing interval, as is common for video, there will be a delay between the first and last packets of a frame, and receivers must buffer packets until the whole frame is received. Figure 6.11 shows the insertion of additional playout delay, T_d, to ensure that the receiver does not attempt to play out the frame before all the fragments have arrived.

Figure 6.11. Buffering Delay, to Group Packets into Frames


Framing interval
Sender
Receiver
Last packet arrives, $t = T_l$
First packet arrives, $t = T_f$
Playout delay $T_d = T_l - T_f$

If the interframe timing and number of packets per frame are known, inserting this additional delay is simple. Assuming that the sender spaces packets evenly, the adjustment will be as follows:

```
adjustment_due_to_fragmentation = (packets_per_frame – 1)
                            x (interframe_time / packets_per_frame)
```

Unfortunately, receivers do not always know these variables in advance. For example, the frame rate may not be signaled during session setup, the frame rate may vary during a session, or the number of packets per frame may vary during a session. This variability can make it difficult to schedule playout because it is unclear how much delay needs to be added to allow all fragments to arrive. The receiver must estimate the required playout delay, and adapt if the estimate proves inaccurate.

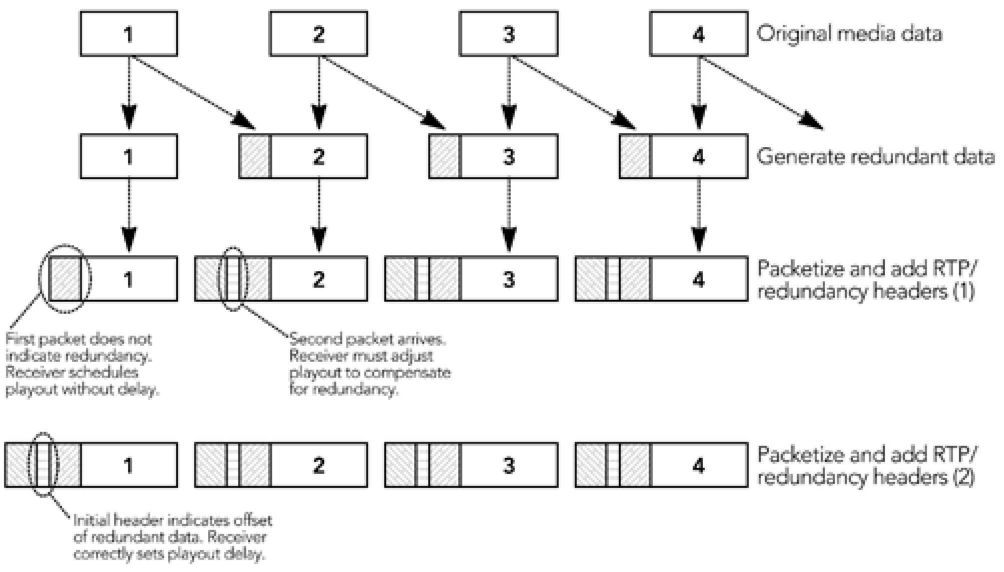
The estimated playout compensation could be calculated by a special-purpose routine that looked at the arrival times of fragments to calculate an average fragmentation delay. Fortunately, this is not necessary; the jitter calculation performs the same role. All packets of a frame have the same timestamp—representing the time of the frame, rather than the time the packet was sent—so fragmented frames cause the appearance of jitter (the receiver cannot differentiate between a packet delayed in the network and one delayed by the sender). The strategies for jitter compensation discussed in the next section can therefore be used to estimate the amount of buffering delay needed to compensate for fragmentation, and there is no need to account for fragmentation in the host component of the playout delay.

Similar issues arise if the sender uses the error correction techniques described in Chapter 9. For error correction packets to be useful, playout must be delayed so that the error correction packets arrive in time to be used. The presence of error correction packets is signaled during session setup, and the signaling may include enough information to allow the receiver to size the playout buffer correctly. Alternatively, the correct playout delay must be inferred from the media stream. The compensation delay needed depends on the type of error correction employed. Three common types of error correction are parity FEC (forward error correction), audio redundancy, and retransmission.

The parity FEC scheme discussed in Chapter 9[32] leaves the data packets unmodified, and sends error correction packets in a separate RTP stream. The error correction packets contain a bit mask in their FEC header to identify the sequence numbers of the packets they protect. By observing the mask, a receiver can determine the delay to add, in packets. If packet spacing is constant, this delay translates to a time offset to add to the playout calculation. If the interpacket spacing is not constant, the receiver must use a conservative estimate of the spacing to derive the required play-out delay.

The audio redundancy scheme[10] discussed in Chapter 9, Error Correction, includes a time offset in redundant packets, and this offset may be used to size the playout buffer. At the start of a talk spurt, redundant audio may be used in two modes: Initial packets may be sent without the redundancy header, or they may be sent with a zero-length redundant block. As Figure 6.12 shows, sizing the playout buffer is easier if a zero-length redundant block is included with initial packets in a talk spurt. Unfortunately, including these blocks is not mandatory in the specification, and implementations may have to guess an appropriate playout delay if it is not present (a single packet offset is most common and makes a reasonable estimate in the absence of other information). Once a media stream has been determined to use redundancy, the offset should be applied to all packets in that stream, including any packets sent without redundancy at the beginning of a talk spurt. If a complete talk spurt is received without redundancy, it can be assumed that the sender has stopped redundant transmission, and future talk spurts can be played without delay.

Figure 6.12. Effects of Audio Redundancy Coding on the Playout Buffer


1
2
3
4
Original media data
Generate redundant data
Packetize and add RTP/ redundancy headers (1)
First packet does not indicate redundancy. Receiver schedules playout without delay.
Second packet arrives. Receiver must adjust playout to compensate for redundancy.
Packetize and add RTP/ redundancy headers (2)
Initial header indicates offset of redundant data. Receiver correctly sets playout delay.

A receiver of either parity FEC or redundancy should initially pick a large playout delay, to ensure that any data packets that arrive are buffered. When the first error correction packet arrives, it will cause the receiver to reduce its playout delay, reschedule, and play out any previously buffered packets. This process avoids a gap in playout caused by an increase in buffering delay, at the expense of slightly delaying the initial packet.

When packet retransmission is used, the playout buffer must be sized larger than the round-trip time between sender and receiver, to allow time for the retransmission request to return to the sender and be serviced. The receiver has no way of knowing the roundtrip time, short of sending a retransmission request and measuring the response time. This does not affect most implementations because retransmission is typically used in noninteractive applications in which the playout buffering delay is larger than the round-trip time, but it may be an issue if the round-trip time is large.

No matter what error correction scheme is used, the sender may be generating an excessive amount of error correction data. For example, when sending to a multicast group, the sender might choose an error correction code based on the worst-case receiver, which will be excessive for other receivers. As noted by Rosen-berg et al.,[101] it may then be possible to repair some fraction of loss with only a subset of the error correction data. In this case a receiver may choose a playout delay smaller than that required for all of the error correction data, instead just waiting long enough to repair the loss it chooses. The decision to ignore some error correction data is made solely by the receiver and based on its view of the transmission quality.

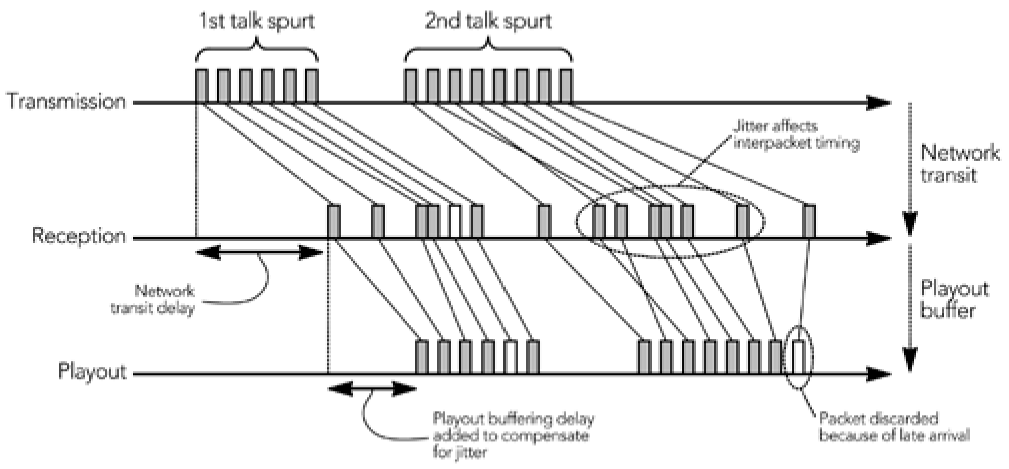
Finally, if the sender interleaves the media stream—as will be described in Chapter 8, Error Concealment—the receiver must allow for this in the playout calculation so that it can sort interleaved packets into playout order. Interleaving parameters are typically signaled during session setup, allowing the receiver to choose an appropriate buffering delay. For example, the AMR payload format[41] defines an `interleaving` parameter that can be signaled in the SDP `a=fmtp:` line, denoting the number of packets per interleaving group (and hence the amount of delay in terms of the number of packets that should be inserted into the playout buffer to compensate). Other codecs that support interleaving should supply a similar parameter.

To summarize, the sender may affect the playout buffer in three ways: by fragmenting frames and delaying sending fragments, by using error correction packets, or by interleaving. The first of these will be compensated for according to the usual jitter compensation algorithm; the others require the receiver to adjust the playout buffer to compensate. This compensation is mostly an issue for interactive applications that use small playout buffers to reduce latency; streaming media systems can simply set a large playout buffer.

COMPENSATION FOR JITTER

When RTP packets flow over a real-world IP network, variation in the interpacket timing is inevitable. This network jitter can be significant, and a receiver must compensate by inserting delay into its playout buffer so that packets held up by the network can be processed. Packets that are delayed too much arrive after their playout time has passed and are discarded; with suitable selection of a playout algorithm, this should be a rare occurrence. Figure 6.13 shows the jitter compensation process.

Figure 6.13. Network Jitter Affects Reception Time and Is Corrected in the Playout Buffer

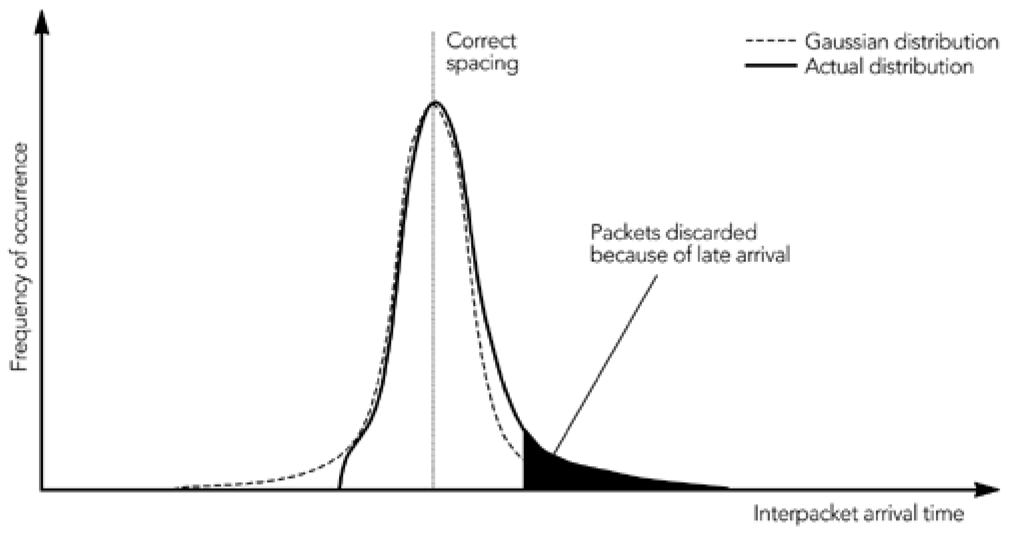


1st talk spurt
2nd talk spurt
Transmission
Jitter affects interpacket timing
Network transit
Reception
Network transit delay
Playout buffer
Playout
Playout buffering delay added to compensate for jitter
Packet discarded because of late arrival

There is no standard algorithm for calculation of the jitter compensation delay; most applications will want to calculate the play-out delay adaptively and may use different algorithms, depending on the application type and network conditions. Applications that are designed for noninteractive scenarios would do well to pick a compensation delay significantly larger than the expected jitter; an appropriate value might be several seconds. More complex is the interactive case, in which the application desires to keep the play-out delay as small as possible (values on the order of tens of milliseconds are not unrealistic, given network and packetization delays). To minimize the playout delay, it is necessary to study the properties of the jitter and use these to derive the minimum suitable playout delay.

In many cases the network-induced jitter is essentially random. A plot of interpacket arrival times versus frequency of occurrence will, in this case, be somewhat similar to the Gaussian distribution shown in Figure 6.14. Most packets are only slightly affected by the network jitter, but some outliers are significantly delayed or are back-to-back with a neighboring packet.

Figure 6.14. Distribution of Network Jitter



How accurate is this approximation? That depends on the network path, of course, but measurements taken by me and by Moon et al.[91] show that the approximation is reasonable in many cases, although real-world data is often skewed toward larger interarrival times and has a sharp minimum cutoff value (as illustrated by the "actual distribution" in Figure 6.14). The difference is usually not critical, because the number of packets in the discard region is small.

If it can be assumed that the jitter distribution does approximate a Gaussian normal distribution, then deriving a suitable playout delay is easily possible. The standard deviation of the jitter is calculated, and from probability theory we know that more than 99% of a normal distribution lies within three times the standard deviation of the mean (average) value. An implementation could choose a playout delay that is equal to three times the standard deviation in interarrival times and expect to discard less than 0.5% of packets because of late arrival. If this delay is too long, using a playout delay of twice the standard deviation will give an expected discard rate due to late arrival of less than 2.5%, again because of probability theory.

How can we measure the standard deviation? The jitter value calculated for insertion into RTCP receiver reports tracks the average variation in network transit time, which can be used to approximate the standard deviation. On the basis of these approximations, the playout delay required to compensate for network jitter can be estimated as three times the RTCP jitter estimate for a particular source. The playout delay for a new frame is set to at least

$T_{playout} = T_{current} + 3J$

where J is the current estimate of the jitter, as described in Chapter 5, RTP Control Protocol. The value of $T_{playout}$ may be modified in a media-dependent manner, as discussed later. Implementations using this value as a base for their playout calculation have shown good performance in a range of real-world conditions.

Although the RTCP jitter estimate provides a convenient value to use in the playout calculation, an implementation can use an alternative jitter estimate if that proves a more robust base for the playout time calculation (the standard jitter estimate *must* still be calculated and returned in RTCP RR packets). In particular, it has been suggested that the phase jitter—the difference between the time a packet arrived and the time it was expected—is a more accurate measure of network timing, although this has not yet been tested in widely deployed implementations. Accurate jitter prediction for interactive playout buffering is a difficult problem, with room for improvement over current algorithms.

The jitter distribution depends both on the path that traffic takes through the network and on the other traffic sharing that path. The primary cause of jitter is competition with other traffic, resulting in varying queuing delays at the intermediate routers; clearly, changes in the other traffic also will affect the jitter seen by a receiver. For this reason receivers should periodically recalculate the amount of playout buffering delay they use, in case the network behavior has changed, adapting if necessary. When should receivers adapt? This is not a trivial question, because any change in the playout delay while the media is playing will disrupt the playout, causing either a gap where there is nothing to play, or forcing the receiver to discard some data to make up lost time. Accordingly, receivers try to limit the number of times they adapt their playout point. Several factors can be used as triggers for adaptation:

- A significant change in the fraction of packets discarded because of late arrival
- Receipt of several consecutive packets that must be discarded because of late arrival (three consecutive packets is a suitable threshold)
- Receipt of packets from a source that has been inactive for a long period of time (ten seconds is a suitable threshold)
- The onset of a spike in the network transit delay

With the exception of spikes in the network transit delay, these factors should be self-explanatory. As shown in Figure 2.12 in Chapter 2, Voice and Video Communication over Packet Networks, the network occasionally causes "spikes" in the transit delay, when several packets are delayed and arrive in a burst. Such spikes can easily bias a jitter estimate, causing the application to choose a larger playout delay than is required. In many applications this increase in playout delay is acceptable, and applications should treat a spike as any other form of jitter and increase their playout delay to compensate. However, some applications prefer increased packet loss to increased latency; these applications should detect the onset of a delay spike and ignore packets in the spike when calculating the playout delay.[91], [96]

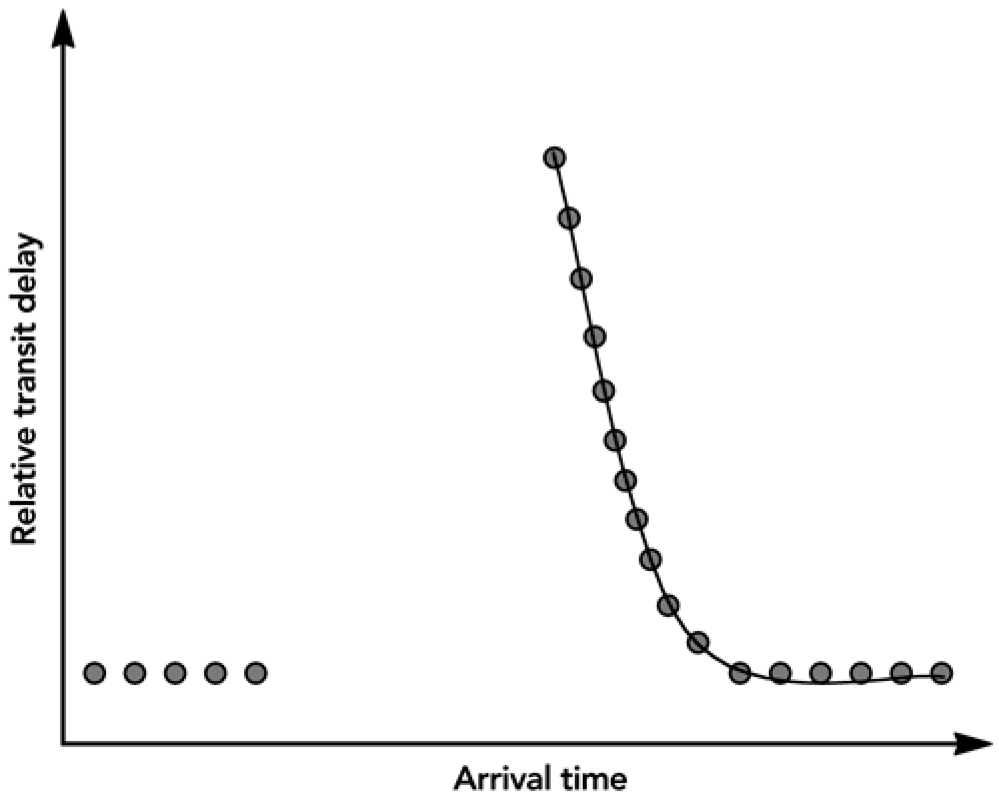
Detecting the start of a delay spike is simple: If the delay between consecutive packets increases suddenly, a delay spike likely has occurred. The scale of a "sudden increase" is open to some interpretation: Ramjee et al.[96] suggest that twice the statistical variance in interarrival time, plus 100 milliseconds, is a suitable threshold; another implementation that I'm familiar with uses a fixed threshold of 375 milliseconds (both are voice-over-IP systems using 8kHz speech).

Certain media events can also cause the delay between consecutive packets to increase, and should not be confused with the onset of a delay spike. For example, audio silence suppression will cause a gap between the last packet in one talk spurt and the first packet in the next. Equally, a change in video frame rate will cause interpacket timing to vary. Implementations should check for this type of event before assuming that a change in interpacket delay implies that a spike has occurred.

Once a delay spike has been detected, an implementation should suspend normal jitter adjustment until the spike has ended. As a result, several packets will probably be discarded because of late arrival, but it is assumed that the application has a strict delay bound, and that this result is preferable to an increased playout delay.

Locating the end of a spike is harder than detecting the onset. One key characteristic of a delay spike is that packets that were evenly spaced at the sender arrive in a burst after the spike, meaning that each packet has a progressively smaller transit delay, as shown in Figure 6.15. The receiver should maintain an estimate of the "slope" of the spike, and once it is sufficiently close to flat, the spike can be assumed to have ended.

Figure 6.15. Network Transit Time during a Delay Spike



Given all these factors, pseudocode to compensate playout delay for the effects of jitter and delay spikes is as follows:

```
int
adjustment_due_to_jitter(...)
{
   delta_transit = abs(transit – last_transit);
   if (delta_transit > SPIKE_THRESHOLD) {
      // A new "delay spike" has started
      playout_mode = SPIKE;
      spike_var = 0;
      adapt = FALSE;
   } else {
      if (playout_mode == SPIKE) {
         // We're within a delay spike; maintain slope estimate
         spike_var = spike_var / 2;
         delta_var = (abs(transit – last_transit) + abs(transit
                                 last_last_transit))/8;
         spike_var = spike_var + delta_var;
         if (spike_var < spike_end) {
            // Slope is flat; return to normal operation
            playout_mode = NORMAL;
        }
        adapt = FALSE;
    } else {
        // Normal operation; significant events can cause us to
        //adapt the playout
        if (consecutive_dropped > DROP_THRESHOLD) {
            // Dropped too many consecutive packets
            adapt = TRUE;
        }
        if ((current_time – last_header_time) >
                                  INACTIVE_THRESHOLD) {
            // Silent source restarted; network conditions have
            //probably changed
            adapt = TRUE;
        }
    }
  }
  desired_playout_offset = 3 * jitter
  if (adapt) {
      playout_offset = desired_playout_offset;
  } else {
      playout_offset = last_playout_offset;
  }
  return playout_offset;
}
```

The key points are that jitter compensation is suspended during a delay spike, and that the actual playout time changes only when a significant event occurs. At other times the `desired_playout_offset` is stored to be instated at a media-specific time (see the section titled Adapting the Playout Point).

COMPENSATION FOR ROUTE CHANGES

Although infrequent, route changes can occur in the network because of link failures or other topology changes. If a change occurs in the route taken by the RTP packets, it will manifest itself as a sudden change in the network transit time. This change will disrupt the playout buffer because either the packets will arrive too late for playout, or they will be early and overlap with previous packets.

The jitter and delay spike compensation algorithms should detect the change in delay and adjust the playout to compensate, but this approach may not be optimal. Faster adaptation can take place if the receiver observes the network transit time directly and adjusts the playout delay in response to a large change. For example, an implementation might adjust the playout delay if the transit delay changes by more than five times the current jitter estimate. The relative network transit time is used as part of the jitter calculation, so such observation is straightforward.

COMPENSATION FOR PACKET REORDERING

In extreme cases, jitter or route changes can result in packets being reordered in the network. As discussed in Chapter 2, Voice and Video Communication over Packet Networks, this is usually a rare occurrence, but it happens frequently enough that implementations need to be able to compensate for its effects and smoothly play out a media stream that contains out-of-order packets.

Reordering should not be an issue for correctly designed receivers: Packets are inserted into the playout buffer according to their RTP timestamp, irrespective of the order in which they arrive. If the playout delay is sufficiently large, they are played out in their correct sequence; otherwise they are discarded as any other late packet. If many packets are discarded because of reordering and late arrival, the standard jitter compensation algorithm will take care of adjusting the playout delay.

Adapting the Playout Point

There are two basic approaches to adapting the playout point: receivers can either slightly adjust the playout time for each frame, making continual small adjustments to the playout point, or they can insert or remove complete frames from the media stream, making a smaller number of large adjustments as they become necessary. No matter how the adjustment is made, the media stream is disrupted to some extent. The aim of the adaptation must be to minimize this disruption, which requires knowledge of the media stream; accordingly, audio and video playout adaptation strategies are discussed separately.

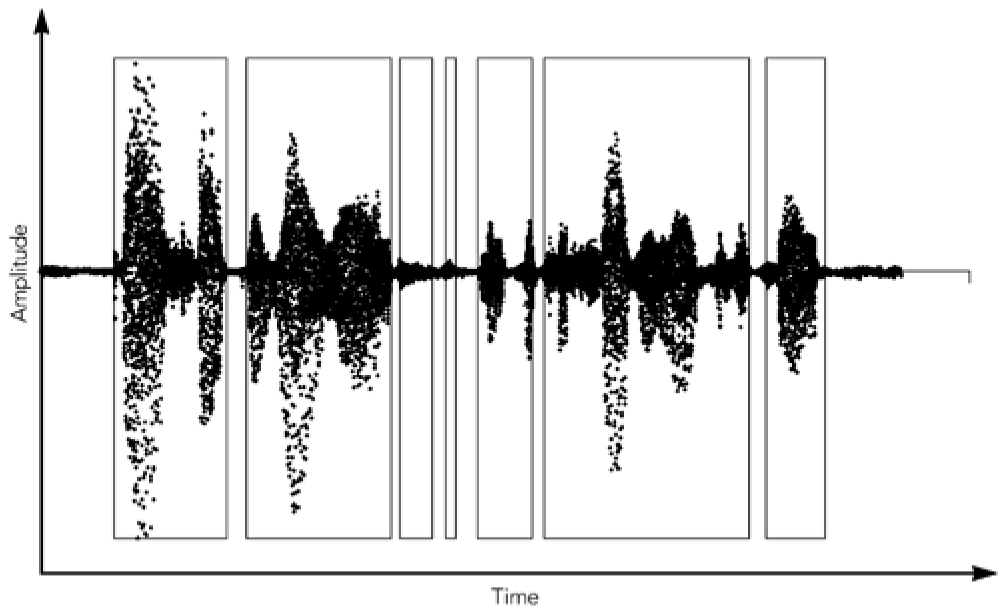
Playout Adaptation for Audio with Silence Suppression

Audio is a continuous media format, meaning that each audio frame occupies a certain amount of time, and the next is scheduled to start immediately after it finishes. There are no gaps between frames unless silence suppression is used, and hence there is no convenient time to adapt the playout delay. For this reason the presence of silence suppression has a significant effect on the design of audio playout buffer algorithms.

For conversational speech signals, an active speaker will generate talk spurts several hundred milliseconds in duration, separated by silent periods. Figure 6.16 shows the presence of talk spurts in a speech signal, and the gaps left between them.

The sender detects frames representing silent periods and suppresses the RTP packets that would otherwise be generated for those frames. The result is a sequence of packets with consecutive sequence numbers, but a jump in the RTP timestamp depending on the length of the silent period.

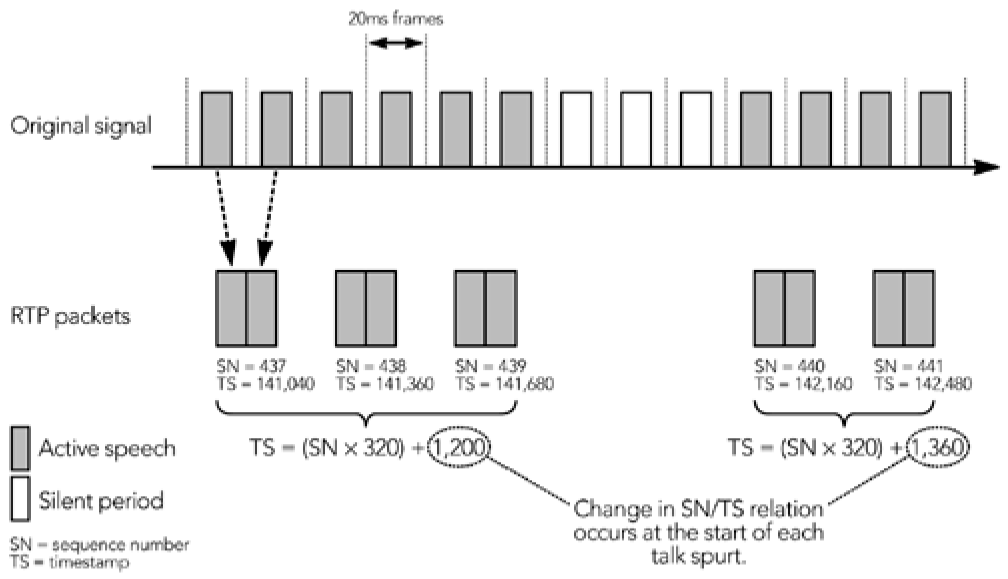
Figure 6.16. Talk Spurts in a Speech Signal



Adjusting the playout point during a talk spurt will cause an audible glitch in the output, but a small change in the length of the silent period between talk spurts will not be noticeable.[92] This is the key point to remember in the design of a playout algorithm for an audio tool: If possible, adjust the playout point only during silent periods.

It is usually a simple matter for a receiver to detect the start of a talk spurt, because the sender is required to set the marker bit on the first packet after a silent period, providing an explicit indication of the start of a talk spurt. Sometimes, however, the first packet in a talk spurt is lost. It is usually still possible to detect that a new talk spurt has started, because the sequence number/timestamp relationship will change as shown in Figure 6.17, providing an implicit indication of the start of the talk spurt.

Figure 6.17. Implicit Indication of the Start of a Talk Spurt



Once the start of the talk spurt has been located, you may adjust the playout point by slightly changing the length of the silent period. The playout delay is then held constant for all of the packets in the talk spurt. The appropriate playout delay is calculated during each talk spurt and used to adapt the playout point for the following talk spurt, under the assumption that conditions are unlikely to change significantly between talk spurts.

Some speech codecs send low-rate comfort noise frames during silent periods so that the receiver can play appropriate background noise to achieve a more pleasant listening experience. The receipt of a comfort noise packet indicates the end of a talk spurt and a suitable time to adapt the playout delay. The length of the comfort noise period can be varied without significant effects on the audio quality. The RTP payload type does not usually indicate the comfort noise frames, so it is necessary to inspect the media data to detect their presence. Older codecs that do not have native comfort noise support may use the RTP payload format for comfort noise,[42] which is indicated by RTP payload type 13.

In exceptional cases it may be necessary to adapt during a talk spurt—for example, if multiple packets are being discarded because of late arrival. These cases are expected to be rare because talk spurts are relatively short and network conditions generally change slowly.

Combining these features produces pseudocode to determine an appropriate time to adjust the playout point, assuming that silence suppression is used, as follows:

```
int
should_adjust_playout(rtp_packet curr, rtp_packet prev, int contdrop) {
    if (curr->marker) {
        return TRUE; // Explicit indication of new talk spurt
    }
    delta_seq = curr->seq – prev->seq;
    delta_ts = curr->ts - prev->ts;
    if (delta_seq * inter_packet_gap != delta_ts) {
        return TRUE; // Implicit indication of new talk spurt
    }
    if (curr->pt == COMFORT_NOISE_PT) || is_comfort_noise(curr)) {
        return TRUE; // Between talk spurts
    }
    if (contdrop > CONSECUTIVE_DROP_THRESHOLD) {
        contdrop = 0;
        return TRUE; // Something has gone badly wrong, so adjust
    }
    return FALSE;
}
```

The variable `contdrop` counts the number of consecutive packets discarded because of inappropriate playout times—for example, if a route change causes packets to arrive too late for playout. An appropriate value for `CONSECUTIVE_DROP_THRESHOLD` is three packets.

If the function `should_adjust_playout()` returns `TRUE`, the receiver is either in a silent period or has miscalculated the playout point. If the calculated playout point has diverged from the currently used value, it should adjust the playout points for future packets, by changing their scheduled playout time. There is no need to generate fill-in data, only to continue playing silence/comfort noise until the next packet is scheduled (this is true even when adjustment has been triggered by multiple consecutive packet drops because this indicates that playout has stopped).

Care needs to be taken when the playout delay is being reduced because a significant change in conditions could bring the start of the next talk spurt so far forward that it would overlap with the end of the previous talk spurt. The amount of adaptation that may be performed is thus limited, because clipping the start of a talk spurt is not desirable.

Playout Adaptation for Audio without Silence Suppression

When receiving audio transmitted without silence suppression, the receiver must adapt the playout point while audio is being played out. The most desirable means of adaptation is tuning the local media clock to match that of the transmitter so that data can be played out directly. If this is not possible, because the necessary hardware support is lacking, the receiver will have to vary the playout point either by generating fill-in data to be inserted into the media stream or by removing some media data from the playout buffer. Either approach inevitably causes some disruption to the playout, and it is important to conceal the effects of adaptation to ensure that it does not disturb the listener.

There are several possible adaptation algorithms, depending on the nature of the output device and the resources of the receiver:

- The audio can be resampled in software to match the rate of the output device. A standard signal-processing text will provide various algorithms, depending on the desired quality and resource trade-off. This is a good, general-purpose solution.
- Sample-by-sample adjustments in the playout delay can be made on the basis of knowledge of the media content. For example, Hodson et al.[79] use a pattern-matching algorithm to detect pitch cycles in speech that are removed or duplicated to adapt the playout (pitch cycles are much shorter than complete frames, so this approach gives fine-grained adaptation). This approach can perform better than resampling, but it is highly content-specific.
- Complete frames can be inserted or deleted, as if packets were lost or duplicated. This algorithm is typically not of high quality, but it may be required if a hardware decoder designed for synchronous networks is used.

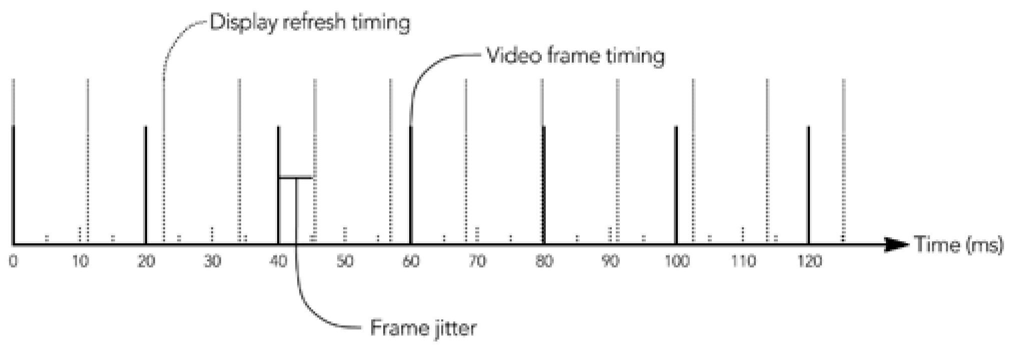
In the absence of silence suppression, there is no obvious time to adjust the playout point. Nevertheless, a receiver can still make intelligent choices regarding playout adaptation, by varying the playout at times where error concealment is more effective—for example, during a period of relative quiet or a period during which the signal is highly repetitive—depending on the codec and error concealment algorithm. Loss concealment strategies are described in detail in Chapter 8, Error Concealment.

Playout Adaptation for Video

Video is a discrete media format in which each frame samples the scene at a particular instant in time and the interval between frames is not recorded. The discrete nature of video provides flexibility in the playout algorithm, allowing the receiver to adapt the playout by slightly varying the interframe timing. Unfortunately, display devices typically operate at a fixed rate and limit possible presentation times. Video playout becomes a problem of minimizing the deviation between the intended and possible frame presentation instant.

For example, consider the problem of displaying a 50-frame-per-second video clip on a monitor with 85Hz refresh rate. In this case the monitor refresh times will not match the video playout times, causing unavoidable variation in the time at which frames are presented to the user, as shown in Figure 6.18. Only a change in the frame rate of the capture device or the refresh rate of the display can address this problem. In practice, the problem is often insoluble because video capture and playback devices often have hardware limits on the set of possible rates. Even when capture and playback devices have nominally the same rate, it may be necessary to adapt the playout according to the effects of jitter or clock skew.

Figure 6.18. Mismatch between Media Frame Times and Output Device Timing



There are three possible occasions for adaptation to occur: (1) when the display device has a higher frame rate than the capture device, (2) when the display device has a lower frame rate than the capture device, and (3) when the display and capture devices run at the same nominal rate.

If the display device has a higher frame rate than the capture device, possible presentation times will surround the desired time, and each frame can be mapped to a unique display refresh interval. The simplest approach is to display frames at the refresh interval closest to their playout time. One can achieve better results by moving frames in the direction of any required playout adjustment: displaying a frame at the refresh interval following its playout time if the receiver clock is relatively fast, or at the earlier interval if the receiver clock is slow. Intermediate refresh intervals, when no new frames are received from the sender, can be filled by repetition of the previous frame.

If the display device has a lower frame rate than the capture device, displaying all frames is not possible, and the receiver must discard some data. For example, the receiver may calculate the difference between the frame playout time and the display times, and choose to display the subset of frames closest to possible display times.

If the display device and capture device run at the same rate, the playout buffer can be slipped so that frame presentation times align with the display refresh times, with the slippage providing some degree of jitter buffering delay. This is an uncommon case: Clock skew is common, and periodic jitter adjustments may upset the time-line. Depending on the direction of the required adjustment, the receiver must then either insert or remove a frame to compensate.

One can insert a frame into the playout sequence simply by repeating it for two intervals. Likewise, removing a frame from the sequence is a straightforward matter. (Note that other frames may be predicted from a nondisplayed frame, so it is often impossible to completely discard a frame without decoding, except for the last predicted frame before a full frame.)

No matter how the adjustment is made, note that the human visual system is somewhat sensitive to nonuniform playout, and the receiver should seek to keep the interframe presentation times as uniform as possible to prevent disturbing artifacts. Inserting or removing frames should be considered a last-resort operation, after smaller playout adjustments (choosing an earlier or later display frame time) have proven insufficient.

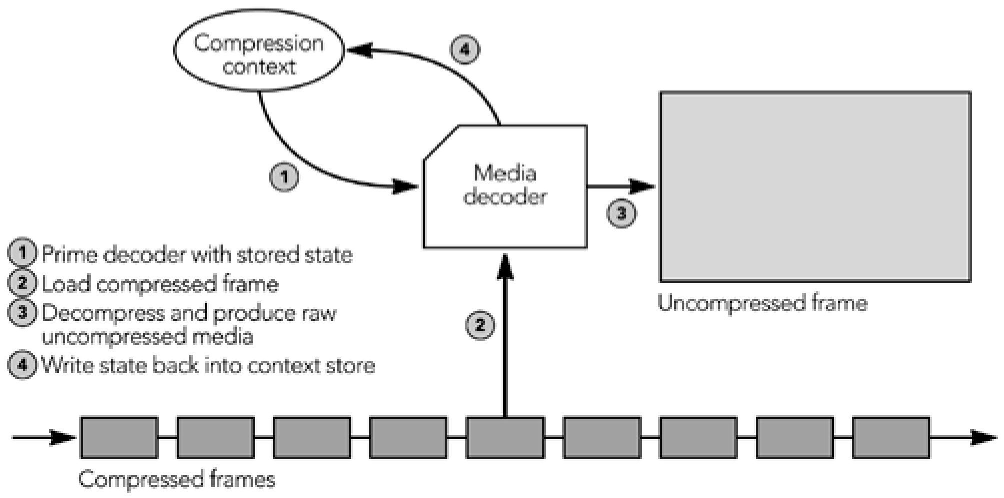
Decoding, Mixing, and Playout

The final stages of the playout process are to decode the compressed media, mix media streams together if there are fewer output channels than active sources, and finally play the media to the user. This section considers each stage in turn.

Decoding

For each active source the application must maintain an instantiation of the media decoder, comprising the decompression routines along with state known as the *compression context*. The decoder may be an actual hardware device or a software function, depending on the system. It converts each compressed frame into uncompressed media data, on the basis of the data in the frame and the compression context. As each frame is decoded, the compression context for the source is updated as shown in Figure 6.19.

Figure 6.19. Operation of a Media Decoder


Compression context
Media decoder
Uncompressed frame
1 Prime decoder with stored state
2 Load compressed frame
3 Decompress and produce raw uncompressed media
4 Write state back into context store
Compressed frames

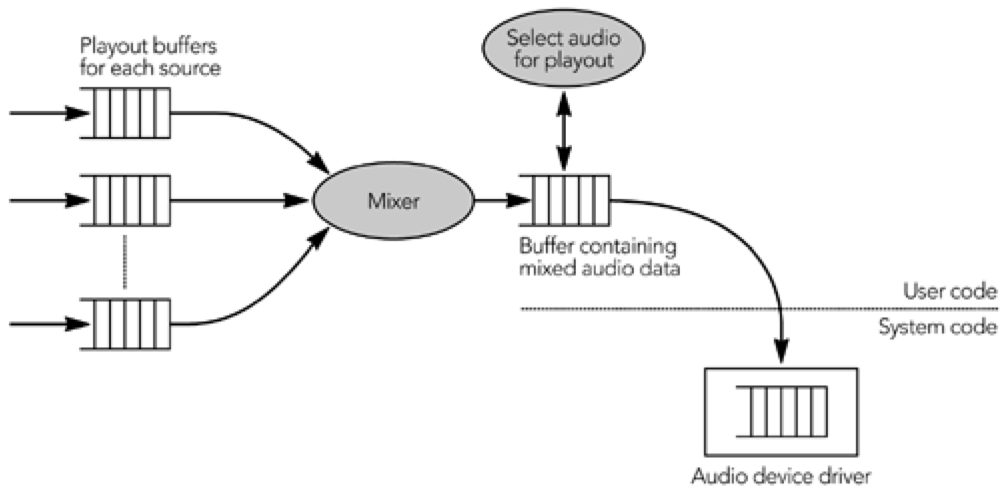
The presence of accurate state in the decompression context is fundamental to correct operation of the decoder, and codecs will produce incorrect results if the context is missing or damaged. This is most often an issue if some data packets are lost because there will be a frame that cannot be decoded. The result will be a gap in the playout where that frame should have been, but the decompression context will also be invalidated and the following frames will be corrupted.

Depending on the codec, it may be possible to feed it an indication that a frame has been lost, allowing the decoder to better repair the context and reduce the damage to the media stream (for example, many speech codecs have the notion of erasure frames to signal losses). Otherwise the receiver should try to repair the context and conceal the effects of the loss, as discussed in Chapter 8, Error Concealment. Many loss concealment algorithms operate on the uncompressed media data, after decoding and before mixing and playout operation.

Audio Mixing

Mixing is the process of combining multiple media streams into one, for output. This is primarily an issue for audio applications because most systems have only a single set of speakers but multiple active sources—for example, in a multiparty teleconference. Once audio streams have been decoded, they must be mixed together before being written to the audio device. The final stages of an audio tool will typically be structured somewhat as shown in Figure 6.20. The decoder produces uncompressed audio data on a per-source basis, written into a per-source playout buffer, and the mixer combines the results into a single buffer for playout (these steps can, of course, be combined into one if the decoder understands the mixing process). Mixing can occur at any time after the media has been decoded, and before it is due for playout.

Figure 6.20. Audio Mixing



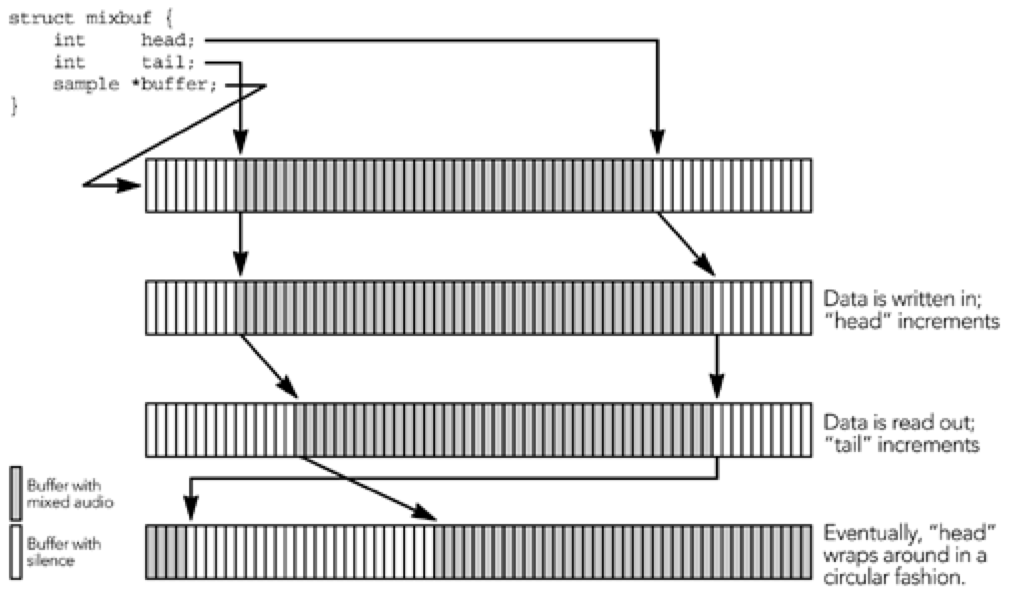
The mix buffer is initially empty—that is, full of silence—and each participant's audio is mixed into the buffer in turn. The simplest approach to mixing is *saturating addition*, in which each participant's audio is added to the buffer in turn, with overflow conditions saturating at extreme values. In pseudocode, assuming 16-bit samples and mixing a new participant (`src`) into the buffer (`mix_buffer`), this becomes

```
audio_mix(sample *mix_buffer, sample *src, int len)
{
    int i, tmp;
    for(i = 0; i < len; i++) {
        tmp = mix_buffer[i] + src[i];
        if (tmp > 32767) {
            tmp = 32767;
        } else if (tmp < -32768) {
            tmp = -32768;
        }
        mix_buffer[i] = tmp;
    }
}
```

Other algorithms are possible if higher-fidelity mixing is required. Mixing samples is a task for which SIMD processors often have instructions. For instance, the Intel MMX (Multimedia Extensions) instructions include saturating-add instructions that add four 16-bit samples at a time, and because the mixing loop no longer has the branch checks, the performance can be up to ten times faster.

The actual mix buffer can be implemented as a circular buffer. The buffer is implemented as an array with start and end pointers, wrapping around to give the illusion of a continuous buffer (see Figure 6.21).

Figure 6.21. Implementation of a Circular Mix Buffer

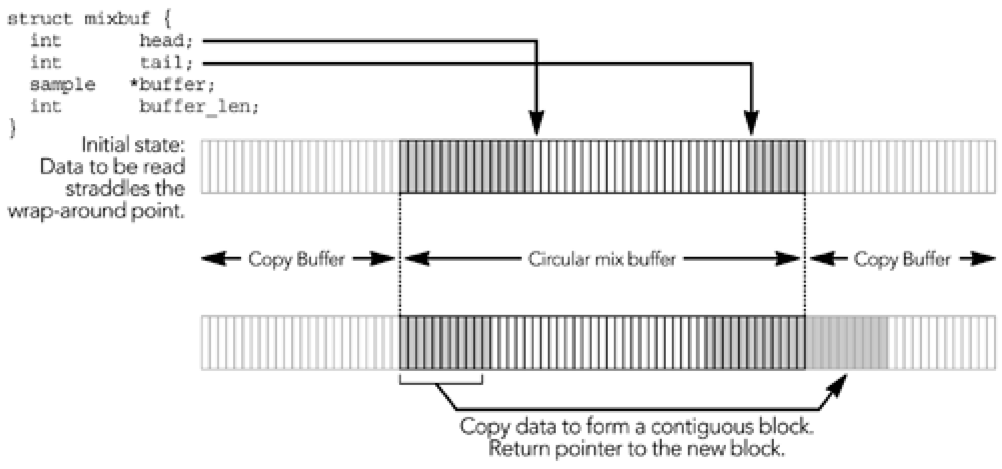


```
struct mixbuf {
    int     head;
    int     tail;
    sample *buffer;
}
```

Data is written in; "head" increments
Data is read out; "tail" increments
Buffer with mixed audio
Buffer with silence
Eventually, "head" wraps around in a circular fashion.

A limitation of the simple circular buffer is that it cannot always make a continuous buffer available for readout. Instead, as the readout nears the wrap-around point, it will be necessary to return two blocks of mixed data: one from the end of the circular buffer, one from the beginning. The need to return two buffers can be avoided if an array of twice the required size is allocated. If the readout routine requests a block that includes the wrap-around point in the circular buffer, the mixer can copy data into the additional space and return a pointer to a continuous block of memory as shown in Figure 6.22. This requires an additional copy of the audio data, up to half the size of the circular buffer, but allows the readout to return a single contiguous buffer, simplifying code that uses the mixer. Normal operation of the circular buffer is unchanged, except for a single copy when data is being read out.

Figure 6.22. Implementation of a Circular Mix Buffer with Additional Copy Buffer



```
struct mixbuf {
  int       head;
  int       tail;
  sample   *buffer;
  int       buffer_len;
}
```

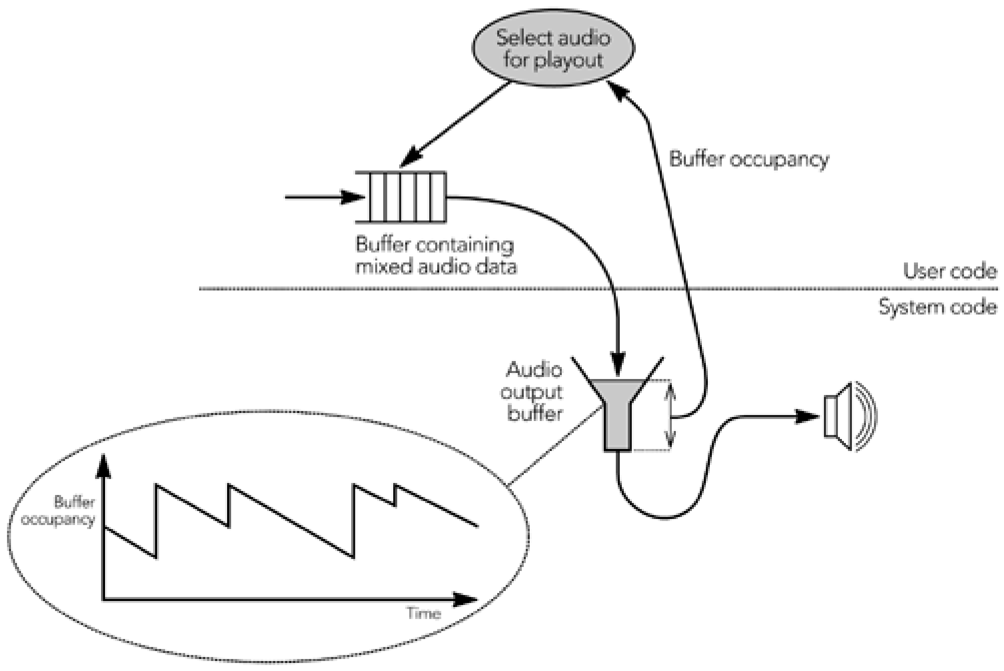
Initial state: Data to be read straddles the wrap-around point.
Copy Buffer
Circular mix buffer
Copy Buffer
Copy data to form a contiguous block. Return pointer to the new block.

Audio Playout

The process by which audio is played to the user is typically asynchronous, allowing the system to play one frame of audio while processing the next. This capability is essential to normal operation because it allows continuous playback even though the application is busy with RTP and media processing. It also shields the application from variations in the behavior of the system, perhaps due to other applications running on that system.

Asynchronous playout is especially important on general-purpose operating systems with limited support for multimedia applications. These systems are typically designed to give good average response, but often they have undesirable worst-case behavior, and typically they cannot guarantee that real-time applications are scheduled appropriately. An application can use asynchronous playout to its advantage, using the audio DMA (directory memory access) hardware to maintain continual playout.[83] As shown in Figure 6.23, an application can monitor the occupancy of the output buffer and adjust the amount it writes to the audio device according to the time since it was last scheduled, such that the buffer occupancy after each iteration is constant.

Figure 6.23. Use of an Audio DMA Buffer for Continual Playout



If the application detects a period of unusual scheduling latency—perhaps due to heavy disk activity on the system—it can preemptively increase the size of the audio DMA buffer, up to the limit imposed by the playout point. If the operating system does not allow direct monitoring of the amount of audio buffered and awaiting playout, it may be possible to derive an estimate from the amount of audio waiting to be read for the encoding side of the application. In many cases, audio playback and recording are driven from the same hardware clock, so an application can count the number of audio samples it records and use this information to derive the occupancy of the playback buffer. Careful monitoring of the audio DMA buffer can ensure continual playout in all but the most extreme environments.

Video Playout

Video playout is largely dictated by the refresh rate of the display, which determines the maximum time between the application writing to the output buffer and the image being presented to the user. The key to smooth video playout is twofold: (1) Frames should be presented at uniform rate, and (2) changes to a frame should be avoided while the video is being rendered. The first point is a matter for the playout buffer, selecting the appropriate display time as described in the section Playout Adaptation for Video earlier in this chapter.

The second point relates to the display: Frames are not presented instantaneously; instead they are drawn in a series of scan lines, left to right, top to bottom. This serial presentation allows the possibility that the application will be able to change a frame while it is being displayed, causing a glitch in the output. Double buffering can solve this problem, one buffer being used to compose a frame while the second buffer is being displayed. The two buffers are switched between frames, synchronized with the interframe gap. The means by which double buffering is achieved is system dependent but usually part of the video display API.

Summary

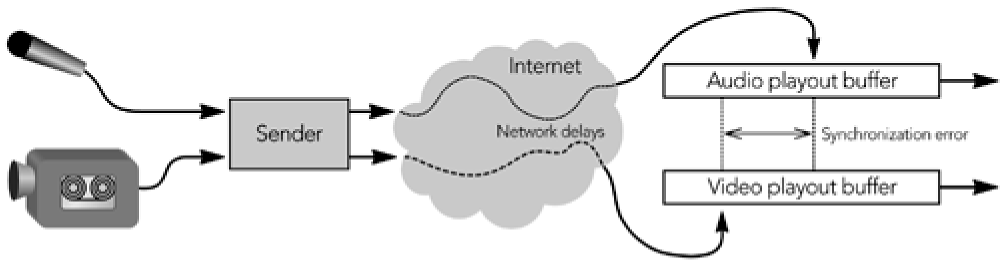
This chapter has described the fundamental behavior of RTP senders and receivers in some detail, focusing especially on the design of the receiver's playout buffer. The playout buffer design is relatively simple for streaming applications, which can accept several hundred milliseconds (or perhaps even seconds) of delay. For applications designed for interactive use, however, the play-out buffer is critical to achieving good performance. These applications need low latency, and hence playout buffer delays of a few tens of milliseconds, taxing the design of algorithms that must balance the needs of low delay with the need to avoid discarding late packets if possible.

RTP systems place a lot of intelligence in the end systems, leaving them to compensate for the variability inherent in a best-effort packet network. Recognizing this is the key to good performance: A well-designed, robust implementation can perform significantly better than a naive design.

Chapter 7. Lip Synchronization

A multimedia session comprises several media streams, and in RTP each is transported via a separate RTP session. Because the delays associated with different encoding formats vary greatly, and because the streams are transported separately across the network, the media will tend to have different playout times. To present multiple media in a synchronized fashion, receivers must realign the streams as shown in Figure 7.1. This chapter describes how RTP provides the information needed to facilitate the synchronization of multiple media streams. The typical use for this technique is to align audio and video streams to provide lip synchronization, although the techniques described may be applied to the synchronization of any set of media streams.

Figure 7.1. Media Flows and the Need for Synchronization



A common question is why media streams are delivered separately, forcing the receiver to resynchronize them, when they could be delivered bundled together and presynchronized. The reasons include the desire to treat audio and video separately in the network, and the heterogeneity of networks, codecs, and application requirements.

It is often appropriate to treat audio and video differently at the transport level to reflect the preferences of the sender or receiver. In a video conference, for instance, the participants often favor audio over video. In a best-effort network, this preference may be reflected in differing amounts of error correction applied to each stream; in an integrated services network, using RSVP (Resource ReSerVation Protocol),[11] this could correspond to reservations with differing quality-of-service (QoS) guarantees for audio and video; and in a Differentiated Services network,[23,24] the audio and video could be assigned to different priority classes. If the different media types were bundled together, these options would either cease to exist or become considerably harder to implement. Similarly, if bundled transport were used, all receivers would have to receive all media; it would not be possible for some participants to receive only the audio, while others received both audio and video. This ability becomes an issue for multiparty sessions, especially those using multicast distribution.

However, even if it is appropriate to use identical QoS for all media, and even if all receivers want to receive all media, the properties of codecs and playout algorithms are such that some type of synchronization step is usually required. For example, audio and video decoders take different and varying amounts of time to decompress the media, perform error correction, and render the data for presentation. Also the means by which the playout buffering delay is adjusted varies with the media format, as we learned in Chapter 6, Media Capture, Playout, and Timing. Each of these processes can affect the playout time and can result in loss of synchronization between audio and video.

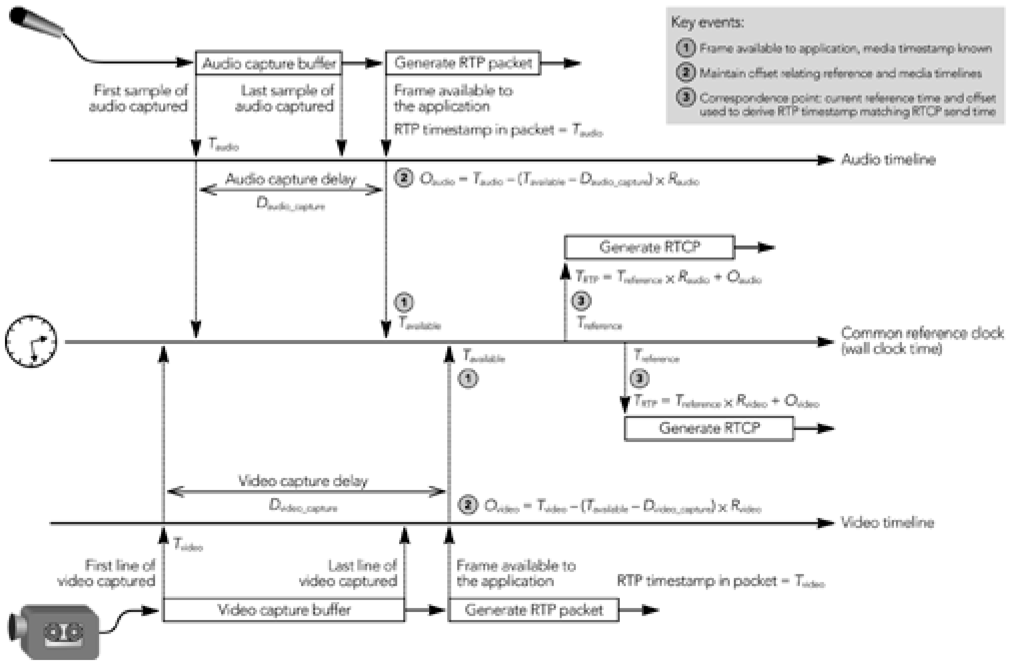
The result is that some type of synchronization function is needed, even if the media are bundled for delivery. As a result, we may as well deliver media separately, allowing them to be treated differently in the network because doing so does not add significant complexity to the receiver.

With these issues in mind, we now turn to a discussion of the synchronization process. There are two parts to this process: The sender is required to assign a common reference clock to the streams, and the receiver needs to resynchronize the media, undoing the timing disruption caused by the network. First up are discussions of the sender and the receiver in turn, followed by some comments on the synchronization accuracy required for common applications.

Sender Behavior

The sender enables synchronization of media streams at the receiver by running a common reference clock and periodically announcing, through RTCP, the relationship between the reference clock time and the media stream time, as well as the identities of the streams to be synchronized. The reference clock runs at a fixed rate; correspondence points between the reference clock and the media stream allow the receiver to work out the relative timing relationship between the media streams. This process is shown in Figure 7.2.

Figure 7.2. Mapping Media Time Lines to a Common Clock at the Sender


Key events:
1 Frame available to application, media timestamp known
2 Maintain offset relating reference and media timelines
3 Correspondence point: current reference time and offset used to derive RTP timestamp matching RTCP send time
Audio capture buffer
Generate RTP packet
First sample of audio captured
Last sample of audio captured
Frame available to the application
RTP timestamp in packet = T_{audio}
T_{audio}
Audio timeline
Audio capture delay
$D_{audio\_capture}$
2 $O_{audio} = T_{audio} - (T_{available} - D_{audio\_capture}) \times R_{audio}$
Generate RTCP
$T_{RTP} = T_{reference} \times R_{audio} + O_{audio}$
3
$T_{reference}$
1
$T_{available}$
Common reference clock (wall clock time)
$T_{available}$
1
$T_{reference}$
3
$T_{RTP} = T_{reference} \times R_{video} + O_{video}$
Generate RTCP
Video capture delay
$D_{video\_capture}$
2 $O_{video} = T_{video} - (T_{available} - D_{video\_capture}) \times R_{video}$
Video timeline
T_{video}
First line of video captured
Last line of video captured
Frame available to the application
RTP timestamp in packet = T_{video}
Video capture buffer
Generate RTP packet

The correspondence between reference clock and media clock is noted when each RTCP packet is generated: A sampling of the reference clock, $T_{reference}$, is included in the packet along with a calculated RTP timestamp, $T_{RTP} = T_{reference} \times R_{audio} + O_{audio}$.

The multiplication must be made modulo 2^{32}, to restrict the result to the range of the 32-bit RTP timestamp. The offset is calculated as $O_{audio} = T_{audio} - (T_{available} - D_{audio\_capture}) \times R_{audio}$, being the conversion factor between media and reference timelines. Operating system latencies can delay $T_{available}$ and cause variation in the offset, which should be filtered by the application to choose a minimum value. (The obvious changes to the formulae are made in the case of video.)

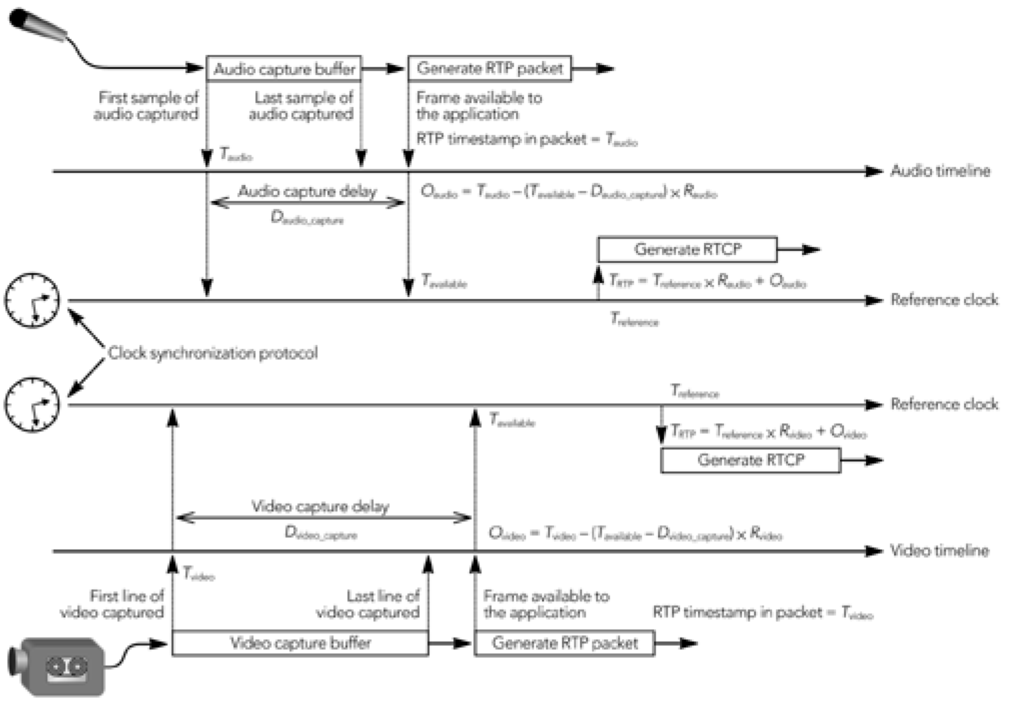
Each application on the sender that is transmitting RTP streams needs access to the common reference clock, $T_{reference}$, and must identify its media with reference to a canonical source identifier. The sending applications should be aware of the media capture delay—for example, $D_{audio\_capture}$—because it can be significant and should be taken into account in the calculation and announcement of the relationship between reference clock times and media clock times.

The common reference clock is the "wall clock" time used by RTCP. It takes the form of an NTP-format timestamp, counting seconds and fractions of a second since midnight UTC (Coordinated Universal Time) on January 1, 1900.[5] (Senders that have no knowledge of the wall clock time may use a system-specific clock such as "system uptime" to calculate NTP-format timestamps as an alternative; the choice of a reference clock does not affect synchronization, as long as it is done consistently for all media.) Senders periodically establish a correspondence between the media clock for each stream and the common reference clock; this is communicated to receivers via RTCP sender report packets as described in the section titled RTCP SR: Sender Reports in Chapter 5, RTP Control Protocol.

In typical scenarios, there is no requirement for the sender or receiver to be synchronized to an external clock. In particular, although the wall clock time in RTCP sender report packets uses the format of an NTP timestamp, it is *not* required to be synchronized to an NTP time source. Sender and receiver clocks never have to be synchronized to each other. Receivers do not care about the absolute value of the NTP format timestamp in RTCP sender report packets, only that the clock is common between media, and of sufficient accuracy and stability to allow synchronization.

Synchronized clocks are required only when media streams generated by different hosts are being synchronized. An example would be multiple cameras giving different viewpoints on a scene, connected to separate hosts with independent network connections. In this instance the sending hosts need to use a time protocol or some other means to align their reference clocks to a common time base. RTP does not mandate any particular method of defining that time base, but the Network Time Protocol[5] may be appropriate, depending on the degree of synchronization required. Figure 7.3 shows the requirements for clock synchronization when media streams from different hosts are to be synchronized at playout.

Figure 7.3. Synchronization of Media Generated by Different Hosts


Audio capture buffer
Generate RTP packet
First sample of audio captured
Last sample of audio captured
Frame available to the application
RTP timestamp in packet = T_{audio}
T_{audio}
Audio timeline
Audio capture delay
$D_{audio\_capture}$
$O_{audio} = T_{audio} - (T_{available} - D_{audio\_capture}) \times R_{audio}$
Generate RTCP
$T_{available}$
$T_{RTP} = T_{reference} \times R_{audio} + O_{audio}$
Reference clock
$T_{reference}$
Clock synchronization protocol
$T_{reference}$
Reference clock
$T_{available}$
$T_{RTP} = T_{reference} \times R_{video} + O_{video}$
Generate RTCP
Video capture delay
$D_{video\_capture}$
$O_{video} = T_{video} - (T_{available} - D_{video\_capture}) \times R_{video}$
Video timeline
T_{video}
First line of video captured
Last line of video captured
Frame available to the application
RTP timestamp in packet = T_{video}
Video capture buffer
Generate RTP packet

The other requirement for synchronization is to identify sources that are to be synchronized. RTP does this by giving the related sources a shared name, so a receiver knows which streams it should attempt to synchronize and which are independent. Each RTP packet contains a synchronization source (SSRC) identifier to associate the source with a media time base. The SSRC identifier is chosen randomly and will not be the same for all the media streams to be synchronized (it may also change during a session if identifiers collide, as explained in Chapter 4, RTP Data Transfer Protocol). A mapping from SSRC identifiers to a persistent canonical name (CNAME) is provided by RTCP source description (SDES) packets. A sender should ensure that RTP sessions to be synchronized on playout have a common CNAME so that receivers know to align the media.

The canonical name is chosen algorithmically according to the user name and network address of the source host (see the section RTCP SDES: Source Description in Chapter 5, RTP Control Protocol). If multiple media streams are being generated by a single host, the task of ensuring that they have a common CNAME, and hence can be synchronized, is simple. If the goal is to synchronize media streams generated by several hosts—for example, if one host is capturing and transmitting audio while another transmits video—the choice of CNAME is less obvious because the default method in the RTP standard would require each host to use its own IP address as part of the CNAME. The solution is for the hosts to conspire in choosing a common CNAME for all streams that are to be synchronized, even if this means that some hosts use a CNAME that doesn't match their network address. The mechanism by which this conspiracy happens is not specified by RTP: One solution might be to use the lowest-numbered IP address of the hosts when constructing the CNAME; another might be for the audio to use the CNAME of the video host (or vice versa). This coordination would typically be provided by a session control protocol—for example, SIP or H.323—outside the scope of RTP. A session control protocol could also indicate which streams should be synchronized by a method that does not rely on the CNAME.

Receiver Behavior

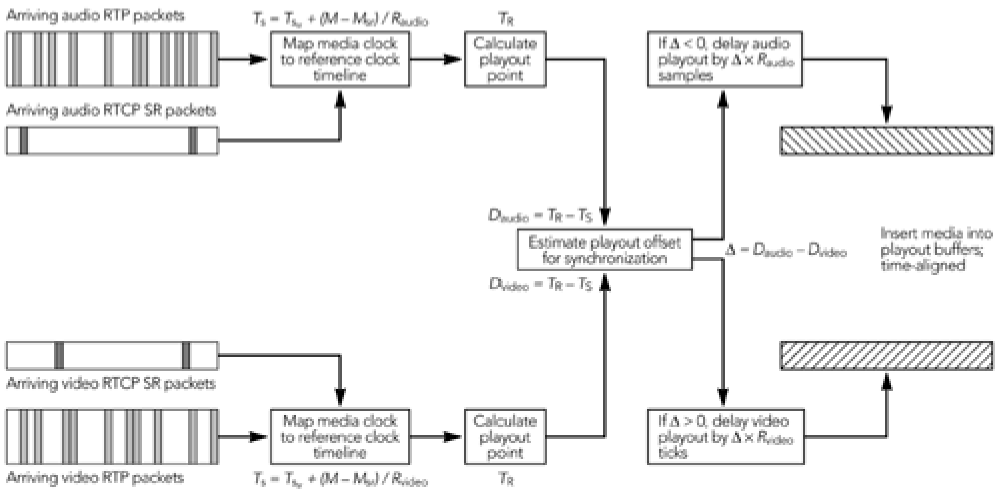
A receiver is expected to determine which media streams should be synchronized, and to align their presentation, on the basis of the information conveyed to it in RTCP packets.

The first part of the process—determining which streams are to be synchronized—is straightforward. The receiver synchronizes those streams that the sender has given the same CNAME in their RTCP source description packets, as described in the previous section. Because RTCP packets are sent every few seconds, there may be a delay between receipt of the first data packet and receipt of the RTCP packet that indicates that particular streams are to be synchronized. The receiver can play out the media data during this time, but it is unable to synchronize them because it doesn't have the required information.

More complex is the actual synchronization operation, in which the receiver time-aligns audio and video for presentation. This operation is triggered by the reception of RTCP sender report packets containing the mapping between the media clock and a reference clock common to both media. Once this mapping has been determined for both audio and video streams, the receiver has the information needed to synchronize playout.

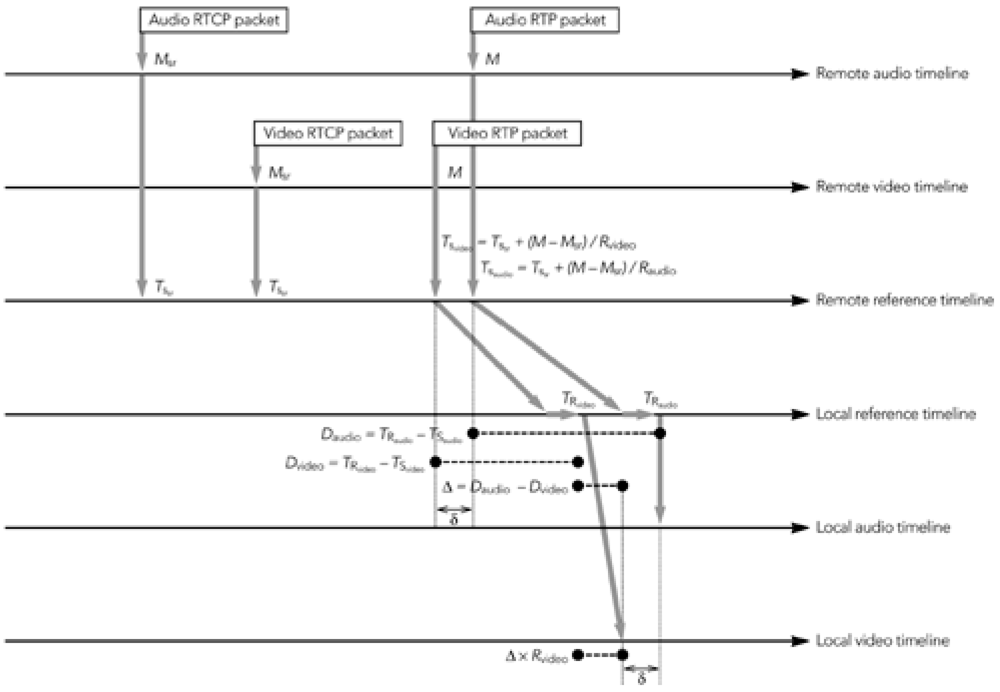
The first step of lip synchronization is to determine, for each stream to be synchronized, when the media data corresponding to a particular reference time is to be presented to the user. Because of differences in the network behavior or other reasons, it is likely that data from two streams that was captured at the same instant will not be scheduled for presentation at the same time if the playout times are determined independently according to the methods described in Chapter 6, Media Capture, Playout, and Timing. The playout time for one stream therefore has to be adjusted to match the other. This adjustment translates into an offset to be added to the playout buffering delay for one stream, such that the media are played out in time alignment. Figures 7.4 and 7.5 illustrate the process.

Figure 7.4. Lip Synchronization at the Receiver



Arriving audio RTP packets
$T_S = T_{S_{sr}} + (M - M_{sr}) / R_{audio}$
Map media clock to reference clock timeline
T_R
Calculate playout point
If Δ < 0, delay audio playout by Δ × R_{audio} samples
Arriving audio RTCP SR packets
$D_{audio} = T_R - T_S$
Estimate playout offset for synchronization
Δ = $D_{audio} - D_{video}$
Insert media into playout buffers; time-aligned
$D_{video} = T_R - T_S$
Arriving video RTCP SR packets
Map media clock to reference clock timeline
Calculate playout point
If Δ > 0, delay video playout by Δ × R_{video} ticks
Arriving video RTP packets
$T_S = T_{S_{sr}} + (M - M_{sr}) / R_{video}$
T_R

Figure 7.5. Mapping between Timelines to Achieve Lip Synchronization at the Receiver



The receiver first observes the mapping between the media clock and reference clock as assigned by the sender, for each media stream it is to synchronize. This mapping is conveyed to the receiver in periodic RTCP sender report packets, and because the nominal rate of the media clock is known from the payload format, the receiver can calculate the reference clock capture time for any data packet once it has received an RTCP sender report from that source. When an RTP data packet with media timestamp *M* is received, the corresponding reference clock capture time, T_S (the RTP timestamp mapped to the reference timeline), can be calculated as follows:

$$T_S = \frac{T_{Ssr} + (M - M_{sr})}{R}$$

where M_{sr} is the media (RTP) timestamp in the last RTCP sender report packet, T_{Ssr} is the corresponding reference clock (NTP) timestamp in seconds (and fractions of a second) from the sender report packet, and R is the nominal media timestamp clock rate in hertz. (Note that this calculation is invalid if more than 2^{32} ticks of the media clock have elapsed between M_{sr} and M, but that this is not expected to occur in typical use.)

The receiver also calculates the presentation time for any particular packet, T_R, according to its local reference clock. This is equal to the RTP timestamp of the packet, mapped to the receiver's reference clock timeline as described earlier, plus the playout buffering delay in seconds and any delay due to the decoding, mixing, and rendering processes. It is important to take into account all aspects of the delay until actual presentation on the display or loudspeaker, if accurate synchronization is to be achieved. In particular, the time taken to decode and render is often significant and should be accounted for as described in Chapter 6, Media Capture, Playout, and Timing.

Once the capture and playout times are known according to the common reference timeline, the receiver can estimate the relative delay between media capture and playout for each stream. If data sampled at time T_S according to the sender's reference clock is presented at time T_R according to the receiver's reference clock, the difference between them, $D = T_R - T_S$, is the relative capture-to-playout delay in seconds. Because the reference clocks at the sender and receiver are not synchronized, this delay includes an offset that is unknown but can be ignored because it is common across all streams and we are interested in only the relative delay between streams.

Once the relative capture-to-playout delay has been estimated for both audio and video streams, a synchronization delay, $D = D_{audio} - D_{video}$, is derived. If the synchronization delay is zero, the streams are synchronized. A nonzero value implies that one stream is being played out ahead of the other, and the synchronization delay gives the relative offset in seconds.

For the media stream that is ahead, the synchronization delay (in seconds) is multiplied by the nominal media clock rate, R, to convert it into media timestamp units, and then it is applied as a constant offset to the playout calculation for that media stream, delaying playout to match the other stream. The result is that packets for one stream reside longer in the playout buffer at the receiver, to compensate for either faster processing in other parts of the system or reduced network delay, and are presented at the same time as packets from the other stream.

The receiver can choose to adjust the playout of either audio or video, depending on its priorities, the relative delay of the streams, and the relative playout disturbance caused by an adjustment to each stream. With many common codecs, the video encoding and decoding times are the dominant factors, but audio is more sensitive to playout adjustments. In this case it may be appropriate to make an initial adjustment by delaying the audio to match the approximate video presentation time, followed by small adjustments to the video playout point to fine-tune the presentation. The relative priorities and delays may be different in other scenarios, depending on the codec, capture, and playout devices, and each application should make a choice based on its particular environment and delay budget.

The synchronization delay should be recalculated when the playout delay for any of the streams is adjusted because any change in the playout delay will affect the relative delay of the two streams. The offset should also be recalculated whenever a new mapping between media time and reference time, in the form of an RTCP sender report packet, is received. A robust receiver does not necessarily trust the sender to keep jitter out of the mapping provided in RTCP sender report packets, and it will filter the sequence of mappings to remove any jitter. An appropriate filter might track the minimum offset between media time and reference time, to avoid a common implementation problem in which the sender uses the media time of the previous data packet in the mapping, instead of the actual time when the sender report packet was generated.

A change in the mapping offset causes a change in the playout point and may require either insertion or deletion of media data from the stream. As with any change to the playout point, such changes should be timed with care, to reduce the impact on the media quality. The issues discussed in the section titled Adapting the Playout Point in Chapter 6, Media Capture, Playout, and Timing, are relevant here.

Synchronization Accuracy

When you're implementing synchronization, the question of accuracy arises: What is the acceptable offset between streams that are supposed to be synchronized? There is no simple answer, unfortunately, because human perception of synchronization depends on what is being synchronized and on the task being performed. For example, the requirements for lip synchronization between audio and video are relatively lax and vary with the video quality and frame rate, whereas the requirements for synchronization of multiple related audio streams are strict.

If the goal is to synchronize a single audio track to a single video track, then synchronization accurate to a few tens of milliseconds is typically sufficient.

Experiments with video conferencing suggest that synchronization errors on the order of 80 milliseconds to 100 milliseconds are below the limit of human perception (see Kouvelas et al. 1996,[84] for example), although this clearly depends on the task being performed and on the picture quality and frame rate. Higher-quality pictures, and video with higher frame rates, make lack of synchronization more noticeable because it is easier to see lip motion. Similarly, if frame rates are low—less than approximately five frames per second—there is little need for lip synchronization because lip motion cannot be perceived as speech, although other visual cues may expose the lack of synchronization.

If the application is synchronizing several audio tracks—for example, the channels in a surround-sound presentation—requirements for synchronization are much stricter. In this scenario, playout must be accurate to a single sample; the slightest synchronization error will be noticeable because of the phase difference between the signals, which disrupts the apparent position of the source.

The information provided by RTP is sufficient for sample-accurate synchronization, if desired, provided that the sender has correctly calculated the mapping between media time and reference time contained in RTCP sender report packets, and provided that the receiver has an appropriate synchronization algorithm. Whether this is achievable in practice is a quality-of-implementation issue.

Summary

Synchronization—like many other features of RTP—is performed by the end systems, which must correct for the variability inherent in a best-effort packet network. This chapter has described how senders signal the time alignment of media, and the process by which receivers can resynchronize media streams. It has also discussed the synchronization accuracy required for lip synchronization between audio and video, and for synchronization among multiple audio streams.

Chapter 8. Error Concealment

- Techniques for Audio Loss Concealment
- Techniques for Video Loss Concealment
- Interleaving

Earlier chapters described how RTP running over UDP/IP provides an unreliable packet delivery service, and how this means an application may have to deal with an incomplete media stream. There are two things the application can do when packet loss occurs: try to correct the error, or try to conceal it. Error correction is discussed in Chapter 9. In this chapter we discuss techniques by which a receiver can conceal the effects of loss.

Techniques for Audio Loss Concealment

When an RTP packet containing audio data—whether music or speech—is lost, the receiver has to generate a replacement to preserve the timing of the media stream. This can be done in many ways, and the choice of concealment algorithm can have a significant impact on the perceived quality of the system in the case of loss.

Measuring Audio Quality

Human perception of sound is a complex process, and the perceptual significance of distortion depends not just on the amount the signal has changed, but also on the type of damage caused, and where it occurred in the signal. Some types of distortion are more noticeable to listeners than others, even if—by some objective measure—they change the signal by the same amount. It is also common for different listeners to perceive a particular type of distortion in different ways, and to rate concealment schemes differently depending on the material to which they are applied.

This makes it very difficult to devise objective quality measurements for different repair schemes. It is not sufficient to measure the difference between the original waveform from the source and the waveform recovered at the receiver, because the perceived quality has no direct relation to the differences in the waveforms. Simple measures, such as the signal-to-noise ratio, are effectively useless. More complex schemes (for example, those in ITU recommendations P.861 and P.862[63], [64]) give results that are approximately correct for speech, but even these are not 100% reliable.

When objective measurements fail, we need to resort to subjective tests. By conducting listening tests with a wide range of subjects, materials, and error conditions, we can measure the effectiveness of different repair schemes in a manner meaningful to the listener. These tests involve playing different types of music, or a range of words, phrases, and sentences subject to different error conditions and concealment techniques, with the listener rating their quality and/or intelligibility according to a particular scale.

The choices of material and rating scale depend on what is being measured. If you are attempting to measure the perceived quality of speech ("does it sound good?"), then rating samples on the basis of a Mean Opinion Score (MOS) is appropriate. The MOS is a five-point rating scale, the results of which are converted into numeric form (excellent = 5, good = 4, fair = 3, poor = 2, bad = 1) and averaged across all the listeners, giving a numeric result between 1 and 5. For the results to be statistically valid, it is necessary to make a large number of tests comparing different samples.

Typical MOS scores for unimpaired speech are 4.2 for the G.711 codec (that is, standard telephone quality) and between 3.5 and 4 for mobile telephony (for example, GSM, QCELP). Packet loss will lower these numbers, with the degree of loss and the type of concealment determining the actual result.

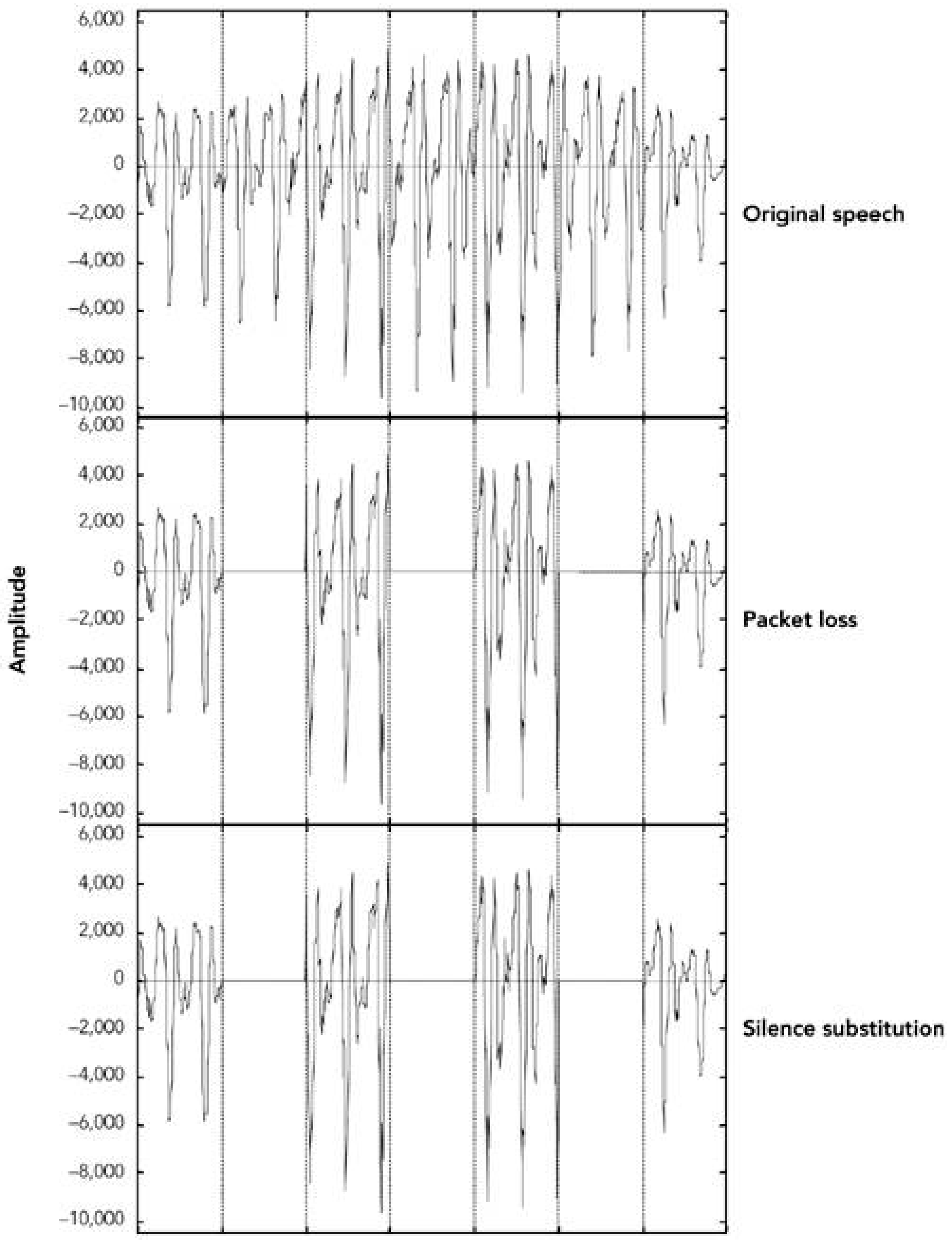
MOS scores provide a reasonable measure of perceived quality, allowing comparison between different codecs and repair techniques, but they do not measure intelligibility (that is, whether the audio is understandable). There is a difference between what sounds good and what conveys information; it is possible to define a concealment scheme that gets very good marks for sound quality but may not produce intelligible speech. In tests for intelligibility, listeners copy down sentences or words played with different impairments, or answer questions on a passage of text, and the result is a measure of how many errors are made. Again, a large number of tests must be conducted for the results to be statistically meaningful.

Perhaps the most important point learned from listening tests is that the results vary depending on the persons listening, the material they are listening to, the type of distortion present in the sound, and the task they are performing. Depending on the application, it may be important to conduct tests of both perceived quality and intelligibility, and it is always necessary to ensure that the test material and packet loss rates match those of typical usage.

Silence Substitution

The simplest possible repair technique is silence substitution, in which gaps caused by packet loss are filled with silence of the appropriate duration, as shown in Figure 8.1. This is the cheapest and easiest method to implement, and one of the most commonly used techniques.

Figure 8.1. Repair Using Silence Substitution (Adapted from C. Perkins, O. Hodson, and V. Hardman, "A Survey of Packet Loss Recovery Techniques for Streaming Media," IEEE Network Magazine, September/October 1998. © 1998 IEEE.)



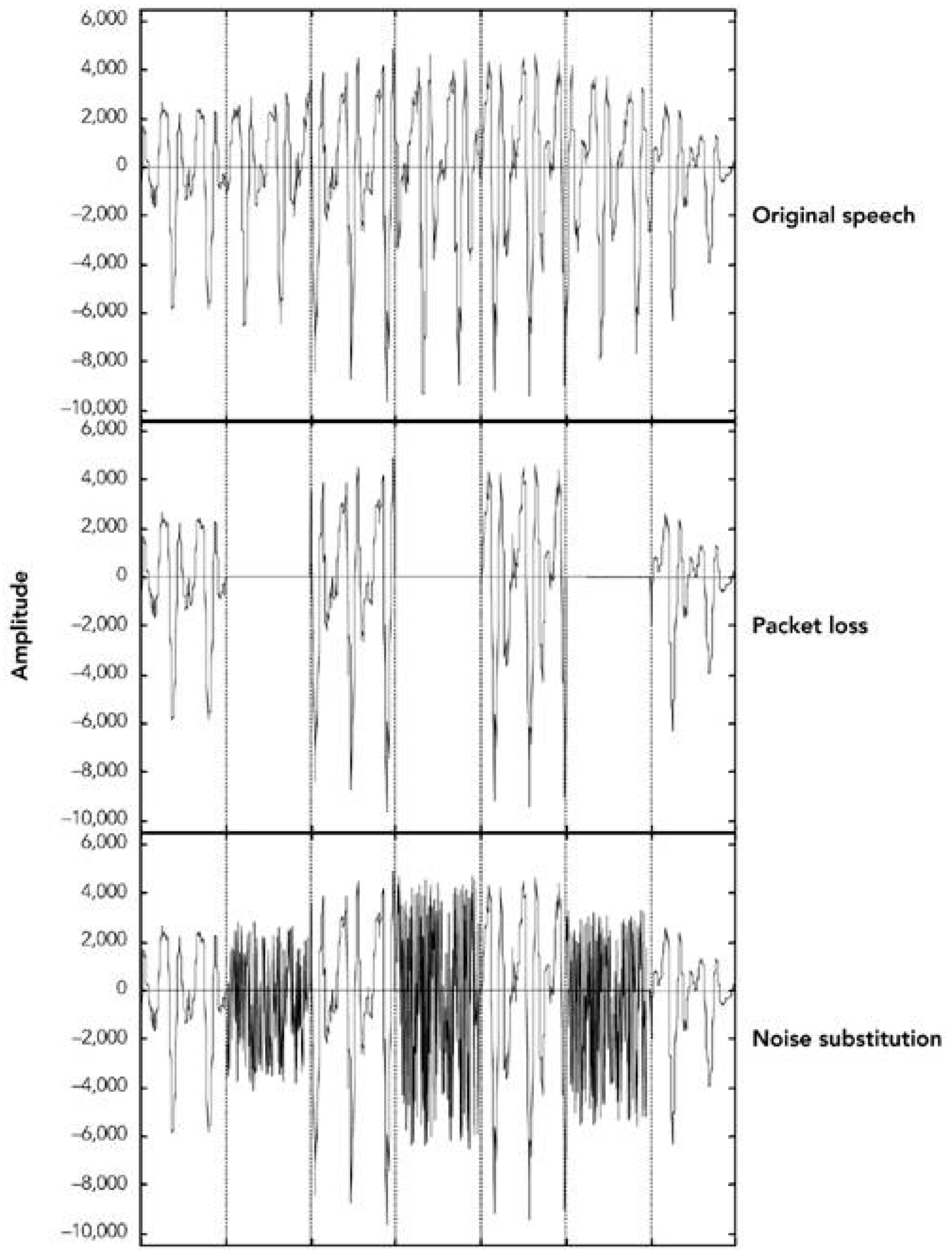
Unfortunately, silence substitution is also the worst repair scheme, consistently rated last in listening tests designed to evaluate repair quality.[114] Listening trials have shown that silence substitution is effective only with short-duration packets (<16 milliseconds) and low packet loss rates (<2%). Performance degrades rapidly as the packet size and loss rate increase, becoming rapidly unusable with the packet sizes used in voice-over-IP applications, and with the loss rates encountered in many networks.[75, 82]

Implementations should not use silence substitution. Any of the techniques described next will give better-quality sound, with a very small increase in complexity.

Noise Substitution

Because silence substitution has been shown to perform poorly, the next choice is to fill the gap left by a lost packet with background noise of some sort—a process known as noise substitution (see Figure 8.2).

Figure 8.2. Repair Using Noise Substitution (Adapted from C. Perkins, O. Hodson, and V. Hardman, "A Survey of Packet Loss Recovery Techniques for Streaming Media," IEEE Network Magazine, September/October 1998. © 1998 IEEE.)



At its simplest, noise substitution is the addition of white noise—noise with uniform amplitude at all frequencies—in place of the missing signal, amplitude matched to the previous packet. Here it is represented in pseudocode:

```
void
substitute_noise(sample previous_frame[samples_per_frame],
        sample missing_frame[samples_per_frame)
{
   double energy;

   // Calculate energy (amplitude) of the previous frame
   energy = 0.0;
   for(j = 0; j < samples_per_frame; j++) {
      energy += previous_frame[j] * previous_frame[j];
   }
   energy = sqrt(energy);
   // Fill in the noise
   for(j = 0; j < samples_per_frame; j++) {
      missing_frame[j] = energy * random(-1,1);
   }
}
```

Note that a real implementation will likely replace the arrays with a rotating buffer of the last few frames received, discarding old frames after they have been played out.

Listening tests with speech have shown that, when compared to silence substitution, the use of white-noise substitution with approximately the same amplitude as the signal gives both subjectively better quality[88] and improved intelligibility.[114] There is a perceptual basis to this improvement in quality: Studies have shown that *phonemic restoration*, the ability of the human brain to subconsciously repair missing segments of speech with the correct sound, occurs for speech repaired with noise, but not when silence is used as the repair. Because white noise is almost as easy to generate as silence, it is to be recommended as a replacement for silence substitution.

If the spectral characteristics of the signal are known, it may be possible to tailor the generated noise to match the original more closely than can be done with white noise. Many payload formats facilitate this task by providing comfort noise indicator packets to be sent during silence periods. Support for comfort noise allows the receiver to play an appropriate form of background noise when there is otherwise nothing to hear, and it can make the system seem more natural.

For payload formats that don't include native support for comfort noise, there is a standard comfort noise payload format that may be used to transport this information (static payload type 13 in the default audio/video profile[42]). This comfort noise format conveys two pieces of information: the amplitude of the noise, and the spectral parameters. The amplitude allows the receiver to generate amplitude-matched noise; the spectral parameters enable shaping of the noise to match the surrounding signal.

Repetition

Depending on the content of the audio signal, it may be possible to provide a replacement for a lost packet that is somewhat similar to the original. This is especially true of speech signals, which are interspersed with repetitive patterns, known as *pitch cycles*, that typically last from 20 milliseconds to 100 milliseconds. Losses that occur during pitch cycles have a good probability of being concealed.

Figure 8.3 shows a typical speech signal. Although many features can be recognized, the main differentiation is between voiced and unvoiced speech. Voiced speech, generated by the periodic opening and closing of the vocal folds (commonly called the vocal cords), generates regular, high-amplitude pitch cycles with frequency in the approximate range of 50Hz to 400Hz. Voiced segments often last for many tens or even hundreds of milliseconds, allowing the loss of a single packet of voice speech to be effectively repaired by substitution of the contents of the preceding packet. This phenomenon is reflected in Figure 8.4, which shows that loss repaired by repetition can be very similar to the original, except for the sharp discontinuity at the edges of the repaired region.

Figure 8.3. Speech Waveform

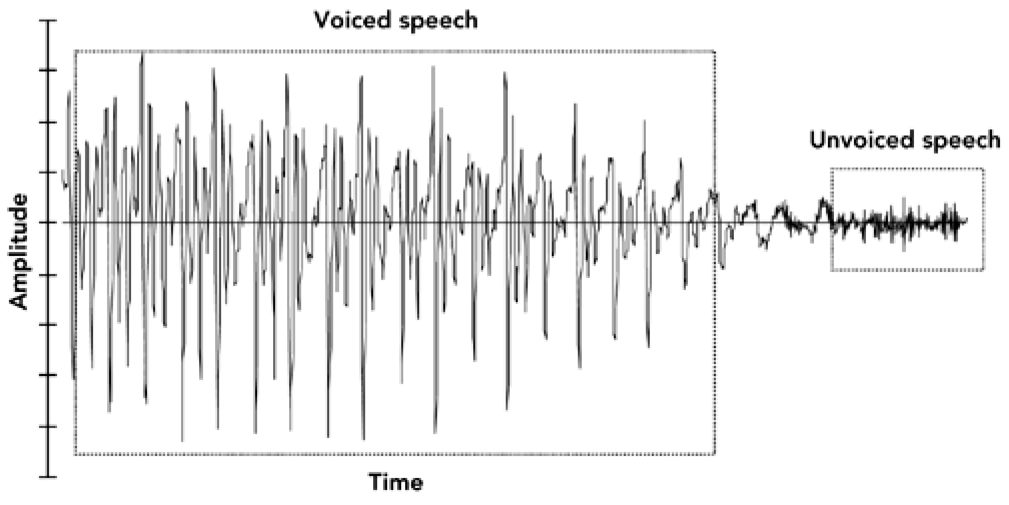
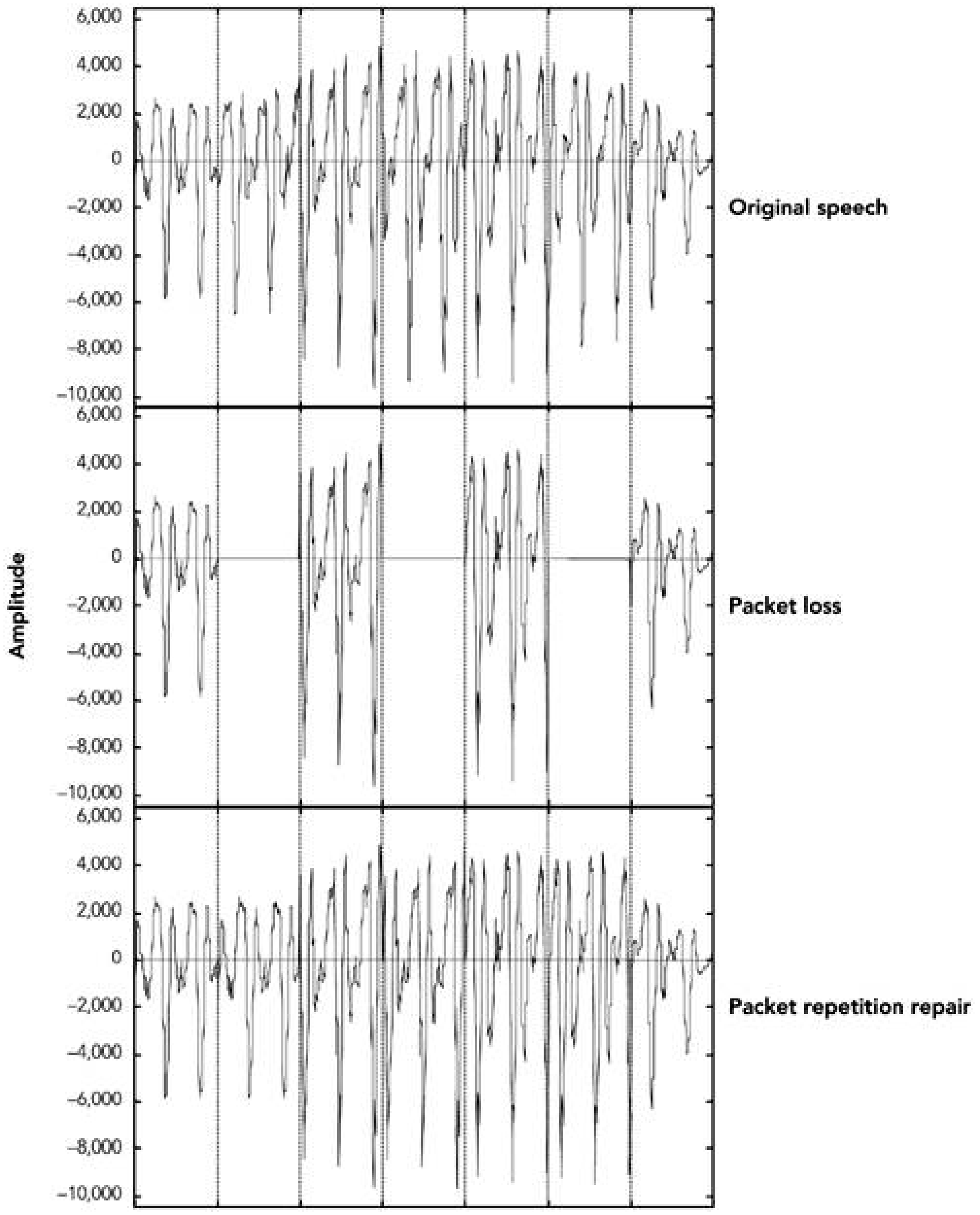


Figure 8.4. Repair of Voiced Speech Using Packet Repetition (Adapted from C. Perkins, O. Hodson, and V. Hardman, "A Survey of Packet Loss Recovery Techniques for Streaming Media," IEEE Network Magazine, September/October 1998. © 1998 IEEE.)



Unvoiced speech—consisting of sounds such as *s, f*, and *sh*—is generated by air being forced through a constriction in the vocal folds, and it closely resembles low-amplitude noise. Again, replacing a lost period of unvoiced speech with the contents of the previous packet produces reasonably good repair.

Repetition clearly works best when the gap is small, because the characteristics of the signal are likely to be similar across the gap. One can improve the performance of repetition with longer gaps by gradually fading the repeated signal. For example, the GSM mobile telephony system recommends an identical repeat for the first lost packet, followed by a gradual fade to zero amplitude over the next 16 packets (320 milliseconds total duration), or until the next packet is received.[60]

The repetition algorithm can be outlined in pseudocode like this:

```
void
repeat_and_fade_frame(sample previous_frame[samples_per_frame],
            sample missing_frame[samples_per_frame],
            int consecutive_lost)
{
   // Repeat previous frame
   for (j = 0; j < samples_per_frame; j++) {
      missing_frame[j] = previous_frame[j];
   }
   // Fade, if we've lost multiple consecutive frames
   if (consecutive_frames_lost > 0) {
      fade_per_sample = 1 / (samples_per_frame *
      fade_duration_in_frames);
      scale_factor = 1.0 – (consecutive_frames_lost *
                         samples_per_frame * fade_per_sample);
      if (scale_factor <= 0.0) {
         // In case consecutive_frames_lost >
         // fade_duration_in_frames
         scale_factor = fade_per_sample = 0.0;
      }
      for (j = 0; j < samples_per_frame; j++) {
         missing_frame[j] *= scale_factor
         scale_factor -= fade_per_sample
      }
   }
}
```

Note that the `previous_frame[]` array represents the previous frame received, not any previously repaired frame. The playout buffer should maintain the variable `consecutive_lost` based on the RTP sequence numbers, and should keep track of original versus repaired frames (in case one of the original frames was merely delayed).

Listening tests show that repetition works better than noise substitution for speech signals, and it is simple to implement. Repetition works better with speech than with music because the characteristics of music are more varied. Noise matched to the frequency spectrum of the signal may be a better choice for music signals.

Other Techniques for Repairing Speech Signals

The three simple repair techniques—silence substitution, noise substitution, and repetition—form the basis of many error concealment systems, and when correctly applied they can give good performance with low implementation complexity. Researchers have also studied a range of more specialized error concealment techniques for speech. These techniques typically trade increased complexity of implementation for a modest improvement in performance, and they are often tailored to particular types of input.

Various techniques based on waveform substitution have been proposed for use with speech. These techniques generate a suitable replacement packet based on characteristics of the speech signal surrounding a lost packet, and they can be viewed as extensions of packet repetition. Unlike basic packet repetition, waveform substitution algorithms adapt the repair to avoid discontinuity at the edges of the gap, and to match the characteristics of the signal better.

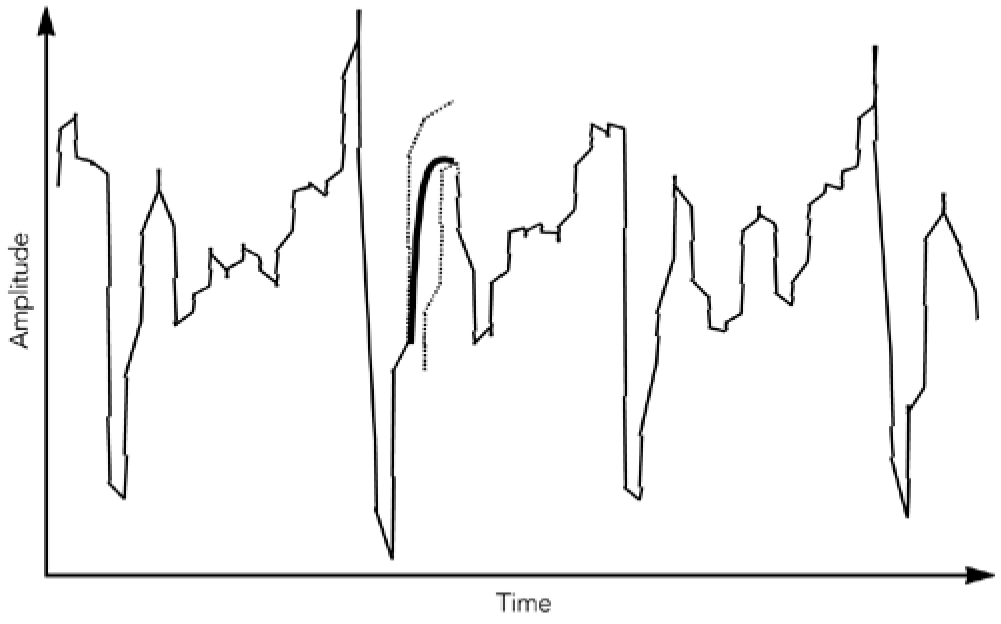
As an example of waveform substitution, consider the algorithm proposed by Wasem et al.,[107] building on earlier work by Goodman et al.[74] This algorithm first classifies speech as voiced or unvoiced (for example, by detecting the periodic spikes due to voiced pitch cycles). If the speech surrounding the loss is unvoiced, packet repetition is used to fill the gap. If the surrounding speech is voiced, a pattern-matching repair algorithm is used to find the region to repeat.

The pattern-matching repair algorithm uses the last few milliseconds of speech before the gap as a template. A sliding window algorithm is then used to compare the template with the rest of the packet, noting the location of the best match. The region between the template and its best match forms a complete pitch cycle, which is repeated to fill in the gap. Because the template closely matches the original, there is no significant discontinuity at the start of the repair. In pseudocode, the algorithm can be written this way:

```
void pattern_match_repair(sample previous_frame[samples_per_frame],
                          sample missing_frame[samples_per_frame],
                          int    consecutive_frames_lost)
{
    // Find best match for the window of the last few samples
    // in the packet
    window_start = samples_per_frame - window_length;
    target = infinity;
    for(i = 0; i < window_start; i ++) {
        score = 0;
        for(j = i, k = 0; k < window_length; j++, k++) {
            score += previous_frame[j] -
                              previous_frame[window_start + k];
        }
        if (score < target) {
           target = score;
           best_match = i; // The start of the best match for the
                          // window
        }
    }
    pattern = best_match + window_length;
    pattern_length = samples_per_frame – pattern;
    // "pattern" now points to the start of the region to repeat.
    // Copy the region into the missing packet
    dest = 0;
    for (remain = samples_per_frame; remain > 0;
                                     remain -= pattern_length) {
        for (j = 0; j < min(remain, pattern_length); j++) {
           missing_frame[dest++] = previous_frame[pattern + j];
        }
    }
    // Fade, if we've lost multiple consecutive frames
    if (consecutive_frames_lost > 0) {
        fade_buffer(missing_frame, consecutive_frames_lost);
    }
}
```

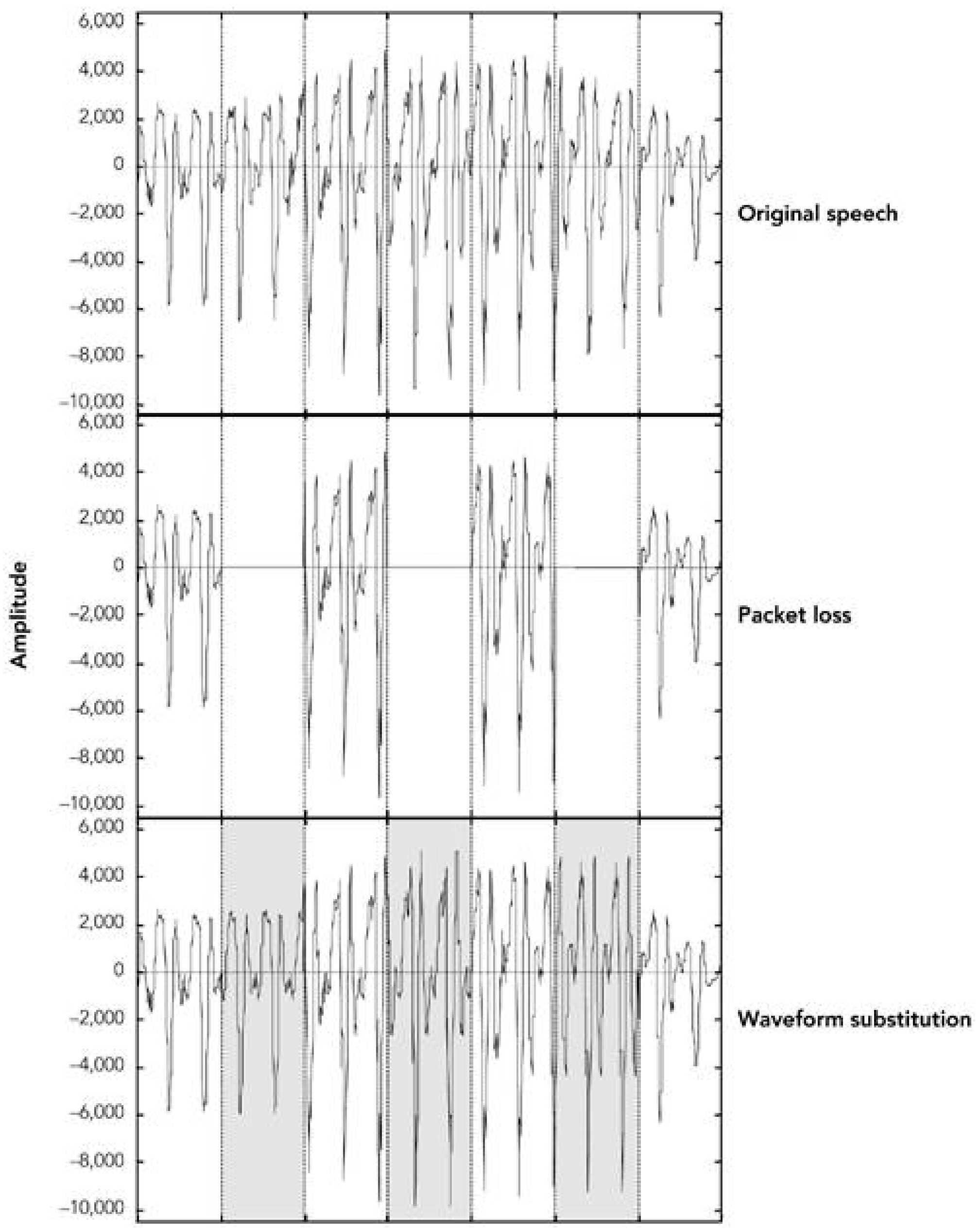
There is still a boundary discontinuity at the end of the repair. We can patch this by merging the repair with the original data, provided that the two overlap. Such a patch is illustrated in Figure 8.5 , where the weighted average of the two waveforms is used in the overlap region, providing a smooth transition. Weighting means to take more of the first waveform at the beginning of the overlap region and more of the second waveform at the end.

Figure 8.5. Packet Merging at the Boundary of a Repair



The result is a very effective repair algorithm for speech, which noticeably outperforms repetition. A sample speech waveform repaired by waveform substitution is shown in Figure 8.6.

Figure 8.6. Repair Using Waveform Substitution (Adapted from C. Perkins, O. Hodson, and V. Hardman, "A Survey of Packet Loss Recovery Techniques for Streaming Media," IEEE Network Magazine, September/October 1998. © 1998 IEEE.)

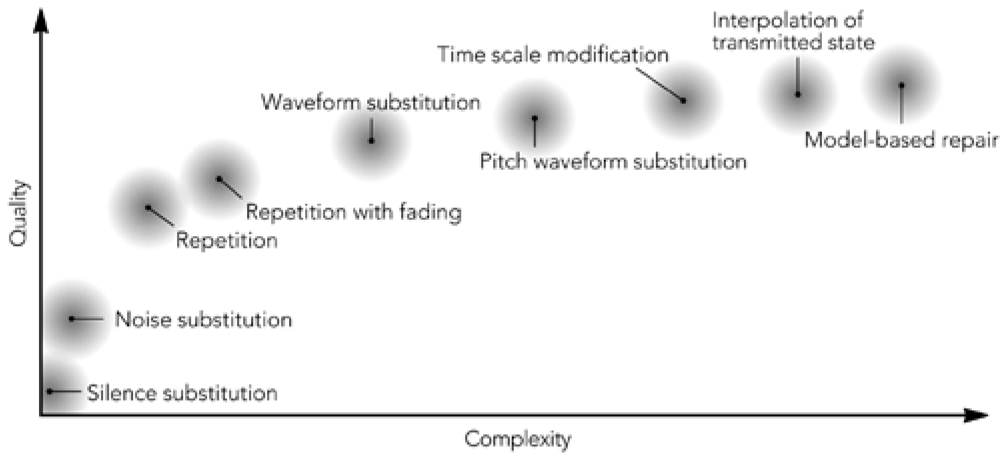


Researchers have proposed a seemingly endless series of error concealment techniques that offer incremental improvements over those discussed here. These techniques include the following:

- **Timescale modification**, which stretches the audio on either side of a loss across the gap. For example, Sanneck et al.[102] have proposed a scheme in which pitch cycles are stretched to cover the loss from either side, and averaged where they meet.
- **Regenerative repair**, which uses knowledge of the audio compression algorithm to derive the appropriate codec parameters to recover a lost packet.
- **Interpolation of codec state**, which allows codecs based on linear prediction (for example, G.723.1) to derive the predictor coefficients by interpolating the frames on either side of a loss.
- **Model-based repair**, which attempts to fit the signals on either side of the loss to a model of the vocal tract/codec, and uses this model to predict the correct fill-in.

Applications that deal solely with speech may want to consider these more complex repair schemes. Be aware, however, that the gains are incremental and the increase in complexity is considerable (see Figure 8.7 for a rough chart of quality versus complexity).

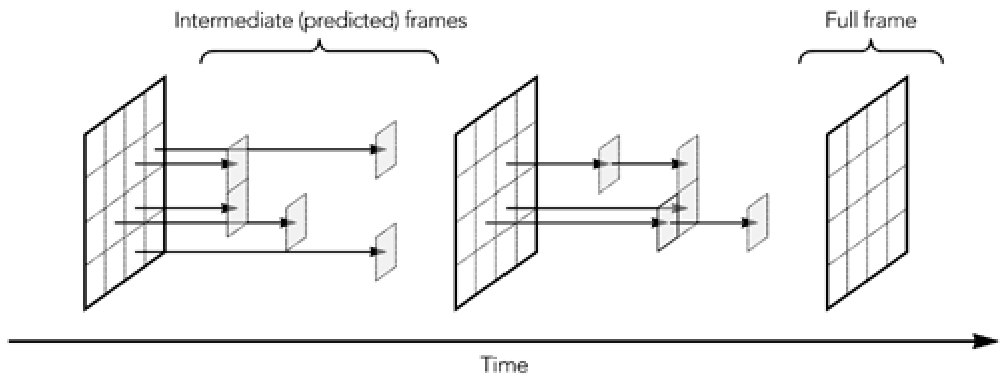
Figure 8.7. Rough Quality/Complexity Trade-off for Error Concealment in Speech Signals (From C. Perkins, O. Hodson, and V. Hardman, "A Survey of Packet Loss Recovery Techniques for Streaming Media," IEEE Network Magazine, September/October 1998. © 1998 IEEE.)



Techniques for Video Loss Concealment

Most video codecs use interframe compression, sending occasional full frames and many intermediate frames as updates to the parts of the frame that have changed or moved, as shown in Figure 8.8. This technique, known as *predictive coding* because each frame is predicted on the basis of the preceding frame, is essential for good compression.

Figure 8.8. Basic Operation of Video Codecs



Predictive coding has several consequences for loss concealment. The first is that loss of an intermediate frame may affect only part of a frame, rather than the whole frame (similar effects occur when a frame is split across multiple packets, some of which are lost). For this reason, concealment algorithms must be able to repair damaged regions of an image, as well as replace an entire lost image. A common way of doing this is through motion-compensated repetition.

The other consequence of predictive coding is that frames are no longer independent. This means that loss of data in one frame may affect future frames, making loss concealment more difficult. This problem is discussed in the section titled Dependency Reduction later in this chapter, along with possible solutions.

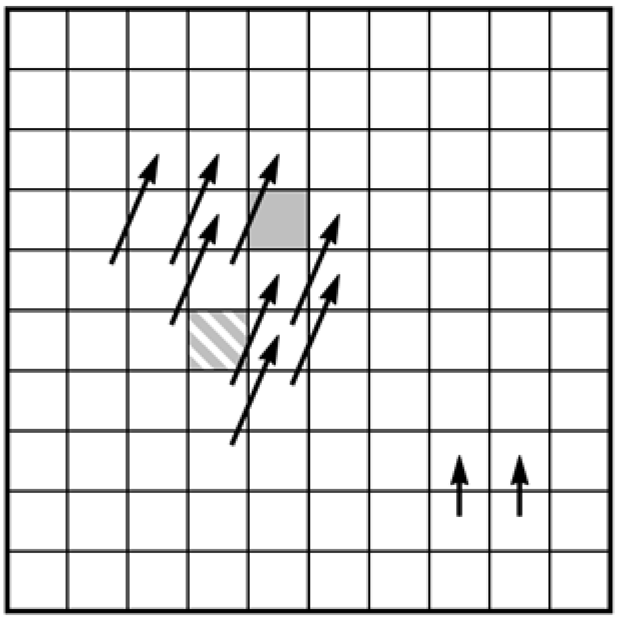
Motion-Compensated Repetition

One of the widely used techniques for video loss concealment is repetition in the time domain. When loss occurs, the part of the frame affected by the loss is replaced with a repeat of the preceding frame. Because most video codecs are block based, and the missing data will often constitute only a small part of the image, this type of repair is usually acceptable.

Of course, repetition works only if the image is relatively constant. If there is significant motion between frames, repeating a portion of the preceding frame will give noticeable visual artifacts. If possible, it is desirable to detect the motion and try to compensate for it when concealing the effects of loss. In many cases this is easier than might be imagined because common video codecs allow the sender to use motion vectors to describe changes in the image, rather than sending a new copy of moved blocks.

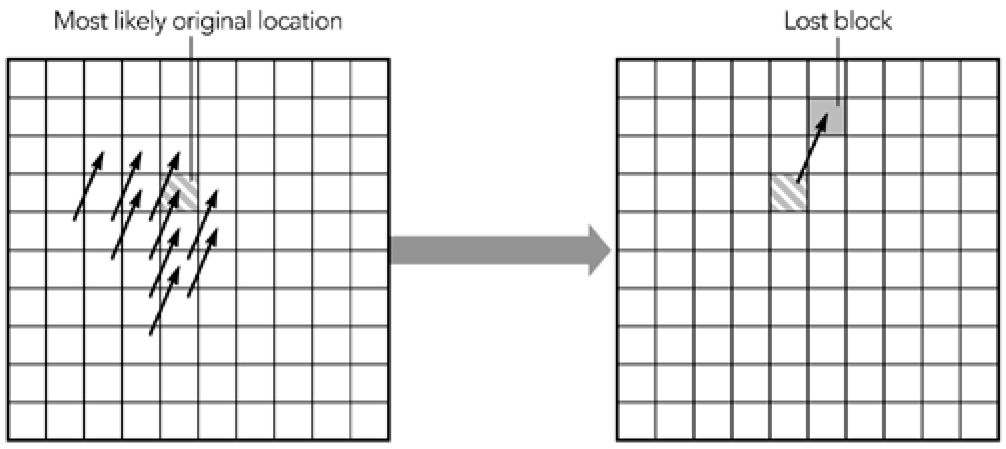
If only a single block of the image is lost, a receiver may use the motion vectors associated with the surrounding blocks to infer the correct position for the missing block, on the basis of the preceding packet. For example, Figure 8.9 shows how the motion of a single missing block of the image can be inferred. If the highlighted block is lost, the original position can be derived because the motion is likely the same as that for the surrounding blocks.

Figure 8.9. Motion-Compensated Repetition of a Missing Video Block



Motion-compensated repetition relies on loss affecting only a part of the image. This makes it well suited to network transport that corrupts single bits in the image, but less suited to transport over IP networks that lose packets containing several, most likely adjacent, blocks. Interleaving—discussed later in this chapter—is one solution to this problem; another is to use the motion vectors from the preceding frame to infer the motion in the current frame, as shown in Figure 8.10. The assumption here is that motion is smooth and continuous across frames—an assumption that is not unreasonable in many environments.

Figure 8.10. Inferred Motion across Frames



The two schemes can work together, inferring lost data either from other blocks in the current frame or from the previous frame.

It is recommended that implementations should, at least, repeat the contents of the preceding frame in the event of packet loss. It is also worth studying the codec operation to determine whether motion compensation is possible, although this is a lesser benefit and may be predetermined by the design of the codec.

Other Techniques for Repairing Video Packet Loss

Besides repetition, two other classes of repair may be used: repair in the spatial domain and repair in the frequency domain.

Repair in the spatial domain relies on interpolation of a missing block, on the basis of the surrounding data. Studies have shown that human perception of video is relatively insensitive to high-frequency components—detail—of the image. Thus a receiver can generate a fill-in that is approximately correct, and as long as this is just a transient, it will not be too visually disturbing. For example, the average pixel color for each of the surrounding blocks can be calculated, and the missing block can be set to the average of those colors.

Similar techniques can be applied in the frequency domain, especially for codecs based on the discrete cosine transform (DCT), such as MPEG, H.261, and H.263. In this case the low-order DCT coefficients can be averaged across the surrounding blocks, to generate a fill-in for a missing block.

Simple spatial and temporal repair techniques give poor results if the error rate is high, and generally they do not work well with packet loss. They work better with networks that give bit errors, and they corrupt a single block rather than losing an entire packet containing several blocks. There are various more advanced spatial and temporal repair techniques—the surveys by Wang et al.[105], [106] provide a good overview—but again, these are generally unsuited to packet networks.

Dependency Reduction

Although predictive coding is essential to achieving good compression, it makes the video sensitive to packet loss and complicates error concealment. On the other hand, if each frame of video is independently coded, a lost packet will affect only a single frame. The result will be a temporary glitch, but it will rapidly be corrected when the next frame arrives. The penalty of independently coded frames is a much higher data rate.

When predictive coding is used and the frames are not independent, loss of a single packet will propagate across multiple frames, causing significant degradation to the video stream. For example, part of a frame is lost and has to be inferred from the preceding frame, producing a repair that is, by necessity, inexact. When the next frame arrives, it contains a motion vector, which refers to the part of the image that was repaired. The result is that the incorrect data remains in the picture across multiple frames, moving around according to the motion vectors.

Error propagation in this manner is a significant problem because it multiplies the effects of any loss and produces results that are visually disturbing. Unfortunately, there is little a receiver can do to correct the problem, because it has insufficient data to repair the loss until a complete frame update arrives. If the loss exceeds a particular threshold, a receiver might find it better to discard frames predicted from lost data, displaying a frozen image, than to use the erroneous state as a basis and display damaged pictures.

A sender can ease this problem by using less predictive coding when packet loss is present, although doing so may reduce the compression efficiency and lead to an increase in the data rate (see Chapter 10, Congestion Control, for a related discussion). If possible, senders should monitor RTCP reception report feedback and reduce the amount of prediction as the loss rate increases. This does not solve the problem, but it means that full frame updates occur more often, allowing receivers to resynchronize with the media stream. To avoid exceeding the available bandwidth, it may be necessary to reduce the frame rate.

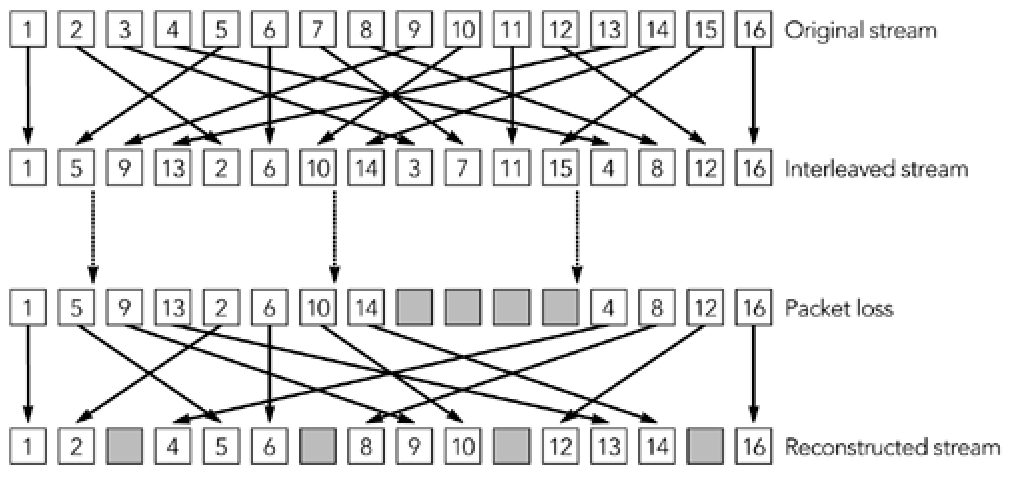
There is a fundamental trade-off between compression efficiency and loss tolerance. Senders must be aware that compression to very low data rates, using predictive coding, is not robust to packet loss.

Interleaving

At the start of this chapter, it was noted that error concealment is something done by a receiver, without help from the sender. In general this is the case, but sometimes a sender can ease the task of error concealment without having to send extra information. One such example was noted for video, in which a sender can reduce the interframe dependency to ease the job of a receiver. A more general-purpose technique is interleaving, which can be used with both audio and video streams, as long as low delay is not a requirement.

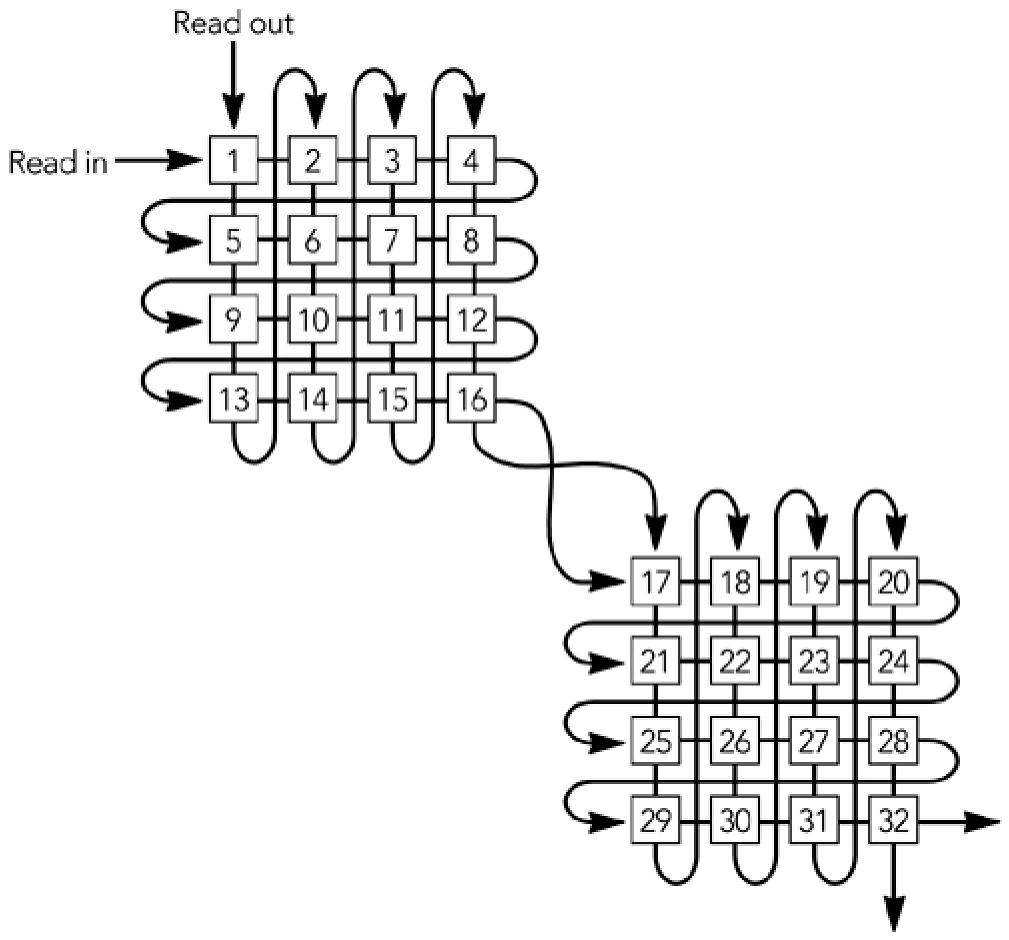
The interleaving process reorders data before transmission so that originally adjacent data is separated by a guaranteed distance during transport. Interleaving is useful because it makes bursts of consecutive packet loss in the transport stream appear as isolated losses when the original order is restored. In Figure 8.11, for example, the loss of four consecutive packets in the interleaved stream is transformed into four single-packet losses when the original order is reconstructed. The actual loss rate is unchanged, but it is typically easier for a receiver to conceal a series of single-packet losses than it is to conceal a longer burst of loss.

Figure 8.11. Interleaving, Transforming Burst Loss to Isolated Loss (From C. Perkins, O. Hodson, and V. Hardman, "A Survey of Packet Loss Recovery Techniques for Streaming Media," IEEE Network Magazine, September/October 1998. © 1998 IEEE.)



The simplest implementation of an interleaving function is with a pair of matrices, as shown in Figure 8.12. Frames of media data are read into the first matrix by rows until that matrix is full. At that time the two matrices are switched, with data being read out of the first by columns, as the second is filled by rows. The process continues, with frames being read into one matrix as they are read out of the other.

Figure 8.12. An Interleaving Matrix



Both sender and receiver must maintain buffer matrices as arrays of an appropriate size. The sender takes the output by columns and inserts it into packets for transport. The receiver takes the packets from the transport stream and passes them into the matrix buffer by rows, and as they are read out by columns the original order is restored.

Interleaving may be done at the RTP packet level, but more often multiple frames within each packet are interleaved. When interleaving is done at the RTP packet level, the codec data is packed into RTP packets as usual. The interleaving operates on complete packets, resulting in a series of packets that have nonconsecutive RTP timestamps. The RTP sequence number should be generated after interleaving, resulting in packets that have consecutive sequence numbers as sent.

When interleaving is done with multiple frames within each packet, the RTP timestamp and sequence number are unchanged from the noninterleaved format. Putting multiple codec frames into each RTP packet so that each column of the interleaver forms an RTP packet keeps these values constant.

The interleaving function should be chosen so that adjacent packets in the original stream are separated by more than the maximum consecutive loss length in the transport stream. A matrix-based implementation with *n* rows of *m* columns will produce output with originally adjacent symbols separated by *n* others. Put another way, as *m* or fewer packets are lost during transport of each matrix, each group of *n* packets after deinterleaving will have at most one loss.

The transport process must communicate the size of the interleaving group—the dimensions of the matrix—and the position of each packet within that group. The size of the interleaving group may be fixed and communicated out of band, or it may be included within each packet, allowing the interleaving function to vary. The position of each packet within an interleaving group must be included with the packet if the group size can vary, or it may be inferred from the RTP sequence number if the group size is fixed.

A good example of the use of interleaving is the loss-tolerant payload format for MPEG Audio Layer-3 (MP3),[38] which was developed in response to the lack of resilience of the original payload format[12] to packet loss. Other examples of interleaving may be found in the payload formats for AMR (Adaptive Multi-Rate) and AMR-WB (Adaptive Multi-Rate Wideband) audio.[41] There is no single standard for interleaving; each payload format must implement it individually.

Interleaving adds considerable latency to the transmission process.[97] The same number of packets as in the interleave group at both sender and receiver will be buffered. For example, if a 5 x 3 matrix implementation is used, 15 packets will be buffered at the sender, and 15 at the receiver, in addition to the network delay. This makes interleaving unsuitable for interactive applications; however, interleaving works well for streaming.

Summary

One of the key points to remember when designing an RTP application is robustness. Error concealment is a major part of this, allowing the application to operate even when the network misbehaves. A good error concealment scheme provides the difference between a tool that can be used in the real world and one that continually fails when subjected to the wide variation in loss rates inherent in the Internet. For this reason, all applications should implement some form of error concealment.

Chapter 9. Error Correction

Although it is clearly important to be able to conceal the effects of transmission errors, it is better if those errors can be avoided or corrected. This chapter presents techniques that the sender can use to help receivers recover from packet loss and other transmission errors.

The techniques used to correct transmission errors fall into two basic categories: forward error correction and retransmission.[80] Forward error correction relies on additional data added by the sender to a media stream, which receivers can then use to correct errors with a certain probability. Retransmission, on the other hand, relies on explicit requests for additional copies of particular packets.

The choice between retransmission and forward error correction depends on the application and on the network characteristics. The details and trade-offs of the different approaches are discussed in more detail in this chapter.

Forward Error Correction

Forward error correction (FEC) algorithms transform a bit stream to make it robust for transmission. The transformation generates a larger bit stream intended for transmission across a lossy medium or network. The additional information in the transformed bit stream allows receivers to exactly reconstruct the original bit stream in the presence of transmission errors. Forward error correction algorithms are notably employed in digital broadcasting systems, such as mobile telephony and space communication systems, and in storage systems, such as compact discs, computer hard disks, and memory. Because the Internet is a lossy medium, and because media applications are sensitive to loss, FEC schemes have been proposed and standardized for RTP applications. These schemes offer both exact and approximate reconstruction of the bit stream, depending on the amount and type of FEC used, and on the nature of the loss.

When an RTP sender uses FEC, it must decide on the amount of FEC to add, on the basis of the loss characteristics of the network. One way of doing this is to look at the RTCP receiver report packets it is getting back, and use the loss fraction statistics to decide on the amount of redundant data to include with the media stream.

In theory, by varying the encoding of the media, it is possible to guarantee that a certain fraction of losses can be corrected. In practice, several factors indicate that FEC can provide only probabilistic repair. Key among those is the fact that adding FEC increases the bandwidth of a stream. This increase in bandwidth limits the amount of FEC that can be added on the basis of the available network capacity, and it may also have adverse effects if loss is caused by congestion. In particular, adding bandwidth to the stream may increase congestion, worsening the loss that the FEC was supposed to correct. This issue is discussed further in the section titled At the

Sender, under Implementation Considerations, later in this chapter, as well as in Chapter 10, Congestion Control.

Note that although the amount of FEC can be varied in response to reception quality reports, there is typically no feedback about individual packet loss events, and no guarantee that all losses are corrected. The aim is to reduce the residual loss rate to something acceptable, then to let error concealment take care of any remaining loss.

If FEC is to work properly, the loss rate must be bounded, and losses must occur in particular patterns. For example, it is clear that an FEC scheme designed to correct 5% loss will not correct all losses if 10% of packets are missing. Less obviously, it might be able to correct 5% loss only if the losses are of nonconsecutive packets.

The key advantage of FEC is that it can scale to very large groups, or groups where no feedback is possible.[54] The amount of redundant data added depends on the average loss rate and on the loss pattern, both of which are independent of the number of receivers. The disadvantage is that the amount of FEC added depends on the average loss rate. A receiver with below-average loss will receive redundant data, which wastes capacity and must be discarded. One with above-average loss will be unable to correct all the errors and will have to rely on concealment. If the loss rates for different receivers are very heterogeneous, it will not be possible to satisfy them all with a single FEC stream (layered coding may help; see Chapter 10, Congestion Control).

Another disadvantage is that FEC may add delay because repair cannot happen until the FEC packets arrive. If FEC packets are sent a long time after the data they protect, then a receiver may have to choose between playing damaged data quickly or waiting for the FEC to arrive and potentially increasing the end-to-end delay. This is primarily an issue with interactive applications, in which it is important to have low delay.

Many FEC schemes exist, and several have been adopted as part of the RTP framework. We will first review some techniques that operate independently of the media format—parity FEC and Reed–Solomon encoding—before studying those specific to particular audio and video formats.

Parity FEC

One of the simplest error detection/correction codes is the parity code. The parity operation can be described mathematically as an exclusive-or (XOR) of the bit stream. The XOR operation is a bitwise logic operation, defined for two inputs in this way:

```
0 XOR 0 = 0
1 XOR 0 = 1
0 XOR 1 = 1
1 XOR 1 = 0
```

The operation may easily be extended to more than two inputs because XOR is associative:

```
A XOR B XOR C = (A XOR B) XOR C = A XOR (B XOR C)
```

Changing a single input to the XOR operation will cause the output to change, allowing a single parity bit to detect any single error. This capability is of limited value by itself, but when multiple parity bits are included, it becomes possible to detect *and correct* multiple errors.

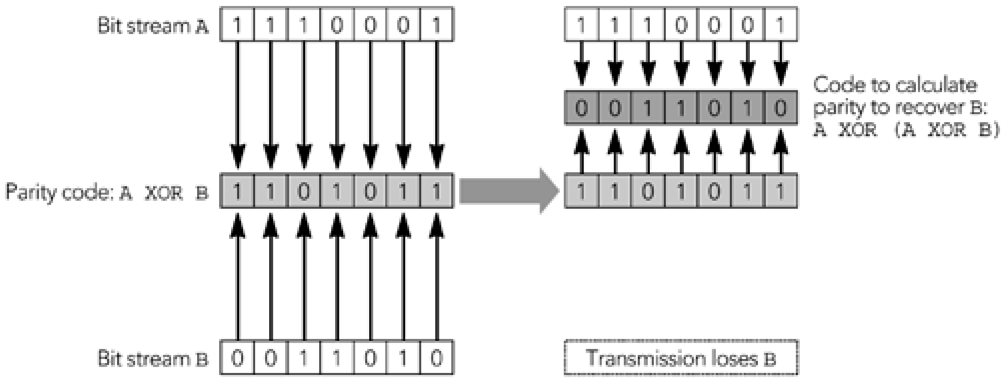
To make parity useful to a system using RTP-over-UDP/IP—in which the dominant error is packet loss, not bit corruption—it is necessary to send the parity bits in a separate packet to the data they are protecting. If there are enough parity bits, they can be used to recover the complete contents of a lost packet. The property that makes this possible is that

```
A XOR B XOR B = A
```

for any values of `A` and `B`.

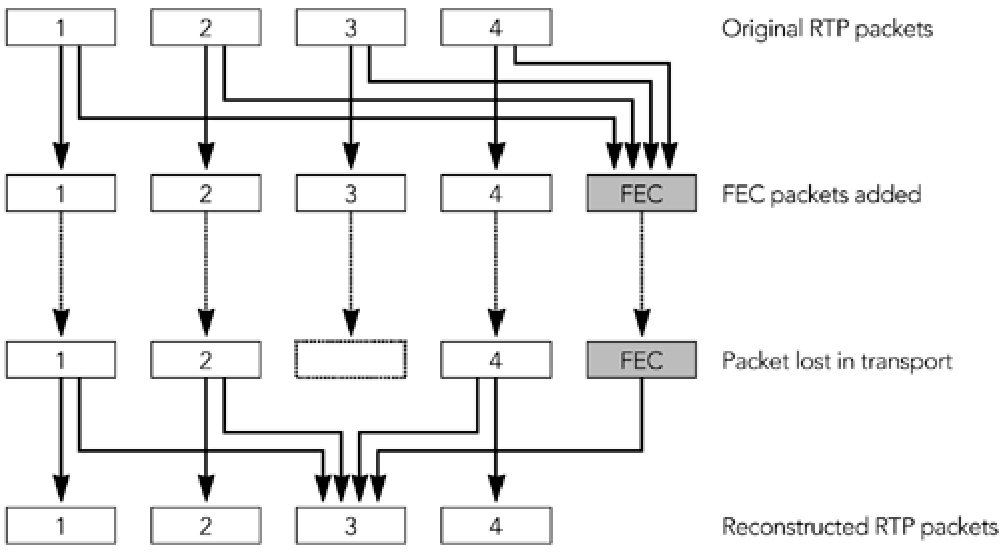
If we somehow transmit the three pieces of information `A`, `B`, and `A XOR B` separately, we need only receive two of the three pieces to recover the values of `A` and `B`. Figure 9.1 shows an example in which a group of seven lost bits is recovered via this process, but it works for bit streams of any length. The process may be directly applied to RTP packets, treating an entire packet as a bit stream and calculating parity packets that are the XOR of original data packets, and that can be used to recover from loss.

Figure 9.1. Use of Parity between Bit Streams to Recover Lost Data



The standard for parity FEC applied to RTP streams is defined by RFC 2733.[32] The aim of this standard is to define a generic FEC scheme for RTP packets that can operate with any payload type and that is backward-compatible with receivers that do not understand FEC. It does this by calculating FEC packets from the original RTP data packets; these FEC packets are then sent as a separate RTP stream, which may be used to repair loss in the original data, as shown in Figure 9.2.

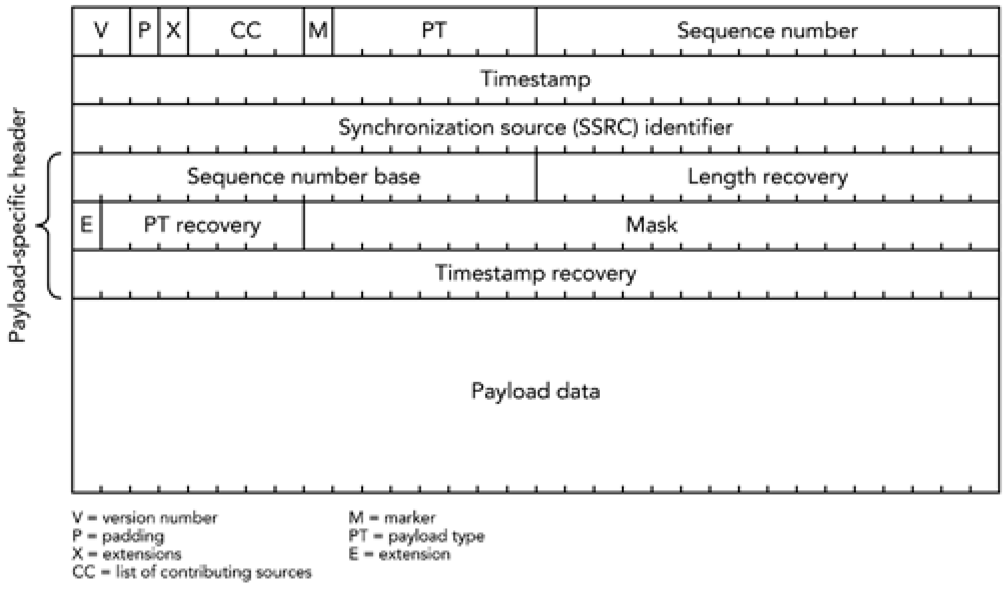
Figure 9.2. Repair Using Parity FEC (From C. Perkins, O. Hodson, and V. Hardman, "A Survey of Packet Loss Recovery Techniques for Streaming Media," IEEE Network Magazine, September/October 1998. © 1998 IEEE.)



FORMAT OF PARITY FEC PACKETS

The format of an FEC packet, shown in Figure 9.3, has three parts to it: the standard RTP header, a payload-specific FEC header, and the payload data itself. With the exception of some fields of the RTP header, the FEC packet is generated from the data packets it is protecting. It is the result of applying the parity operation to the data packets.

Figure 9.3. Format of a Parity FEC Packet



The fields of the RTP header are used as detailed here:

- The version number, payload type, sequence number, and timestamp are assigned in the usual manner. The payload type is dynamically assigned, according to the RTP profile in use; the sequence number increases by one for each FEC packet sent; and the timestamp is set to the value of the RTP media clock at the instant the FEC packet is transmitted. (The timestamp is unlikely to be equal to the timestamp of the surrounding RTP packets.) As a result, the timestamps in FEC packets increase monotonically, independently of the FEC scheme.
- The SSRC (synchronization source) value is the same as the SSRC of the original data packets.
- The padding, extension, CC, and marker bits are calculated as the XOR of the equivalent bits in the original data packets. This allows those fields to be recovered if the original packets are lost.
- The CSRC (contributing source) list and header extension are never present, independent of the values of the CC field and X bit. If they are present in the

original data packets, they are included as part of the payload section of the FEC packet (after the FEC payload header).

Note that the prohibition of CSRC list and header extension in parity FEC packets means that it is not always possible to treat FEC streams according to the standard, payload format–independent, RTP processing rules. In particular, an FEC stream cannot pass through an RTP mixer (the media data can, but the mixer will have to generate a new FEC stream for the mixed data)

.

The payload header protects the fields of the original RTP headers that are not protected in the RTP header of the FEC packet. These are the six fields of the payload header:

1. **Sequence number base**. The minimum sequence number of the original packets composing this FEC packet.
2. **Length recovery**. The XOR of the lengths of the original data packets. The lengths are calculated as the total length of the payload data, CSRC list, header extension, and padding of the original packets. This calculation allows the FEC procedure to be applied even when the lengths of the media packets are not identical.
3. **Extension (E)**. An indicator of the presence of additional fields in the FEC payload header. It is usually set to zero, indicating that no extension is present (the ULP format, described later in this chapter, uses the extension field to indicate the presence of additional layered FEC).
4. **Payload type (PT) recovery**. The XOR of the payload type fields of the original data packets.
5. **Mask**. A bit mask indicating which of the packets following the sequence number base are included in the parity FEC operation. If bit *i* in the mask is set to 1, the original data packet with sequence number *N* + *i* is associated with this FEC packet, where *N* is the sequence number base. The least significant bit corresponds to *i* = 0, and the most significant to *i* = 23, allowing for the parity FEC to be calculated over up to 24 packets, which may be nonconsecutive.
6. **Timestamp recovery**. The XOR of the timestamps of the original data packets.

The payload data is derived as the XOR of the CSRC list (if present), header extension (if present), and payload data of the packets to be protected. If the data packets are different lengths, the XOR is calculated as if the short packets were padded out to match the length of the largest (the contents of the padding bits are unimportant, as long as the same values are used each time a particular packet is processed; it is probably easiest to use all zero bits).

USE OF PARITY FEC

The number of FEC packets and how they are generated depend on the FEC scheme employed by the sender. The payload format places relatively few restrictions on the mapping process: Packets from a group of up to 24 consecutive original packets are input to the parity operation, and each may be used in the generation of multiple FEC packets.

The sequence number base and mask in the payload header are used to indicate which packets were used to generate each FEC packet; there is no need for additional signaling. Accordingly, the packets used in the FEC operation can change during an RTP session, perhaps in response to the reception quality information contained in RTCP RR packets. The ability of the FEC operation to change gives the sender much flexibility: The sender can adapt the amount of FEC in use according to network conditions and be certain that the receivers will still be able to use the FEC for recovery.

A sender is expected to generate an appropriate number of FEC packets in real time, as the original data packets are sent. There is no single correct approach for choosing the amount of FEC to add because the choice depends on the loss characteristics of the network, and the standard does not mandate a particular scheme. Following are some possible choices:

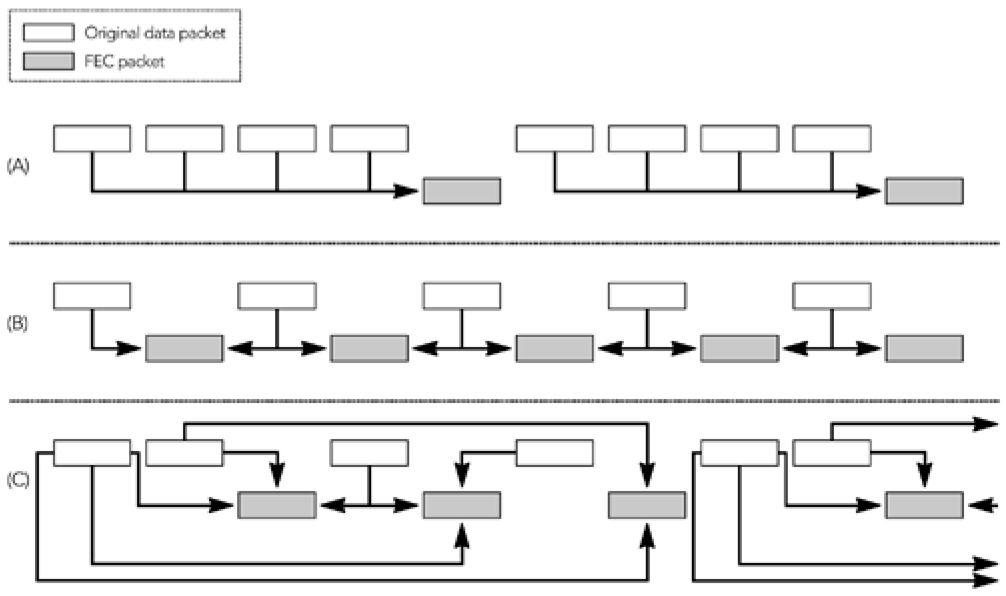
- The simplest approach is to send one FEC packet for every *n* – 1 data packets, as shown in Figure 9.4A, allowing recovery provided that there is at most one loss for every *n* packets. This FEC scheme has low overhead, is easy to compute, and is easy to adapt (because the fraction of packets that are FEC packets directly corresponds to the loss fraction reported in RTCP RR packets).

  If the probability that a packet is lost is uniform, this approach works well; however, bursts of consecutive loss cannot be recovered. If bursts of loss are common—as in the public Internet—the parity can be calculated across widely spaced packets, rather than over adjacent packets, resulting in more robust protection. The result is a scheme that works well for streaming but has a large delay, making it unsuitable for interactive applications.

- A more robust scheme, but one with significantly higher overhead, is to send an FEC packet between each pair of data packets, as shown in Figure 9.4B. This approach allows the receiver to correct every single packet loss, and many double losses. The bandwidth overhead of this approach is high, but the amount of delay added is relatively small, making it more suitable for interactive applications.

- Higher-order schemes allow recovery from more consecutive losses. For example, Figure 9.4C shows a scheme that can recover from loss of up to three consecutive packets. Because of the need to calculate FEC over multiple packets, the delay introduced is relatively high, so these schemes are unlikely to be suitable for interactive use. They can be useful in streaming applications, though.

Figure 9.4. Some Possible FEC Schemes



To make parity FEC backward-compatible, it is essential that older receivers do not see the FEC packets. Thus the packets are usually sent as a separate RTP stream, on a different UDP port but to the same destination address. For example, consider a session in which the original RTP data packets use static payload type 0 (G.711 µ-law) and are sent on port 49170, with RTCP on port 49171. The FEC packets could be sent on port 49172, with their corresponding RTCP on port 49173. The FEC packets use a dynamic payload type—for example, 122. This scenario could be described in SDP like this:

```
v=0
o=hamming 2890844526 2890842807 IN IP4 128.16.64.32
s=FEC Seminar
c=IN IP4 10.1.76.48/127
t=0 0
m=audio 49170 RTP/AVP 0 122
a=rtpmap:122 parityfec/8000
a=fmtp:122 49172 IN IP4 10.1.76.48/127
```

An alternative—described in the section titled Audio Redundancy Coding later in this chapter—is to transport parity FEC packets as if they were a redundant encoding of the media.

RECOVERING FROM LOSS

At the receiver the FEC packets and the original data packets are received. If no data packets are lost, the parity FEC can be ignored. In the event of loss, the FEC packets can be combined with the remaining data packets, allowing the receiver to recover lost packets.

There are two stages to the recovery process. First, it is necessary to determine which of the original data packets and the FEC packets must be combined in order to recover a missing packet. After this is done, the second step is to reconstruct the data.

Any suitable algorithm can be used to determine which packets must be combined. RFC 2733 gives an example, which operates as shown here:

- When an FEC packet is received, the sequence number base and mask fields are checked to determine which packets it protects. If all those packets have been received, the FEC packet is redundant and is discarded. If some of those packets are missing, *and they have sequence numbers smaller than the highest received sequence number*, recovery is attempted; if recovery is successful, the FEC packet is discarded and the recovered packet is stored into the playout buffer. Otherwise the FEC packet is stored for possible later use.
- When a data packet is received, any stored FEC packets are checked to see whether the new data packet makes recovery possible. If so, after recovery the FEC packet is discarded and the recovered packet entered into the playout buffer.
- Recovered packets are treated as if they were received packets, possibly triggering further recovery attempts.

Eventually, all FEC packets will be used or discarded as redundant, and all recoverable lost packets will be reconstructed.

The algorithm relies on an ability to determine whether a particular set of data packets and FEC packets makes it possible to recover from a loss. Making the determination requires looking at the set of packets referenced by an FEC packet; if only one is missing, it can be recovered. The recovery process is similar to that used to generate the FEC data. The parity (XOR) operation is conducted on the equivalent fields in the data packets and the FEC packets; the result is the original data packet.

In more detail, this is the recovery process:

1. The SSRC of the recovered packet is set to the SSRC of the other packets.
2. The padding, header extension, CC, and marker bits of the recovered packet are generated as the XOR of the same fields in the original and FEC packets.
3. The sequence number of the recovered packet is known from the gap in the original sequence numbers (that is, there is no need to recover it, because it is directly known).
4. The payload type of the recovered packet is generated as the XOR of the payload type fields in the original packets, and the payload type recovery field of the FEC packets. The timestamp is recovered in the same manner.
5. The length of the payload is calculated as the XOR of lengths of the original packets and the length recovery field of the FEC packets.
6. The CSRC lists (if present), header extension (if present), and payload of the recovered packet are calculated as the XOR of those fields in the original packets, plus the payload of the FEC packets (because the FEC packet never contains a CSRC list or header extension itself, and it carries the protected version of the original fields as part of its payload).

The result is an exact reconstruction of the missing packet, bitwise identical to the original. There is no partial recovery with the RFC 2733 FEC scheme. If there are sufficient FEC packets, the lost packet can be perfectly recovered; if not, nothing can be saved.

Unequal Error Protection

Although some payload formats must be recovered exactly, there are other formats in which some parts of the data are more important than others. In these cases it is sometimes possible to get most of the effect while recovering only part of the packet. For example, some audio codecs have a minimum number of bits that need to be recovered to provide intelligible speech, with additional bits that are not essential but improve the audio quality if they can be recovered. A recovery scheme that recovers only the minimum data will be lower in quality than one that recovers the complete packet, but it may have significantly less overhead.

Alternatively, it is possible to protect the entire packet against some degree of packet loss but give the most important part of the packet a greater degree of protection. In this case the entire packet is recovered with some probability, but the important parts have a higher chance of recovery.

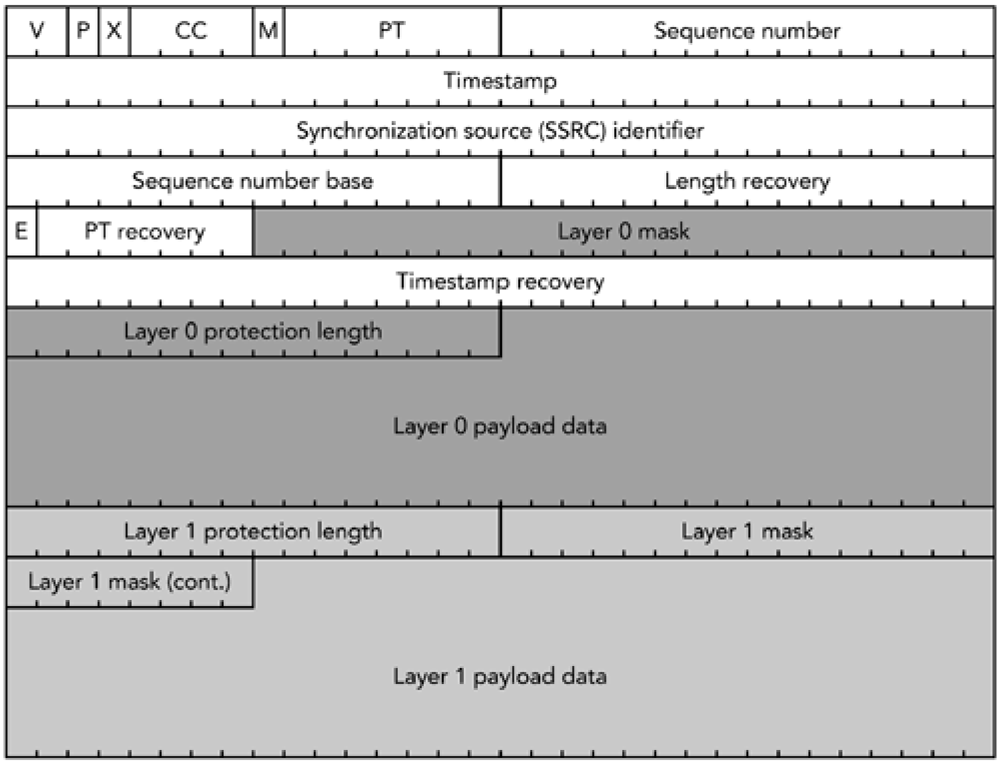
Schemes such as these are known as unequal layered protection (ULP) codes. At the time of this writing, there is no standard for ULP codes applied to RTP. However, there is ongoing work in the IETF to define an extension to the parity FEC codes in

RFC 2733, which will provide this function.[47] This work is incomplete, and the final standard may be slightly different from that described here.

The extension provides for layered coding, with each layer protecting a certain portion of the packet. Each layer may have a different length, up to the length of the longest packet in the group. Layers are arranged so that multiple layers protect the start of the packet, with later parts being protected by fewer layers. This arrangement makes it more likely that the start of the packet can be recovered.

The proposed RTP payload format for ULP based on parity FEC is shown in Figure 9.5. The start of the payload header is identical to that of RFC 2733, but the extension bit is set, and additional payload headers follow to describe the layered FEC operation. The payload data section of the packet contains the protected data for each layer, in order.

Figure 9.5. The RTP Payload Format for ULP Based on Parity FEC



At the time of this writing, there is a move to revise the RTP payload format for ULP-based parity FEC described here, so that in addition to providing layered protection, it also updates the parity FEC format of RFC 2733 to support RTP mixers better. These changes are not expected to change the layered coding concepts described, but it is likely that the details of the packet format will change.

The operation of the ULP-based parity FEC format is similar to that of the standard parity FEC format, except that the FEC for each layer is computed over only part of the packet (rather than the entire packet). Each layer must protect the packets protected by the lower layers, making the amount of FEC protecting the lower layers cumulative with the number of layers. Each FEC packet can potentially contain data for all layers, stacked one after the other in the payload section of the packet. The FEC for the lowest layer appears in all FEC packets; higher layers appear in a subset of the packets, depending on the FEC operation. There is only one FEC stream, independent of the number of layers of protection.

Recovery operates on a per-layer basis, with each layer potentially allowing recovery of part of the packet. The algorithm for recovery of each layer is identical to that of the standard parity FEC format. Each layer is recovered in turn, starting with the base layer, until all possible recovery operations have been performed.

The use of ULP is not appropriate for all payload formats, because for it to work, the decoder must be able to process partial packets. When such partial data is useful, ULP can provide a significant gain in quality, with less overhead than complete FEC protection requires.

Reed–Solomon Codes

Reed–Solomon codes[98] are an alternative to parity codes that offer protection with less bandwidth overhead, at the expense of additional complexity. In particular, they offer good protection against burst loss, where conventional parity codes are less efficient.

Reed–Solomon encoding involves treating each block of data as the coefficient of a polynomial equation. The equation is evaluated over all possible inputs in a certain number base, resulting in the FEC data to be transmitted. Often the procedure operates per octet, making implementation simpler. A full treatment is outside the scope of this book, but the encoding procedure is actually relatively straightforward, and there are optimized decoding algorithms.

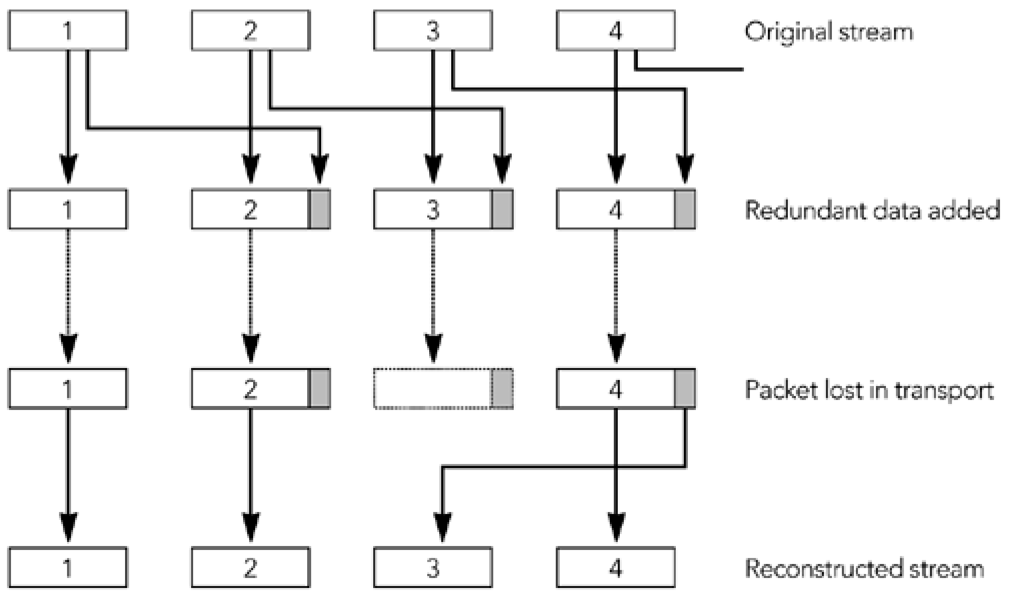
Despite advantages of Reed–Solomon codes compared to parity codes, there is no standard for their use with RTP. Both equal and unequal FEC[48] using Reed–Solomon codes has generated some interest, and a standard is expected to be developed in the future.

Audio Redundancy Coding

The error correction schemes we have discussed so far are independent of the media format being used. However, it is also possible to correct errors in a media-specific way, an approach that can often lead to improved performance.

The first media-specific error correction scheme defined for RTP was audio redundancy coding, specified in RFC 2198.[10], [77] The motivation for this coding scheme was interactive voice telecon-ferences, in which it is more important to repair lost packets quickly than it is to repair them exactly. Accordingly, each packet contains both an original frame of audio data and a redundant copy of a preceding frame, in a more heavily compressed format. The coding scheme is illustrated in Figure 9.6.

Figure 9.6. Audio Redundancy Coding (From C. Perkins, O. Hodson, and V. Hardman, "A Survey of Packet Loss Recovery Techniques for Streaming Media," IEEE Network Magazine, September/October 1998. © 1998 IEEE.)

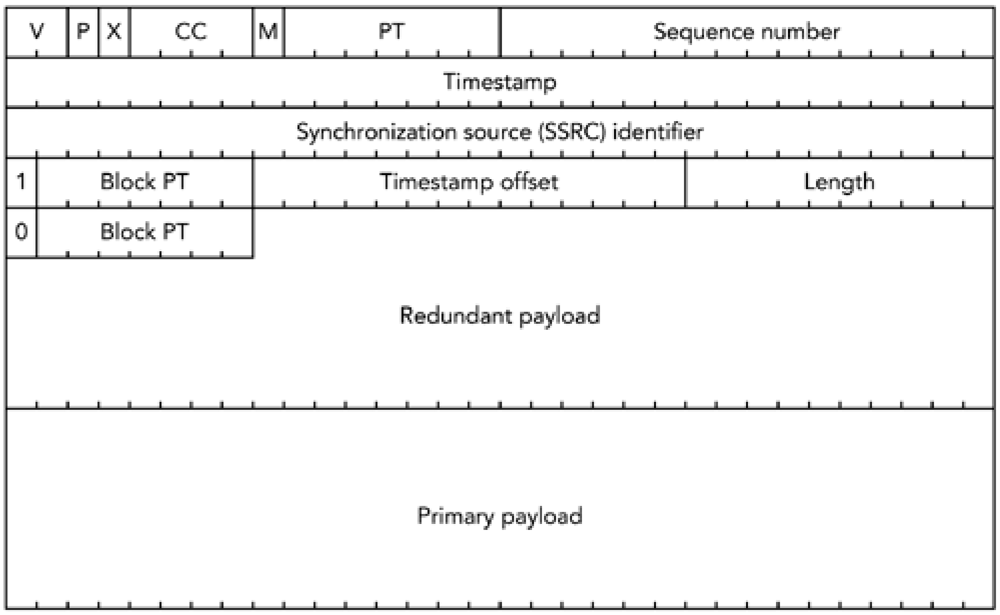


When receiving a redundant audio stream, the receiver can use the redundant copies to fill in any gaps in the original data stream. Because the redundant copy is typically more heavily compressed than the primary, the repair will not be exact, but it is perceptually better than a gap in the stream.

FORMAT OF REDUNDANT AUDIO PACKETS

The redundant audio payload format is shown in Figure 9.7. The RTP header has the standard values, and the payload type is a dynamic payload type representing redundant audio.

Figure 9.7. The RTP Payload Format for Audio Redundancy Coding



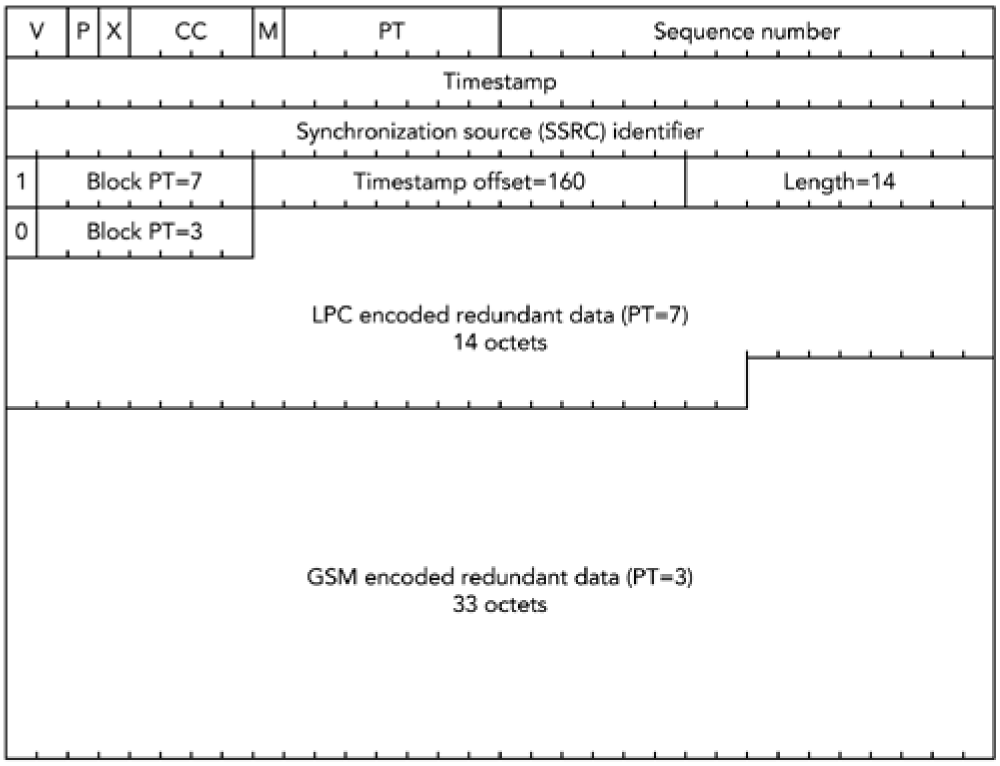
The payload header contains four octets for each redundant encoding of the data, plus a final octet indicating the payload type of the original media. The four-octet payload header for each redundant encoding contains several fields:

- A single bit indicating whether this is a redundant encoding or the primary encoding.
- The payload type of the redundant encoding.
- The length of the redundant encoding in octets, stored as a 10-bit unsigned integer.
- A timestamp offset, stored as a 14-bit unsigned integer. This value is subtracted from the timestamp of the packet, to indicate the original playout time of the redundant data.

The final payload header is a single octet, consisting of one bit to indicate that this is the last header, and the seven-bit payload type of the primary data. The payload header is followed immediately by the data blocks, stored in the same order as the headers. There is no padding or other delimiter between the data blocks, and they are typically not 32-bit aligned (although they are octet aligned).

For example, if the primary encoding is GSM sent with one frame—20 milliseconds—per packet, and the redundant encoding is a low-rate LPC codec sent with one packet delay, a complete redundant audio packet would be as shown in Figure 9.8. Note that the timestamp offset is 160 because 160 ticks of an 8kHz clock represent a 20-millisecond offset (8,000 ticks per second x 0.020 seconds = 160 ticks).

Figure 9.8. A Sample Redundant Audio Packet


V
P
X
CC
M
PT
Sequence number
Timestamp
Synchronization source (SSRC) identifier
1
Block PT=7
Timestamp offset=160
Length=14
0
Block PT=3
LPC encoded redundant data (PT=7)
14 octets
GSM encoded redundant data (PT=3)
33 octets

The format allows the redundant copy to be delayed more than one packet, as a means of countering burst loss at the expense of additional delay. For example, if bursts of two consecutive packet losses are common, the redundant copy may be sent two packets after the original.

The choice of redundant encoding used should reflect the bandwidth requirements of those encodings. The redundant encoding is expected to use significantly less bandwidth than the primary encoding—the exception being the case in which the primary has a very low bandwidth and a high processing requirement, in which case a copy of the primary may be used as the redundancy. The redundant encoding shouldn't have a higher bandwidth than the primary.

It is also possible to send multiple redundant data blocks in each packet, allowing each packet to repair multiple loss events. The use of multiple levels of redundancy is rarely necessary because in practice you can often achieve similar protection with lower overhead by delaying the redundancy. If multiple levels of redundancy are used, however, the bandwidth required by each level is expected to be significantly less than that of the preceding level.

The redundant audio format is signaled in SDP as in the following example:

```
m=audio 1234 RTP/AVP 121 0 5
a=rtpmap:121 red/8000/1
a=fmtp:121 0/5
```

In this case the redundant audio uses dynamic payload type 121, with the primary and secondary encoding being payload type 0 (PCM µ-law) and 5 (DVI).

It is also possible to use dynamic payload types as the primary or secondary encoding—for example:

```
m=audio 1234 RTP/AVP 121 0 122
a=rtpmap:121 red/8000/1
a=fmtp:121 0/122
a=rtpmap:122 g729/8000/1
```

in which the primary is PCM µ-law and the secondary is G.729 using dynamic payload type 122.

Note that the payload types of the primary and secondary encoding appear in both the `m=` and `a=fmtp:` lines of the SDP fragment. Thus the receiver must be prepared to receive both redundant and nonredundant audio using these codecs, both of which are necessary because the first and last packets sent in a talk spurt may be nonredundant.

Implementations of redundant audio are not consistent in the way they handle the first and last packets in a talk spurt. The first packet cannot be sent with a secondary encoding, because there is no preceding data: Some implementations send it using the primary payload format, and others use the redundant audio format, with the secondary encoding having zero length. Likewise, it is difficult to send a redundant copy of the last packet because there is nothing with which to piggyback it: Most implementations have no way of recovering the last packet, but it may be possible to send a nonredundant packet with just the secondary encoding.

LIMITATIONS OF REDUNDANT AUDIO

Although redundant audio encoding can provide exact repair—if the redundant copy is identical to the primary—it is more likely for the redundant encoding to have a lower bandwidth, and hence lower quality, and to provide only approximate repair.

The payload format for redundant audio also does not preserve the complete RTP headers for each of the redundant encodings. In particular, the RTP marker bit and CSRC list are not preserved. Loss of the marker bit does not cause undue problems, because even if the marker bit were transmitted with the redundant information, there would still be the possibility of its loss, so applications would still have to be written with this in mind. Likewise, because the CSRC list in an audio stream is expected to change relatively infrequently, it is recommended that applications requiring this information assume that the CSRC data in the RTP header may be applied to the reconstructed redundant data.

USE OF REDUNDANT AUDIO

The redundant audio payload format was designed primarily for audio teleconferencing. To some extent it performs that job very well; however, advances in codec technology since the format was defined mean that the overhead of the payload format is perhaps too high now.

For example, the original paper proposing redundant audio suggested the use of PCM-encoded audio—160 octets per frame—as the primary, with LPC encoding as the secondary. In this case, the five octets of payload header constitute an acceptable overhead. However, if the primary is G.729 with ten octets per frame, the overhead of the payload header may be considered unacceptable.

In addition to audio teleconferencing, in which adoption of redundant audio has been somewhat limited, redundant audio is used in two scenarios: with parity FEC and with DTMF tones.

The parity FEC format described previously requires the FEC data to be sent separately from the original data packets. A common way of doing this is to send the FEC as an additional RTP stream on a different port; however, an alternative is to treat it as a redundant encoding of the media and piggyback it onto the original media using the redundant audio format. This approach reduces the overhead of the FEC, but it means that the receivers have to understand the redundant audio format, reducing the backward compatibility.

The RTP payload format for DTMF tones and other telephone events[34] suggests the use of redundant encodings because these tones need to be delivered reliably (for example, telephone voice menu systems in which selection is made via DTMF touch tones would be even more annoying if the tones were not reliably recognized). Encoding multiple redundant copies of each tone makes it possible to achieve very high levels of reliability for the tones, even in the presence of packet loss.

Channel Coding

Forward error correction, which relies on the addition of information to the media stream to provide protection against packet loss, is one form of channel coding. The media stream can be matched to the loss characteristics of a particular network path in other ways as well, some of which are discussed in the following sections.

Partial Checksum

Most packet loss in the public Internet is caused by congestion in the network. However, as noted in Chapter 2, Voice and Video Communication over Packet Networks, in some classes of network—for example, wireless—noncongestive loss and packet corruption are common. Although discarding packets with corrupted bits is appropriate in many cases, some RTP payload formats can make use of corrupted data (for example, the AMR audio codecs[41]). You can make use of partially corrupt RTP packets either by disabling the UDP checksum (if IPv4 is used) or by using a transport with a partial checksum.

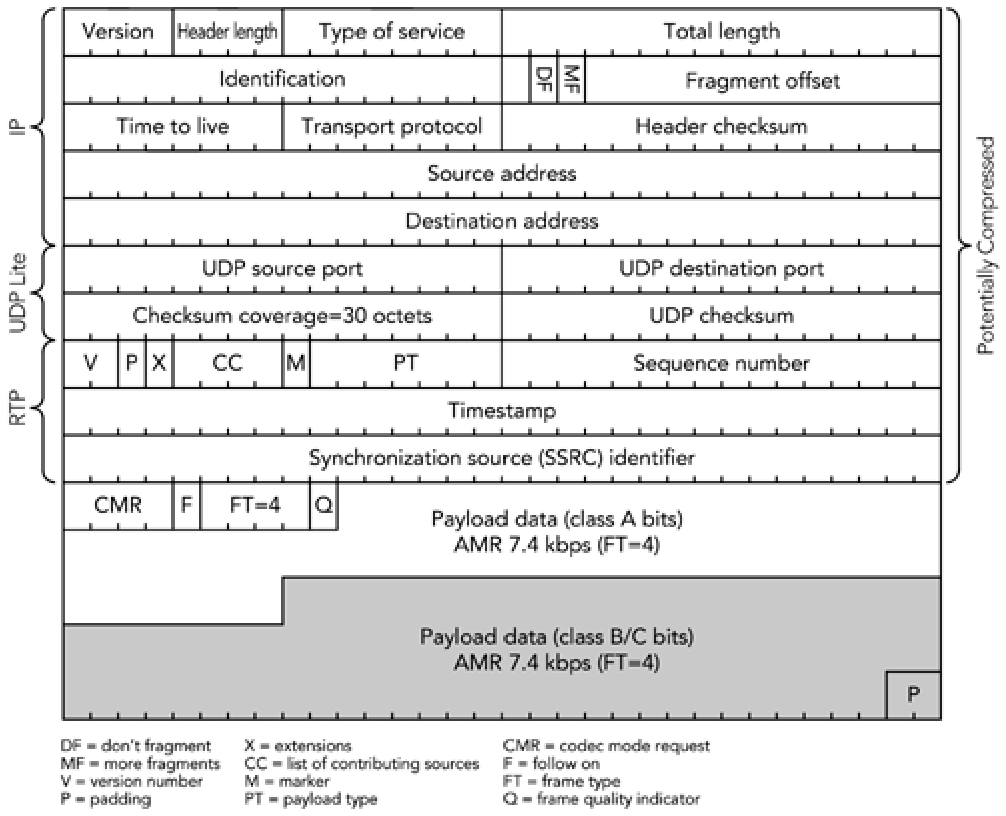
When using RTP with a standard UDP/IPv4 stack, it is possible to disable the UDP checksum entirely (for example, using `sysctlnet.inet.udp.checksum=0` on UNIX machines supporting `sysctl`, or using the `UDP_NOCHECKSUM` socket option with Winsock2). Disabling the UDP checksum has the advantage that packets with corrupted payload data are delivered to the application, allowing some part of the data to be salvaged. The disadvantage is that the packet header may be corrupted, resulting in packets being misdirected or otherwise made unusable.

Note that some platforms do not allow UDP checksums to be disabled, and others allow it as a global setting but not on a per-stream basis. In IPv6-based implementations, the UDP checksum is mandatory and must not be disabled (although the UDP Lite proposal may be used).

A better approach is to use a transport with a partial checksum, such as UDP Lite.[53] This is a work in progress that extends UDP to allow the checksum to cover only part of the packet, rather than all or none of it. For example, the checksum could cover just the RTP/UDP/IP headers, or the headers and the first part of the payload. With a partial checksum, the transport can discard packets in which the headers—or other important parts of the payload—are corrupted, yet pass those that have errors only in the unimportant parts of the payload.

The first RTP payload format to make significant use of partial checksum was the AMR audio codec.[41] This is the codec selected for many third-generation cellular telephony systems, and hence the designers of its RTP payload format placed high priority on robustness to bit errors. Each frame of the codec bit stream is split into class A bits, which are vital for decoding, and class B and C bits, which improve quality if they are received, but are not vital. One or more frames of AMR output are placed into each RTP packet, with the option of using a partial checksum that covers the RTP/UDP/IP headers and class A bits, while the other bits are left unprotected. This lack of protection allows an application to ignore errors in the class B and class C bits, rather than discarding the packets. In Figure 9.9, for example, the shaded bits are not protected by a checksum. This approach appears to offer little advantage, because there are relatively few unprotected bits, but when header compression (see Chapter 11) is used, the IP/UDP/RTP headers and checksum are reduced to only four octets, increasing the gain due to the partial checksum.

Figure 9.9. An Example of the Use of Partial Checksums in the AMR Payload Format



The AMR payload format also supports interleaving and redundant transmission, for increased robustness. The result is a very robust format that copes well with the bit corruption that is common in cellular networks.

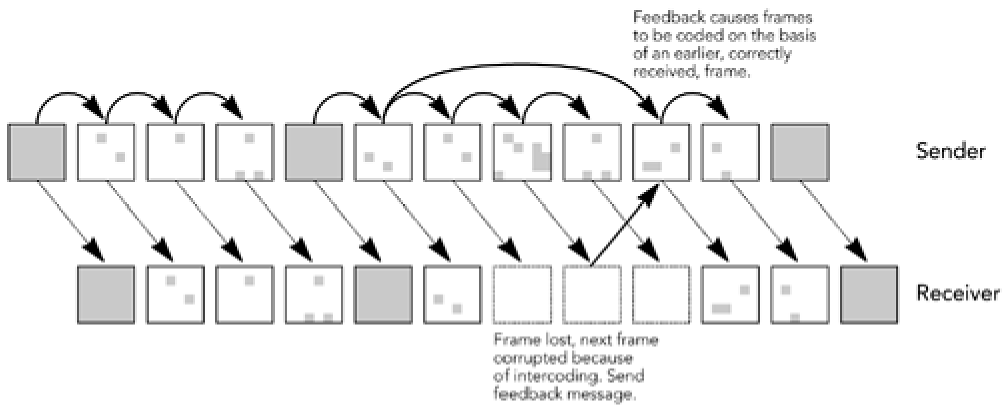
Partial checksums are not a general-purpose tool, because they don't improve performance in networks in which packet loss is due to congestion. As wireless networks become more common, however, it is expected that future payload formats will also make use of partial checksums.

Reference Picture Selection

Many payload formats rely on interframe coding, in which it is not possible to decode a frame without using data sent in previous frames. Interfame coding is most often used in video codecs, in which motion vectors allow panning of the image, or motion of parts of the image, to occur without resending the parts of the preceding frame that have moved. Interframe coding is vital to achieving good compression efficiency, but it amplifies the effects of packet loss (clearly, if a frame depends on the packet that is lost, that frame cannot be decoded).

One solution to making interframe encodings more robust to packet loss is reference picture selection, as used in some variants of H.263 and MPEG-4. This is another form of channel coding, in which if a frame on which others are predicted is lost, future frames are recoded on the basis of another frame that was received (see Figure 9.10). This process saves significant bandwidth compared to sending the next frame with no interframe compression (only intraframe compression).

Figure 9.10. Reference Picture Selection


Feedback causes frames to be coded on the basis of an earlier, correctly received, frame.
Sender
Receiver
Frame lost, next frame corrupted because of intercoding. Send feedback message.

To change the reference picture, it is necessary for the receiver to report individual packet losses to the sender. Mechanisms for feedback are discussed in the next section in the context of retransmission; the same techniques can be used for reference picture selection with minor modification. Work on a standard for the use of reference picture selection in RTP is ongoing, as part of the retransmission profile discussed next.

Retransmission

Losses may also be recovered if the receivers send feedback to the sender, asking it to retransmit packets lost in transit. Retransmission is a natural approach to error correction, and it works well in some scenarios. It is, however, not without problems that can limit its applicability. Retransmission is not a part of standard RTP; however, an RTP profile is under development[44] that provides an RTCP-based framework for retransmission requests and other immediate feedback.

RTCP as a Framework for Retransmission

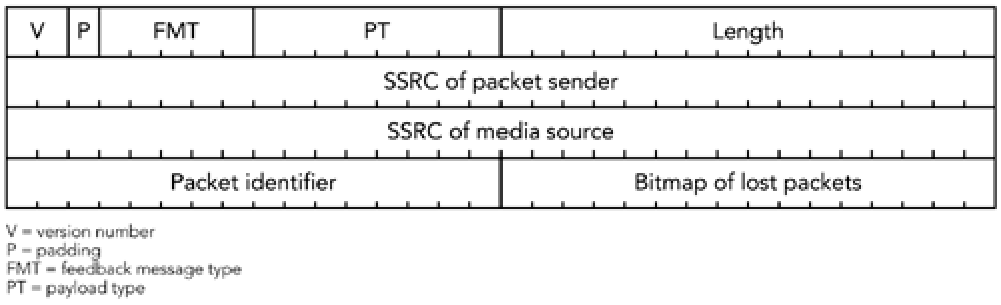
Because RTP includes a feedback channel—RTCP—for reception reports and other data, it is natural to use that channel for retransmission requests too. Two steps are required: Packet formats need to be defined for retransmission requests, and the timing rules must be modified to allow immediate feedback.

PACKET FORMATS

The profile for retransmission-based feedback defines two additional RTCP packet types, representing positive and negative acknowledgments. The most common type is expected to be negative acknowledgments, reporting that a particular set of packets was lost. A positive acknowledgment reports that packets were correctly received.

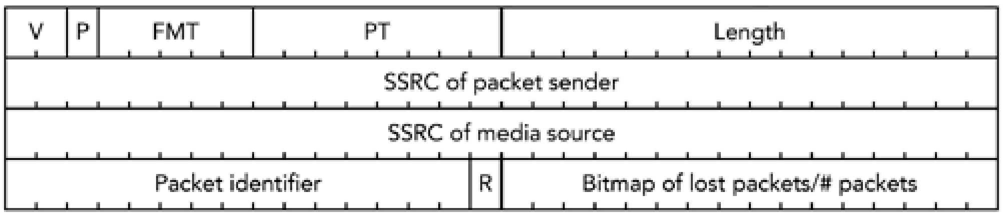
The format of a negative acknowledgment (NACK) is shown in Figure 9.11. The NACK contains a packet identifier representing a lost packet, and a bitmap showing which of the following 16 packets were lost, with a value of 1 indicating loss. The sender should not assume that a receiver has received a packet just because the corresponding position in the bit mask is set to zero; all it knows is that the receiver has not reported the packet lost at this time. On receiving a NACK, the sender is expected to retransmit the packets marked as missing, although it is under no obligation to do so.

Figure 9.11. Format of an RTCP Feedback Negative Acknowledgment



The format of a positive acknowledgment (ACK) is shown in Figure 9.12. The ACK contains a packet identifier representing a correctly received packet, and either a bitmap or a count of the following packets. If the R bit is set to 1, the final field is a count of the number of correctly received packets following the packet identifier. If the R bit is set to zero, the final field is a bitmap showing which of the following 15 packets were also received. The two options allow both long runs of ACKs with few losses (R = 1) and occasional ACKs interspersed with loss (R = 0) to be signaled efficiently.

Figure 9.12. Format of an RTCP Feedback Positive Acknowledgment



The choice between ACK and NACK depends on the repair algorithm in use, and on the desired semantics. An ACK signals that some packets were received; the sender may assume others were lost. On the other hand, a NACK signals loss of some packets but provides no information about the rest (for example, a receiver may send a NACK when an important packet is lost but silently ignore the loss of unimportant data).

Feedback packets are sent as part of a compound RTCP packet, in the same way as all other RTCP packets. They are placed last in the compound packet, after the SR/RR and SDES items. (See Chapter 5, RTP Control Protocol, for a review of RTCP packet formats.)

TIMING RULES

The standard definition of RTCP has strict timing rules, which specify when a packet can be sent and limit the bandwidth consumption of RTCP. The retransmission profile modifies these rules to allow feedback packets to be sent earlier than normal, at the expense of delaying the following packet. The result is a short-term violation of the bandwidth limit, although the longer-term RTCP transmission rate remains the same. The modified timing rules can be summarized as follows:

- When no feedback messages need to be sent, RTCP packets are sent according to the standard timing rules, except that the 5-second minimum interval between RTCP reports is not enforced (the reduced minimum discussed in the section titled Reporting Interval in Chapter 5, RTP Control Protocol, should be used instead).
- If a receiver wants to send feedback before the regularly scheduled RTCP transmission time, it should wait for a short, random dither interval and check whether it has already seen a corresponding feedback message from another receiver. If so, it must refrain from sending and follow the regular RTCP schedule. If the receiver does not see a similar feedback message from any other receiver, and if it has not sent feedback during this reporting interval, it may send the feedback message as part of a compound RTCP packet.
- If feedback is sent, the next scheduled RTCP packet transmission time is reconsidered on the basis of twice the standard interval. The receiver may not send any more feedback until the reconsidered packet has been sent (that is, it may send a feedback packet once for each regular RTCP report).

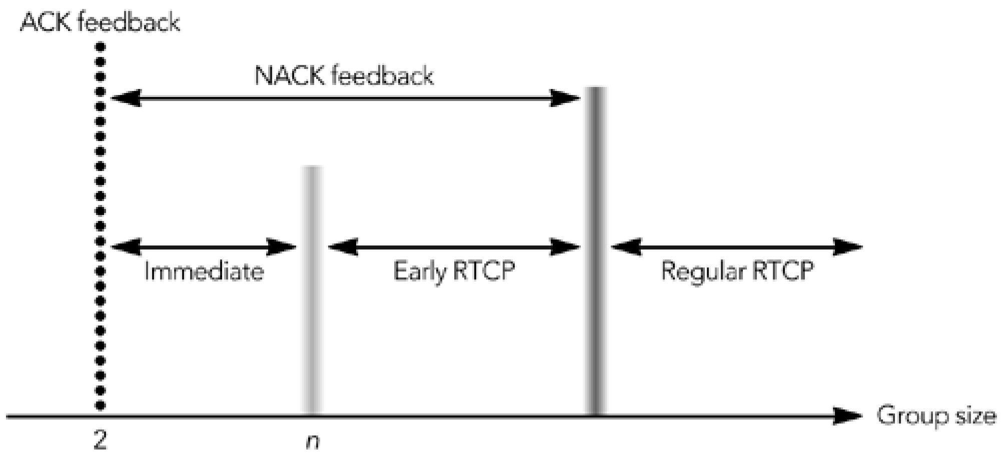
The dither interval is chosen on the basis of the group size and the RTCP bandwidth. If the session has only two participants, the dither interval is set to zero; otherwise, it is set to half of the round-trip time between sender and receiver, multiplied by the number of members (if the round-trip time is unknown, it is set to half of the RTCP reporting interval).

The algorithm for choosing the dither interval allows each receiver to send feedback almost immediately for small sessions. As the number of receivers increases, the rate at which each can send retransmission requests is reduced, but the chance that another receiver will see the same loss and send the same feedback increases.

MODES OF OPERATION

The RTP retransmission profile allows feedback to be sent at a higher rate than standard RTCP, but it still imposes some limitations on allowable send times. Depending on the group size, bandwidth available, data rate, packet loss probability, and desired reporting granularity, an application will operate in one of three modes—immediate, early, and regular—which are illustrated in Figure 9.13.

Figure 9.13. Modes of Feedback



In *immediate feedback mode*, there is sufficient bandwidth to send feedback for each event of interest. In *early feedback mode*, there is not enough bandwidth to provide feedback on all events, and the receiver has to report on a subset of the possible events. Performance is best in immediate mode. As an application moves into early feedback mode, it begins to rely on statistical sampling of the loss and gives only approximate feedback to the sender. The boundary between immediate and early modes, indicated by group size *n* in Figure 9.13, varies depending on the data rate, group size, and fraction of senders.

In both immediate and early modes, only NACK packets are allowed. If the session has only two participants, ACK mode can be used. In ACK mode, positive acknowledgments of each event are sent, providing more detailed feedback to the sender (for example, ACK mode might allow a video application to acknowledge each complete frame, enabling reference picture selection to operate efficiently). Again, the bandwidth limitations of the retransmission profile must be respected.

Applicability

The main factor that limits the applicability of retransmission is feedback delay. It takes at least one round-trip time for the retransmission request to reach the sender and for the retransmitted packet to reach the receiver. This delay can affect interactive applications because the time taken for a retransmission may exceed the acceptable delay bounds. For streaming, and other applications in which the delay bounds are less strict, retransmission can be effective.[69]

Retransmission allows a receiver to request repair of only those packets that are lost, and allows it to accept loss of some packets. The result can be very efficient repair, given the right circumstances. But retransmission becomes inefficient in certain cases, such as these:

- Each retransmission request uses some bandwidth. When the loss rate is low, the bandwidth used by the requests is low, but as losses become more common, the amount of bandwidth consumed by requests increases.
- If the group is large and many receivers see the same loss, they may all request retransmissions at once. Many requests use a lot of bandwidth, and the implosion of requests may overwhelm the sender.
- If the group is large and each receiver sees a different loss, the sender will have to retransmit most packets even though each receiver lost only a small fraction of the packets.

Retransmission works best when groups are small and the loss rate is relatively low. When the number of receivers, or the loss rate, increases, requesting retransmission of lost packets rapidly becomes inefficient. Eventually, a cutoff is reached beyond which the use of forward error correction is more effective.

For example, Handley has observed[122] multicast groups in which *most* packets are lost by at least one receiver. The result could be a retransmission request for almost every packet, which would require tremendous overhead. If forward error correction is used, each FEC packet repairs multiple losses, and the amount of repair data that has to be sent is much lower.

The retransmitted packet does not have to be identical to the original. This flexibility allows retransmission to be used in cases when it might otherwise be inefficient, because the sender can respond to requests by sending an FEC packet, rather than another copy of the original.[85] The fact that the retransmitted and original packets do not have to be identical may also allow a single retransmission to repair many losses.

Implementation Considerations

If error correction is used, an RTP implementation can be made significantly more robust to the adverse effects of IP networks. These techniques come at a price, though: The implementation becomes somewhat more complex, with the receiver needing a more sophisticated playout buffer algorithm, and the sender needing logic to decide how much recovery data to include and when to discard that data.

At a Receiver

Use of these error correction techniques requires that the application have a more sophisticated playout buffer and channel-coding framework than it might otherwise need. In particular, it needs to incorporate FEC and/or retransmission delay into its playout point calculation, and it needs to allow for the presence of repair data in playout buffers.

When calculating the playout point for the media, a receiver has to allow sufficient time for the recovery data to arrive. This may mean delaying the playout of audio/video beyond its natural time, depending on the time needed to receive the recovery data, and the desired playout point of the media.

For example, an interactive voice telephony application might want to operate with a short jitter buffer and a playout delay of only one or two packets' worth of audio. If the sender uses a parity FEC scheme such as that shown in Figure 9.2, in which an FEC packet is sent after every four data packets, the FEC data will be useless because it will arrive after the application has played out the original data it was protecting.

How does an application know when recovery data is going to arrive? In some cases the configuration of the repair is fixed and can be signaled in advance, allowing the receiver to size its playout buffers. Either the signaling can be implicit (for example, RFC 2198 redundancy in which the sender can insert zero-length redundant data into the first few packets of an audio stream, allowing the receiver to know that real redundancy data will follow in later packets), or it can be explicit as part of session setup (for example, included in the SDP during a SIP invitation).

Unfortunately, advance signaling is not always possible, because the repair scheme can change dynamically, or because the repair time cannot be known in advance (for example, when retransmission is used, the receiver has to measure the round-trip time to the sender). In such cases it is the responsibility of the receiver to adapt to make the best use of any repair data it receives, by either delaying media playout or discarding repair data that arrives late. Generally the receiver must make such adaptation without the help of the sender, relying instead on its own knowledge of the application scenario.

A receiver will need to buffer arriving repair data, along with the original media packets. How this is done depends on the form of repair: Some schemes are weakly coupled with the original media, and a generic channel-coding layer can be used; others are tightly coupled to the media and must be integrated with the codec.

Examples of weak coupling include parity FEC and retransmission in which repairs can be made by a general-purpose layer, with no knowledge of the contents of the packets. The reason is that the repair operates on the RTP packets, rather than on the media data itself.

In other cases the repair operation is tightly coupled with the media codec. For example, the AMR payload format[41] includes support for partial checksums and redundant transmission. Unlike the audio redundancy defined in RFC 2198, this form of redundant transmission has no separate header and is specific to AMR: Each packet contains multiple frames, overlapping in time with the following packet. In this case the AMR packetization code must be aware of the overlap, and it must ensure that the frames are correctly added to the playout buffer (and that duplicates are discarded). Another example is the reference picture selection available in MPEG-4 and some modes of H.263, in which the channel coding depends on the shared state between encoder and decoder.

At the Sender

When error correction is in use, the sender is also required to buffer media data longer than it normally would. The amount of buffering depends on the correction technique in use: An FEC scheme requires the sender to hold on to enough data to generate the FEC packets; a retransmission scheme requires the sender to hold on to the data until it is sure that the receivers will no longer request retransmission.

The sender has an advantage over the receiver when it comes to buffering because it knows the details of the repair scheme used and can size its buffers appropriately. This is obviously the case when FEC is being used, but it is also true if retransmission is in use (because RTCP allows the sender to calculate the round-trip time to each receiver).

The sender must also be aware of how its media stream is affecting the network. Most techniques discussed in this chapter add additional information to a media stream, which can then be used to repair loss. This approach necessarily increases the data rate of the stream. If the loss were due to network congestion—which is the common case in the public Internet—then this increase in data rate could lead to a worsening of the congestion, and could actually increase the packet loss rate. To avoid these problems, error correction has to be tied to congestion control, which is the subject of Chapter 10.

Summary

In this chapter we have discussed various ways in which errors due to packet loss can be corrected. The schemes in use today include various types of forward error correction and channel coding, as well as retransmission of lost packets.

When used correctly, error correction provides a significant benefit to the perceived quality of a media stream, and it can make the difference between a system being usable or not. If used incorrectly, however, it can lead to a worsening of the problems it was intended to solve, and it can cause significant network problems. The issue of congestion control—adapting the amount of data sent to match the network capacity, as discussed in more detail in Chapter 10, Congestion Control—forms an essential counterpoint to the use of error correction.

One thing that should be clear from this chapter is that error correction usually works by adding some redundancy to a media stream, which can be used to repair lost data. This mode of operation is somewhat at odds with the goal of media compression, which seeks to remove redundancy from the stream. There is a trade-off to be made between compression and error tolerance: At some stage, extra effort spent compressing a media stream is counterproductive, and it is better to use the inherent redundancy for error resilience. Of course, the point at which that line is passed depends on the network, the codec, and the application.

Chapter 10. Congestion Control

Until now we have assumed that it is possible to send a media stream at its natural rate, and that attempts to exceed the natural rate result in packet loss that we must conceal and correct. In the real world, however, our multimedia stream most likely shares the network with other traffic. We must consider the effects we are having on that traffic, and how to be good network citizens.

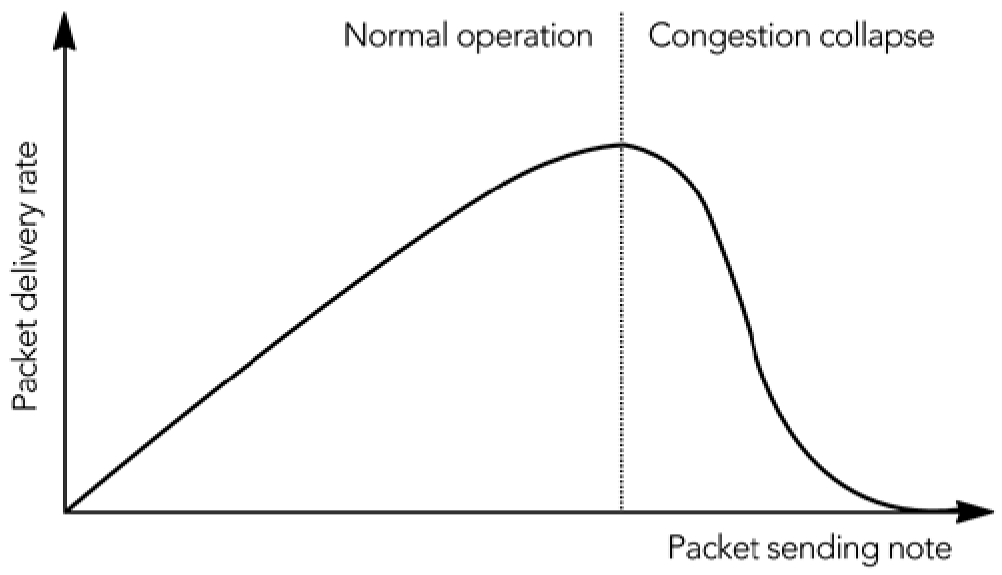
The Need for Congestion Control

Before we discuss the congestion control mechanisms employed on the Internet, and the implications they have for multimedia traffic, it is important to understand what congestion control is, why it is needed, and what happens if applications are not congestion-controlled.

As we discussed in Chapter 2, Voice and Video Communication over Packet Networks, an IP network provides a best-effort packet-switched service. One of the defining characteristics of such a network is that there is no admission control. The network layer accepts all packets and makes its best effort to deliver them. However, it makes no guarantees of delivery, and it will discard excess packets if links become congested. This works well, provided that the higher-layer protocols note packet drops and reduce their sending rate when congestion occurs. If they do not, there is a potential for congestion collapse of the network.

Congestion collapse is defined as the situation in which an increase in the network load results in a decrease in the number of packets delivered usefully by the network. Figure 10.1 shows the effect, with a sudden drop in delivery rate occurring when the congestion collapse threshold is exceeded. Congestion collapse arises when capacity is wasted sending packets through the network that are dropped before they reach their destination—for example, because of congestion at intermediate nodes.

Figure 10.1. Congestion Collapse



Consider a source sending a packet that is dropped because of congestion. That source then retransmits the packet, which again is dropped. If the source continues to resend the packet, and the other network flows remain stable, the process can continue with the network being fully loaded yet no useful data being delivered. This is congestion collapse. It can be avoided if sources detect the onset of congestion and use it as a signal to reduce their sending rate, allowing the congestion to ease.

Congestion collapse is not only a theoretical problem. In the early days of the Internet, TCP did not have an effective congestion control algorithm, and the result was networkwide congestion collapse on several occasions in the mid-1980s.[3] To prevent such collapse, the TCP protocol was augmented with the mechanisms developed by Van Jacobson,[81] which are described in the next section, Congestion Control on the Internet.

With the increase in non-congestion-controlled multimedia traffic, the potential for congestion collapse is again becoming a concern. Various mechanisms have been proposed by which routers can detect flows that do not respond to congestion[73] and penalize those flows with higher loss rates than those that are responsive. Although these mechanisms are not widely deployed at present, it is likely that in the future the network will be more actively policed for compliance with the principles of congestion control.

In addition to congestion collapse, there is another reason for employing congestion control: fairness. The policy for sharing network capacity among flows is not defined by IP, but by the congestion control algorithms of the transport protocols that run above it. The model adopted by TCP is to share the available capacity approximately evenly among all flows. Multimedia flows that use different congestion control algorithms upset this fair sharing of resources, usually to the detriment of TCP (which is squeezed out by the more aggressive multimedia traffic).

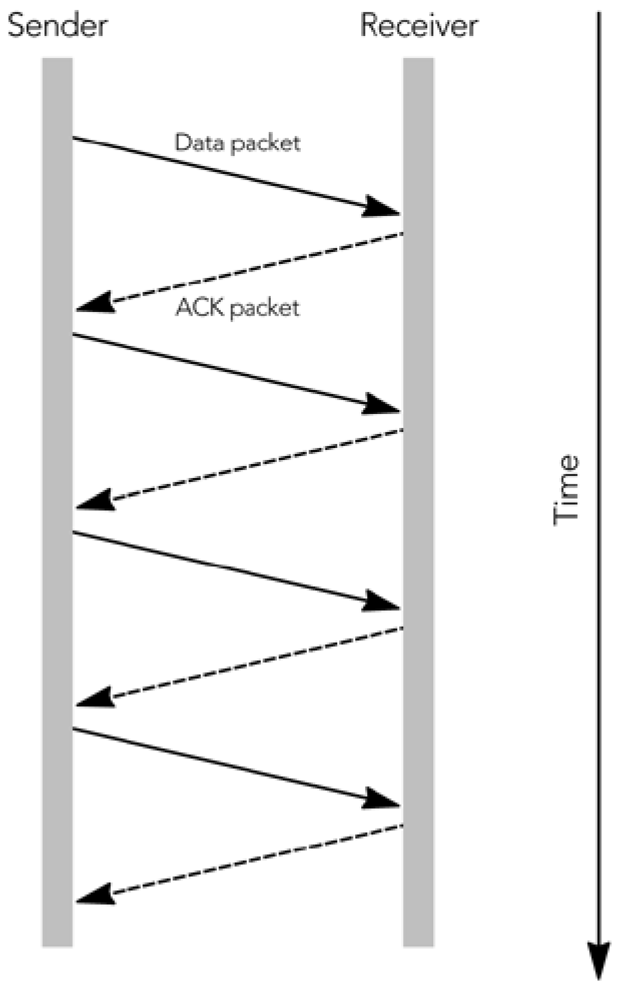
Although it may be desirable to give multimedia traffic priority in some cases, it is not clear that multimedia traffic is always more important than TCP traffic, and that it should always get a larger share of the network capacity. Mechanisms such as Differentiated Services[23], [24] and Integrated Services/RSVP[11] allow for some flows to be given priority in a controlled way, based on application needs, rather than in an undifferentiated manner based on the transport protocol they employ. However, these service prioritization mechanisms are not widely deployed on the public Internet.

Congestion Control on the Internet

Congestion control on the Internet is implemented by the transport protocols layered above IP. It is a simple matter to explain the congestion control functions of UDP—there are none unless they are implemented by the applications layered above UDP—but TCP has an extensive set of algorithms for congestion control.[29], [81], [112]

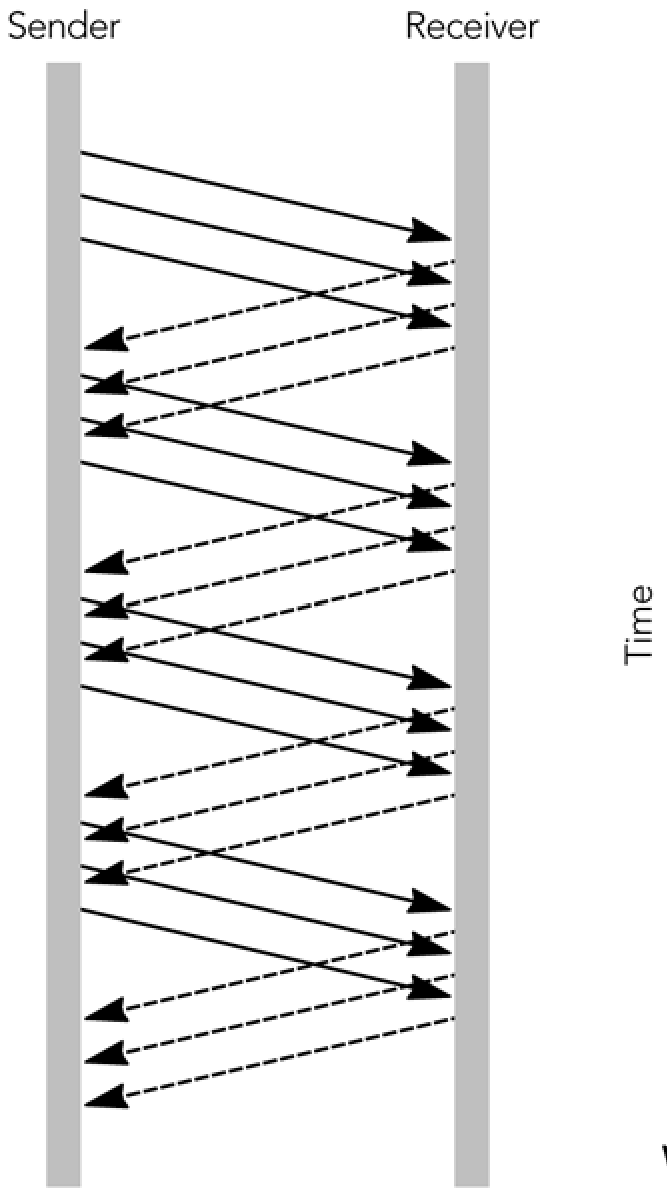
TCP is an example of a sliding window protocol. The source includes sequence numbers with each data packet it sends, and these are echoed back to it in ACK (positive acknowledgment) packets from the receiver. The simplest sliding window protocol requires each packet to be acknowledged immediately, before the next can be sent, as shown in Figure 10.2. This is known as a *stop-and-wait protocol* because the sender must wait for the acknowledgment before it can send the next packet. Clearly, a stop-and-wait protocol provides flow control—preventing the sender from overrunning the receiver—but it also hinders performance, especially if the round-trip time between sender and receiver is long.

Figure 10.2. A Simple Stop-and-Wait Protocol



Use of a larger window allows more than one packet to be sent before an ACK is received. Each TCP ACK packet includes a receiver window, indicating to the source how many octets of data can be accepted at any time, and the highest continuous sequence number from the data packets. The source is allowed to send enough packets to fill the receive window, before it receives an ACK. As ACKs come in, the receive window slides along, allowing more data to be sent. This process is shown in Figure 10.3, in which the window allows three outstanding packets. The larger receive window improves performance compared to a simple stop-and-wait protocol.

Figure 10.3. Use of a Sliding Receiver Window

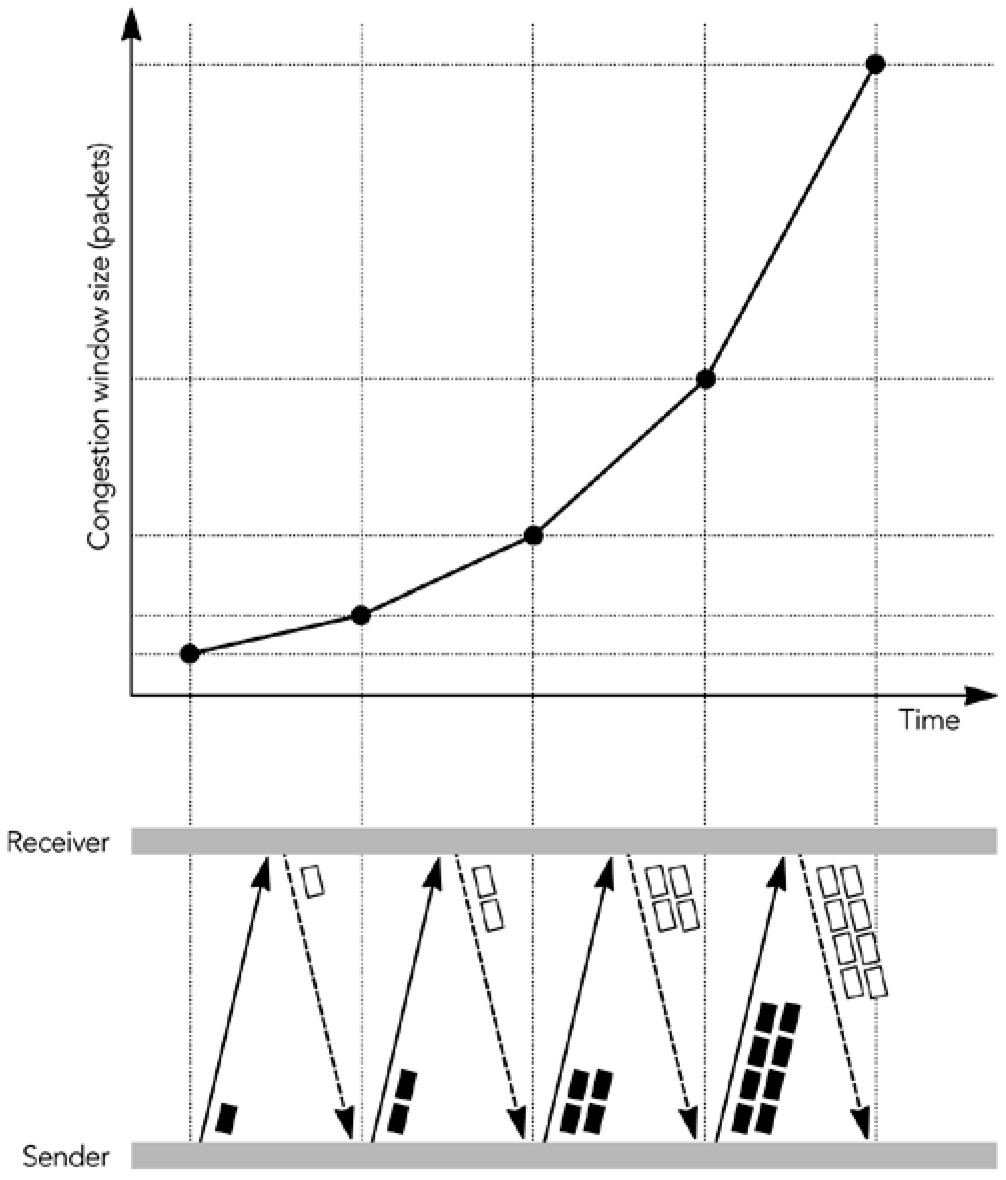


In addition to the receive window, TCP senders implement a congestion window. The congestion window is sized according to an estimate of the network capacity, and it prevents the sender from overrunning the network (that is, it provides congestion control). At any time, the sender can send enough packets to fill the minimum of the congestion window and the receive window.

The congestion window starts at the size of one packet. It increases according to either the slow-start algorithm or the congestion avoidance algorithm, as long as no loss occurs. New connections initially use slow start and then transition to congestion avoidance.

In slow-start mode, the congestion window increases by the size of a packet with each ACK received. As a result, the sender gradually builds up to the full rate the network can handle, as shown in Figure 10.4. Slow start continues until a packet is lost, or until the slow-start threshold is exceeded.

Figure 10.4. TCP Slow Start



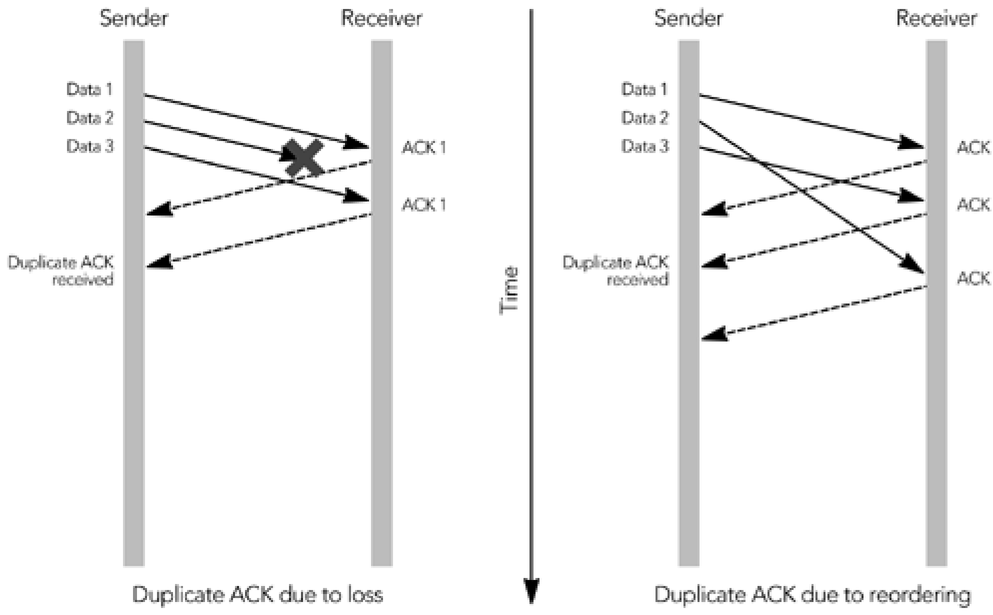
In congestion avoidance mode, the congestion window is increased by the size of one packet per round-trip time (instead of one per ACK). The result is linear growth of the congestion window, an additive increase in the sending rate.

TCP connections increase their sending rate according to one of these two algorithms until a packet is lost. Loss of a packet indicates that the network capacity has been reached and momentary congestion has occurred. The sender can detect congestion in two ways: by a timeout or by receipt of a triple duplicate ACK.

If no ACK is received for an extended time after data has been sent, data is assumed to have been lost. This is a timeout situation: it can occur when the last packet in the receive window is lost, or if there is a (temporary) failure that prevents packets from reaching their destination. When a timeout occurs, the sender sets the slow-start threshold to be one-half the number of packets in flight, or two packets, whichever is larger. It then sets the congestion window to the size of one packet, and enters slow start. The slow-start process continues until the slow-start threshold is exceeded, when the sender enters congestion avoidance mode. Prolonged timeout will cause the sender to give up, breaking the connection.

The other way a sender can detect congestion is by the presence of duplicate ACK packets, which occur when a data packet is lost or reordered (see Figure 10.5). The ACK packets contain the highest continuous sequence number received, so if a data packet is lost, the following ACK packets will contain the sequence number before the loss, until the missing packet is retransmitted. A duplicate ACK will also be generated if a packet is reordered, but in this case the ACK sequence returns to normal when the reordered packet finally arrives.

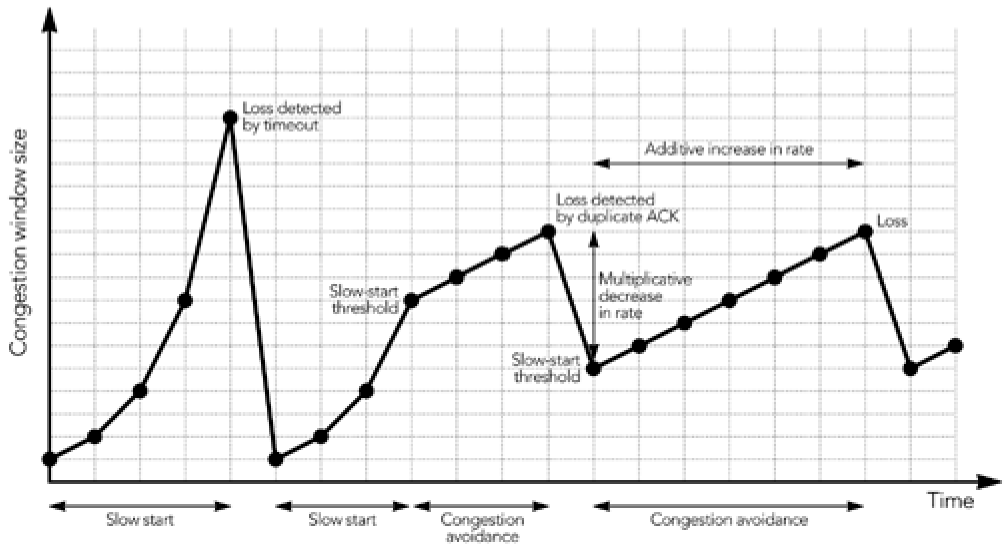
Figure 10.5. Generation of Duplicate ACK Packets



If the sender receives three duplicate ACK packets, it assumes that a packet was lost because of congestion. The sender responds by setting its congestion window and slow-start threshold to one-half the number of packets in flight, or two packets, whichever is larger. It then retransmits the lost packet and enters congestion avoidance mode.

The combination of these algorithms gives TCP connection throughput as shown in Figure 10.6. The characteristic is an additive-increase, multiplicative-decrease (AIMD) sawtooth pattern, with large variations in throughput over short time intervals.

Figure 10.6. Variation in the TCP Sending Rate



This rapid variation in throughput is actually the key behavior that makes a system using TCP stable. The multiplicative decrease ensures a rapid response to congestion, preventing congestion collapse; additive increase, probing for the maximum possible throughput, ensures that the network capacity is fully utilized.

There have been many studies of the variation in TCP throughput, and of the behavior of competing TCP flows.[73], [81], [94], [95], [124] These studies have shown that competing flows get approximately equal shares of the capacity on average, although the jagged throughput profile means that their instantaneous share of the bandwidth is unlikely to be fair.

There is one systematic unfairness in TCP: Because the algorithm responds to feedback, connections with lower round-trip times return feedback faster and therefore work better. Thus, connections with longer network round-trip time systematically receive a lower average share of the network capacity.

Implications for Multimedia

As noted in Chapter 2, Voice and Video Communication over Packet Networks, the vast majority of traffic—over 95% of octets transported—uses TCP as its transport protocol. Because it is desirable for multimedia traffic to coexist peacefully with other traffic on the network, the multimedia traffic should employ a congestion control algorithm that is fair to that of TCP.

As we have seen, TCP adapts to network congestion on very short timescales, and it gives a somewhat higher share of the network bandwidth to connections with short network round-trip times. Also the evolution of TCP has involved several variants in the congestion control algorithms, each having slightly different behavior. These factors make it difficult to define fairness for TCP: Over what timescale is fairness measured—instantaneous or long-term average? Is it a problem that long-distance connections get less bandwidth than local connections? What about changes in the protocol that affect behavior in some conditions?

The more studies that are undertaken, the more it becomes clear that TCP is not entirely fair. Over the short term, one flow will always win out over another. Over the long term, these variations mostly average out, except that connections with longer round-trip times generally achieve lower throughput on average. The different variants of TCP also have an effect—for example, SACK-TCP gets better throughput with certain loss patterns. At best, TCP is fair within a factor of 2 or 3, over the long term.

The variation in TCP behavior actually makes our job, as designers of congestion control for multimedia applications, easier. We don't have to worry too much about being exactly fair to any particular type of TCP, because TCP is not exactly fair to itself. We have to implement some form of congestion control—to avoid the danger of congestion collapse—but as long as it is approximately fair to TCP, that's the best that can be expected.

Some have argued that a multimedia application doesn't want to be fair to TCP, and that it is acceptable for such traffic to take more than its fair share of the bandwidth. After all, multimedia traffic has strict timing requirements that are not shared by the more elastic TCP flows. To some extent this argument has merit, but it is important to understand the problems that can arise from such unfairness.

For example, consider a user browsing the Web while listening to an online radio station. In many cases it might be argued that the correct behavior is for the streaming audio to be more aggressive than the TCP flow so that it steals capacity from the Web browser. The result is music that does not skip, but less responsive Web downloads. In many cases this is the desired behavior. Similarly, a user sending e-mail while making a voice-over-IP telephone call usually prefers having the e-mail delivered more slowly, rather than having the audio break up.

This prioritization of multimedia traffic over TCP traffic cannot be guaranteed, though. Perhaps the Web page the user is submitting is a bid in an online auction, or a stock-trading request for which it's vital that the request be acted on immediately. In these cases, users may be less than pleased if their online radio station delays the TCP traffic.

If your application needs higher priority than normal traffic gets, this requirement should be signaled to the network rather than being implied by a particular transport protocol. Such communication allows the network to make an intelligent choice between whether to permit or deny the higher priority on the basis of the available capacity and the user's preferences. Protocols such as RSVP/Integrated Services[11] and Differentiated Services[24] provide signaling for this purpose. Support for these services is extremely limited at the time of this writing; they are available on some private networks but not the public Internet. Applications intended for the public Internet should consider some form of TCP-friendly congestion control.

Congestion Control for Multimedia

At the time of this writing, there are no standards for congestion control of audio/video streams on the Internet. It is possible either to use TCP directly or to emulate its behavior, as discussed in the next section, TCP-Like Rate Control, although mimicking TCP has various problems in practice. There is also work in progress in the IETF to define a standard for TCP-friendly rate control (see the section titled TCP-Friendly Rate Control) that will likely be more suitable for unicast multimedia applications. The state of the art for multicast congestion control is less clear, but the layered coding techniques discussed later in this chapter have, perhaps, the most promise.

TCP-Like Rate Control

The obvious congestion control technique for audio/video applications is either to use TCP or to emulate the TCP congestion control algorithm.

As discussed in Chapter 2, Voice and Video Communication over Packet Networks, TCP has several properties that make it unsuitable for real-time applications, in particular the emphasis on reliability over timeliness. Nevertheless, some multimedia applications do use TCP, and an RTP-over-TCP encapsulation is defined for use with RTSP (Real-Time Streaming Protocol).[14]

Instead of using TCP directly, it might also be possible to emulate the congestion control algorithm of TCP without the reliability mechanisms. Although no standards exist yet, there have been several attempts to produce such a protocol, with perhaps the most complete being the Rate Adaptation Protocol (RAP), by Rejaie et al.[99] Much like TCP, a RAP source sends data packets containing sequence numbers, which are acknowledged by the receiver. Using the acknowledgment feedback from the receiver, a sender can detect loss and maintain a smoothed average of the round-trip time.

A RAP sender adjusts its transmission rate using an additive-increase, multiplicative-decrease (AIMD) algorithm, in much the same manner as a TCP sender, although since it is rate-based, it exhibits somewhat smoother variation than TCP. Unlike TCP, the congestion control in RAP is separate from the reliability mechanisms. When loss is detected, a RAP sender must reduce its transmission rate but is under no obligation to resend the lost packet. Indeed, the most likely response would be to adapt the codec output to match the new rate and continue without recovering the lost data.

Protocols, like RAP, that emulate—to some degree—the behavior of TCP congestion control exhibit behavior that is most fair to existing traffic. They also give an application more flexibility than it would have with standard TCP, allowing it to send data in any order or format desired, rather than being stuck with the reliable, in-order delivery provided by TCP.

The downside of using TCP, or a TCP-like protocol, is that the application has to adapt its sending rate rapidly, to match the rate of adaptation of TCP traffic. It also has to follow the AIMD model of TCP, with the sudden rate changes that that implies. This is problematic for most audio/video applications because few codecs can adapt quickly and over such large ranges, and because rapid changes in picture or sound quality have been found to be disturbing to the viewer.

These problems do not necessarily mean that TCP, or TCP-like, behavior is inappropriate for all audio/video applications, merely that care must be taken to determine its applicability. The main problem with these congestion control algorithms is the rapid rate changes that are implied. To some extent you can insulate the application from these changes by buffering the output, hiding the short-term variation in rate, and feeding back a smoothed average rate to the codec. This can work well for noninteractive applications, which can tolerate the increased end-to-end delay implied by the buffering, but it is not suitable for interactive use.

Ongoing research into protocols combines TCP-like congestion control with unreliable delivery. If one of these is found suitable for use with RTP, it is expected to be possible to extend RTP to support the necessary feedback (using, for example, the RTCP extensions described in Chapter 9, Error Correction[44]). The difficulty remains in the design of a suitable congestion control algorithm.

At the time of this writing, none of these new protocols are complete. Applications that want to use TCP-like congestion control are probably best suited to the direct use of TCP.

TCP-Friendly Rate Control

The main problem that makes TCP, or TCP-like, congestion control unsuitable for interactive audio/video transport is the large rate changes that can occur over short periods. Many audio codecs are nonadaptive and operate at a single fixed rate (for example, GSM, G.711), or can adapt only between a fixed set of rates (for example, AMR). Video codecs generally have more scope for rate adaptation because both the frame rate and the compression ratio can be adjusted, but the rate at which they can adapt is often low. Even when the media codec can adapt rapidly, it is unclear that doing so is necessarily appropriate: Studies have shown that users prefer stable quality, even if the variable-quality stream has a higher average quality.

Various *TCP-friendly* rate control algorithms have been devised that attempt to smooth the short-term variation in sending rate,[72], [125] resulting in an algorithm more suitable for audio/video applications. These algorithms achieve fairness with TCP when averaged over intervals of several seconds but are potentially unfair in the short term. They have considerable potential for use with unicast audio/video applications, and there is work in progress in the IETF to define a standard mechanism.[78]

TCP-friendly rate control is based on emulation of the steady-state response function for TCP, derived by Padhye et al.[94] The response function is a mathematical model for the throughput of a TCP connection, a predication of the average throughput, given the loss rate and round-trip time of the network. The derivation of the response function is somewhat complex, but Padhye has shown that the average throughput of a TCP connection, *T*, under steady conditions can be modeled in this way:

$$T = \frac{s}{R\sqrt{\frac{2p}{3}} + 3p(1 + 32p^2) \times T_{rto}\sqrt{\frac{3p}{8}}}$$

In this formula, *s* is the packet size in octets, *R* is the round-trip time between sender and receiver in seconds, *p* is the loss event rate (which is not quite the same as the fraction of packets lost; see the following discussion), and T_{rto} is the TCP retransmit time out in seconds.

This equation looks complex, but the parameters are relatively simple to measure. An RTP-based application knows the size of the data packets it is sending, the round-trip time may be obtained from the information in RTCP SR and RR packets, and an approximation of the loss event rate is reported in RTCP RR packets. This leaves only the TCP retransmit timeout, T_{rto}, for which a satisfactory approximation[78] is four times the round-trip time, $T_{rto} = 4R$.

Having measured these parameters, a sender can calculate the average throughput that a TCP connection would achieve over a similar network path, in the steady state—that is, the throughout averaged over several seconds, assuming that the loss rate is constant. This data can then be used as part of a congestion control scheme. If the application is sending at a rate higher than that calculated for TCP, it should reduce the rate of transmission to match the calculated value, or it risks congesting the network. If it is sending at a lower rate, it may increase its rate to match the rate that TCP would achieve. The application operates a feedback loop: change the transmission rate, measure the loss event rate, change the transmission rate to match, measure the loss event rate, repeat. For applications using RTP, this feedback loop can be driven by the arrival of RTCP reception report packets. These reports cause the application to reevaluate and possibly change its sending rate, which will have an effect measured in the next reception report.

For example, if the reported round-trip time is 100 milliseconds, the application is sending PCM µ-law audio with 20-millisecond packets ($s = 200$, including RTP/UDP/IP headers), and the loss event rate is 1% ($p = 0.01$), the TCP equivalent throughput will be $T = 22{,}466$ octets per second (21.9 Kbps).

Because this is less than the actual data rate of a 64-kilobit PCM audio stream, the sender knows that it is causing congestion and must reduce its transmission rate. It can do this by switching to a lower-rate codec—for example, GSM.

This seems to be a simple matter, but in practice there are issues to be resolved. The most critical matter is how the loss rate is measured and averaged, but there are secondary issues with packet sizing, slow start, and noncontinuous transmission:

- The most important issue is the algorithm for calculating the loss event rate, to be fed back to the sender. RTP applications do not directly measure the loss event rate, but instead count the number of packets lost over each RTCP reporting interval and include that number in the RTCP reception report packets as a loss fraction. It is not clear that this is the correct measurement for a TCP-friendly rate control algorithm.

  The loss fraction may not be a sufficient measurement for two reasons: First, TCP responds primarily to loss events, rather than to the actual number of lost packets, with most implementations halving their congestion window only once in response to any number of losses within a single round-trip time. A measure of loss events that treats multiple consecutive lost packets within a round-trip time as a single event, rather than counting individual lost packets, should compete more equally with TCP. The loss event rate could be reported via an extension to RTCP receiver reports, but there is no standard solution at present.

  The second reason is that the reported loss fraction during any particular interval is not necessarily a reflection of the underlying loss rate, because there can be sudden changes in the loss fraction as a result of unrepresentative bursts of loss. This is a problem because oscillatory behavior can result if a sender uses the loss fraction to compute its sending rate directly. To some extent, such behavior is unavoidable—AIMD algorithms are inherently oscillatory[68]—but the oscillations should be reduced as much as possible.

  The solution for reducing oscillations is to average loss reports over a particular time period, but there is no clear consensus among researchers on the correct averaging algorithm. Sisalem and Schulzrinne suggested the use of an exponentially weighted moving average of the loss fraction, which is better than the raw loss rate but can still give rise to oscillations in some cases.[125] Handley et al. suggest the use of a weighted average of the loss event rate, modified to exclude the most recent loss event unless that would increase the average loss event rate.[72],[78] This modified weighted average prevents isolated, unrepresentative loss from corrupting the loss estimate and hence reduces the chance of oscillations.

  Although it is not optimal, it may be sufficient to approximate the loss event rate using an exponentially weighted moving average of the loss fraction reported in RTCP reception reports. The goal is not to be exactly fair to TCP connections, but to be somewhat fair, not cause congestion, and still be usable for audio/video applications. A more complete implementation will extend RTCP reception reports, to return the loss event rate directly.

- The TCP-friendly rate control algorithm assumes that the packet size, *s*, is fixed while the transmission rate, *R*, is varied. A fixed *s* and varied *R* are easy to achieve with some codecs, but difficult with others. The effect on fairness of codec changes that affect both packet size and rate is the subject of ongoing research.

  Likewise, the application probably is using a codec with a limited range of adaptation, and may be unable to send at the rate specified by the TCP-friendly algorithm. The safe solution is to send at the next lower possible rate; and if no lower rate is possible, the transmission may have to be stopped.

- As noted previously, TCP implements a slow-start algorithm when it starts transmission. This slow start allows a sender to probe the network capacity in a gradual manner, rather than starting with a burst that might cause congestion. Slow start is simple to implement for TCP because there is immediate feedback when loss occurs, but it is more difficult with a rate-based protocol that sends feedback at longer intervals.

  The solution adapted for TCP-friendly rate control is for the receiver to send feedback on the reception rate once per round-trip time during slow start. The sender starts at a low rate (one packet per second is suggested) and doubles its transmission rate each time a feedback packet arrives, until the first loss is detected. After loss occurs, the system begins normal operation, starting from the rate used immediately before the loss. This gives a gradual increase up to a "reasonable" value, at which point the rate control algorithm takes over.

  This solution can feasibly be adapted for use with RTP, with the receiver sending acknowledgments according to the extended RTCP feedback profile (see Chapter 9, Error Correction, and Ott et al. 2003[44]) and the sender doubling its rate once per round-trip time until loss occurs. The receiver then reverts to normal RTCP operation, with the sender following the TCP-friendly rate.

  Some applications may not be able to perform this rate-doubling algorithm because of limitations in the set of codecs they support. Such applications might consider sending dummy data initially, and then switching to their most appropriate codec after the sustainable transmission rate is known.

- The final problem is that of discontinuous transmission. If a source stops sending for some time, no feedback on the correct rate will be received. The available capacity might change during the pause, perhaps if another source starts up, so the last-used transmission rate might not be appropriate when the sender resumes. Little can be done to solve this problem, except perhaps to begin again from zero—or from another reduced rate—with the slow-start algorithm, until the correct rate has been determined.

If these problems can be solved, TCP-friendly rate control has the potential to become the standard approach for congestion control of unicast audio/video applications. It is strongly recommended that all unicast RTP implementations include some form of TCP-friendly congestion control.

Implementations should, at least, observe the loss fraction reported in RTCP RR packets, and compare their sending rate with the TCP-friendly rate derived from that loss fraction. If an implementation finds that it is sending significantly faster than the TCP-friendly rate, it should either switch to a lower-rate codec or cease transmission if a lower rate is not possible. These measures prevent congestion collapse and ensure correct functioning of the network.

Implementing the full TCP-friendly rate control algorithm will let an application optimize its transmission to match the network, giving the user the best possible quality. In the process, it will also be fair to other traffic, so as not to disrupt other applications that the user is running. If the application has a suitable codec, or set of codecs, it is strongly recommended that rate control be used—not just to reduce the rate in times of network congestion, but to allow an application to increase its quality when the network is lightly loaded.

Layered Coding

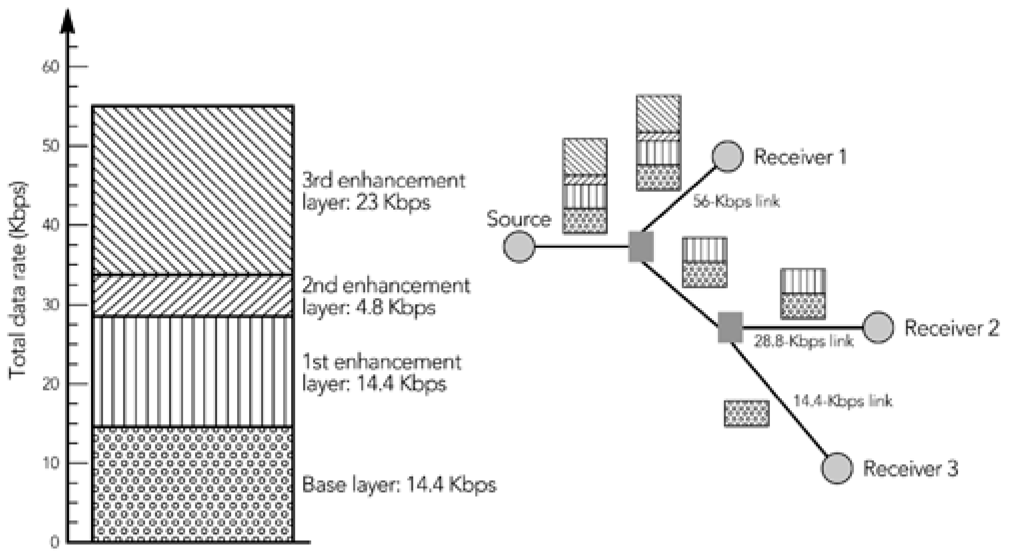
Multicast makes the problem of congestion control significantly more difficult: A sender is required to adapt its transmission to suit many receivers simultaneously, a requirement that seems impossible at first glance. The advantage of multicast is that it allows a sender to efficiently deliver identical data to a group of receivers, yet congestion control requires each receiver to get a media stream that is adapted to its particular network environment. The two requirements seem to be fundamentally at odds with each other.

The solution comes from layered coding, in which the sender splits its transmission across multiple multicast groups, and the receivers join only a subset of the available groups. The burden of congestion control is moved from the source, which is unable to satisfy the conflicting demands of each receiver, to the receivers that can adapt to their individual circumstances.[86]

Layered coding requires a media codec that can encode a signal into multiple layers that can be incrementally combined to provide progressively higher quality. A receiver that receives only the base layer will get a low-fidelity signal, and one that receives the base and one additional layer will get higher quality, with each additional layer increasing the fidelity of the received signal. With the exception of the base, layers are not usable on their own: They merely refine the signal provided by the sum of the lower layers.

The simplest use of layered coding gives each receiver a static subscription to one or more layers. For example, the sender could generate layers arranged as shown in Figure 10.7, in which the base layer corresponds to the capacity of a 14.4-Kbps modem, the combination of base layer and first enhancement layer matches that of a 28.8-Kbps modem, the combination of base and first two enhancement layers matches a 33.6-Kbps modem, and so on. Each layer is sent on a separate multicast group, with the receivers joining the appropriate set of groups so that they receive only the layers of interest. The multicast-capable routers within the network ensure that traffic flows only on links that lead to interested receivers, placing the burden of adaptation on the receivers and the network.

Figure 10.7. Layered Coding



A related solution involves the use of simulcast, in which the sender generates several complete media streams, adapted to different rates, and receivers join the single most appropriate group. This solution uses more bandwidth at the sender—the sum of the possible rates, rather than the highest possible rate—but it is simpler to implement. It doesn't solve the issues due to transient congestion, but it does provide a good solution to the rate selection problem.

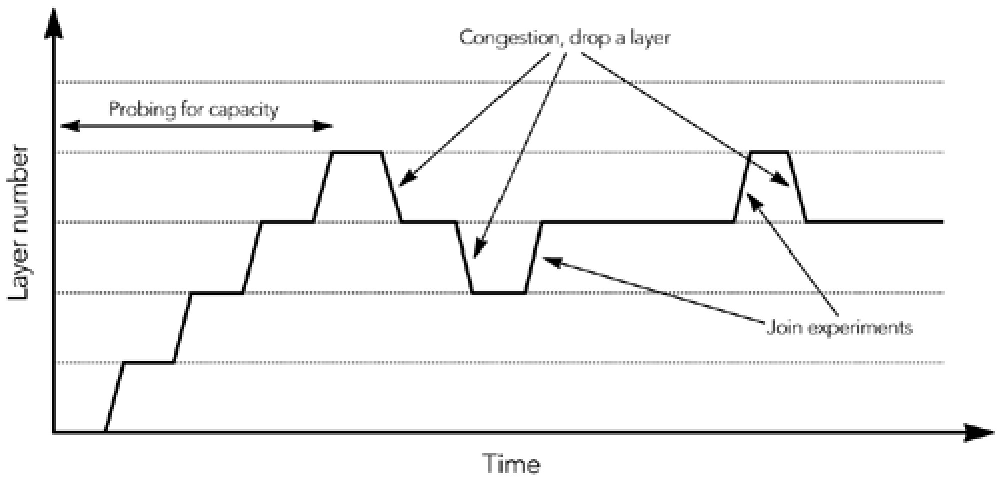
Although static assignment of layers solves the rate selection problem by adapting a media stream to serve many receivers, it doesn't respond to transient congestion due to cross-traffic. It is clear, though, that allowing receivers to dynamically change their layer subscription in response to congestion might provide a solution for multicast congestion control. The basic idea is for each receiver to run a simple control loop:

- If there is congestion, drop one or more layers.
- If there is spare capacity, add a layer.

If the layers are chosen appropriately, the receivers search for the optimal level of subscription, changing their received bandwidth in much the same way that a TCP source probes the network capacity during the slow-start phase. The receivers join layers until congestion is observed, then back off to a lower subscription level.

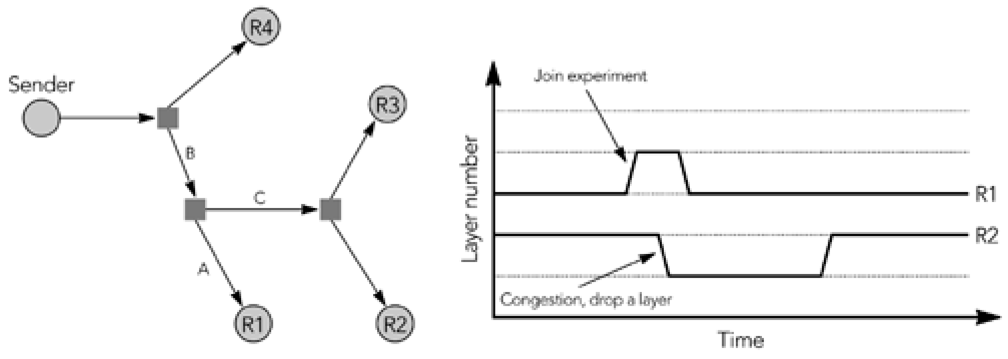
To drive the adaptation, receivers must determine whether they are at too high or too low a subscription level. It is easy to detect over-subscription because congestion will occur and the receiver will see packet loss. Undersubscription is harder to detect because there is no signal to indicate that the network can support a higher rate. Instead, a receiver must try to join an additional layer and immediately leave that layer if it causes congestion, a process known as a *join experiment*.[86] The result looks as shown in Figure 10.8, with the subscription level varying according to network congestion.

Figure 10.8. Adaptation by Varying Subscription Level



The difficulty with join experiments is in trying to achieve shared learning. Consider the network shown in Figure 10.9, in which receiver R1 performs a join experiment but R2 and R3 do not. If the bottleneck link between the source and R1 is link A, everything will work correctly. If the bottleneck is link B, however, a join experiment performed by R1 will cause R2 and R3 to see congestion because they share the capacity of the bottleneck link. If R2 and R3 do not know that R1 is performing a join experiment, they will treat congestion as a signal to drop a layer—which is not the desired outcome!

Figure 10.9. Difficulties with Join Experiments



There is a second problem also. If link C is the bottleneck and R2 leaves a layer, the traffic flowing through the bottleneck will not be affected unless R3 also leaves a layer. Because R2 still sees congestion, it will leave another layer, a process that will repeat until either R2 leaves the session in disgust or R3 also drops a layer.

The solution to both problems is to synchronize receiver join experiments. This synchronization can be achieved if each receiver notifies all others that it is about to join or leave a layer, but such notification is difficult to implement. A better solution is for the sender to include synchronization points—specially marked packets—within the data stream, telling receivers when to perform join experiments.[104]

Other issues relate to the operation of multicast routing. Although multicast joins are fast, processing of a leave request often takes some time. Receivers must allow time for processing of leave requests before they treat the continuing presence of congestion as a signal to leave additional layers. Furthermore, rapid joins or leaves can cause large amounts of routing-control traffic, and this may be problematic.

If these issues can be resolved, and with appropriate choice of bandwidth for each layer, it may be possible to achieve TCP-friendly congestion control with layered coding. The difficulty in applying this sort of congestion control to audio/video applications would then be in finding a codec that could generate cumulative layers with the appropriate bandwidth.

Layered coding is the most promising solution for multicast congestion control, allowing each receiver to choose an appropriate rate without burdening the sender. The Reliable Multicast Transport working group in the IETF is developing a standard for layered congestion control, and it is likely that this work will form the basis for a future congestion control standard for multicast audio/video.

Summary

Congestion control is a necessary evil. As an application designer, you can ignore it, and your application may even appear to work better if you do. However, you may be affecting other traffic on the network—perhaps other instances of your own application—in adverse ways, and you have no way of knowing the relative importance of that traffic. For this reason, and to avoid the possibility of causing congestion collapse, applications should implement some form of congestion control. It is also desirable that the chosen algorithm be approximately fair to TCP traffic, allowing unfairness—priority—to be introduced into the network in a controlled manner.

The standards for congestion control are still evolving, so it is difficult to provide a detailed specification for implementers. Nonetheless, it is strongly recommended that unicast applications implement a TCP-friendly rate control algorithm, because these are the best developed. For multicast applications, the choice of congestion control is less clear, but layered coding seems to be the best alternative at present.

Chapter 11. Header Compression

One area where RTP is often criticized is the size of its headers. Some argue that 12 octets of RTP header plus 28 octets of UDP/IPv4 header is excessive for, say, a 14-octet audio packet, and that a more efficient reduced header could be used instead. This is true to some extent, but the argument neglects the other uses of RTP in which the complete header is necessary, and it neglects the benefit to the community of having a single open standard for audio/video transport.

Header compression achieves a balance between these two worlds: When applied to a link layer, it can compress the *entire* 40-octet RTP/UDP/IP header into 2 octets, giving greater efficiency than a proprietary protocol over UDP/IP could achieve, with the benefits of a single standard. This chapter provides an introduction to the principles of header compression, and a closer look at two compression standards: Compressed RTP (CRTP) and Robust Header Compression (ROHC).

Introductory Concepts

The use of header compression has a well-established history in the Internet community, since the definition of TCP/IP header compression in 1990[4] and its widespread implementation along with PPP for dial-up links. More recently, the standard for TCP/IP header compression has been revised and updated,[25] and new standards for UDP/IP and RTP/UDP/IP header compression have been developed.[26], [27], [37]

A typical scenario for header compression comprises a host connected to the network via a low-speed dial-up modem or wireless link. The host and the first-hop router compress packets passing over the low-speed link, improving efficiency on that link without affecting the rest of the network. In almost all cases, there is a particular bottleneck link where it is desirable to use the bandwidth more efficiently, and there is no need for compression in the rest of the network. Header compression works transparently, on a per-link basis, so it is well suited to this scenario: The compressed link looks like any other IP link, and applications cannot detect the presence of header compression.

These features—per-link operation and application transparency—mean that header compression is usually implemented as part of the operating system (often as part of a PPP implementation). An application usually does not need to be aware of the presence of header compression, although in some circumstances, consideration for the behavior of header compression can increase performance significantly. These circumstances are discussed in the section titled Considerations for RTP Applications later in this chapter.

Patterns, Robustness, and Local Implementation

Compression relies on patterns in the packet headers: Many fields are constant or change in a predictable manner between consecutive packets belonging to the same packet stream. If we can recognize these patterns, we can compress those fields to an indication that "the header changed in the expected manner," rather than explicitly sending them. Only header fields that change in an unpredictable manner need to be transmitted in every header.

An important principle in the design of header compression standards has been robustness. There are two aspects to this: robustness to packet loss, and robustness to misidentified streams. Network links—especially wireless links—can lose or corrupt packets, and the header compression scheme must be able to function in the presence of such damage. The most important requirement is that damage to the compressed bit stream not cause undetectable corruption in the uncompressed stream. If a packet is damaged, the following packets will either be decompressed correctly or discarded. The decompressor should never produce a corrupted packet.

The second robustness issue is that RTP flows are not self-identifying. The UDP header contains no field indicating that the data being transported is RTP, and there is no way for the compressor to determine unambiguously that a particular sequence of UDP packets contains RTP traffic. The compressor needs to be informed explicitly that a stream contains RTP, or it needs to be capable of making an educated guess on the basis of observations of packet sequences. Robust engineering requires that compression must not break anything if mistakenly applied to a non-RTP flow. A misinformed compressor is not expected to compress other types of UDP flows, but it must not damage them.

Together, the principles of pattern recognition and robustness allow the formulation of a general principle for header compression: The compressor will send occasional packets containing full headers, followed by incremental updates indicating which header fields changed in the expected manner and containing the "random" fields that cannot be compressed. The full headers provide robustness: Damage to the incremental packets may confuse the decompressor and prevent operation, but this will be corrected when the next full header arrives.

Finally, it is important that header compression be locally implementable. The aim is to develop a header compression scheme that operates over a single link without end-to-end support: If saving bandwidth is important, two systems can spend processor cycles to compress; otherwise, if processing is too expensive, uncompressed headers are sent. If two systems decide to compress headers on a single link, they should be able to do so in a manner that is invisible to all other systems. As a consequence, header compression should be implemented in the network protocol stack, not in the application. Applications are not expected to implement header compression themselves; an application designer should understand header compression and its consequences, but the implementation should be done as part of the protocol stack of the machines on either side of the link.

Standards

There are two standards for RTP header compression. The original Compressed RTP (CRTP) specification was designed for use with dial-up modems and is a straightforward extension of TCP header compression to work with RTP/UDP/IP. However, with the development of third-generation cellular technology, which uses voice-over-IP as its bearer, it was found that CRTP does not perform well in environments with high link delay and packet loss, so an alternative protocol—Robust Header Compression (ROHC)—was developed.

CRTP remains the protocol of choice for dial-up use because of its simplicity and good performance in that environment. ROHC is considerably more complex but performs much better in a wireless environment. The following sections discuss each standard in detail.

Compressed RTP

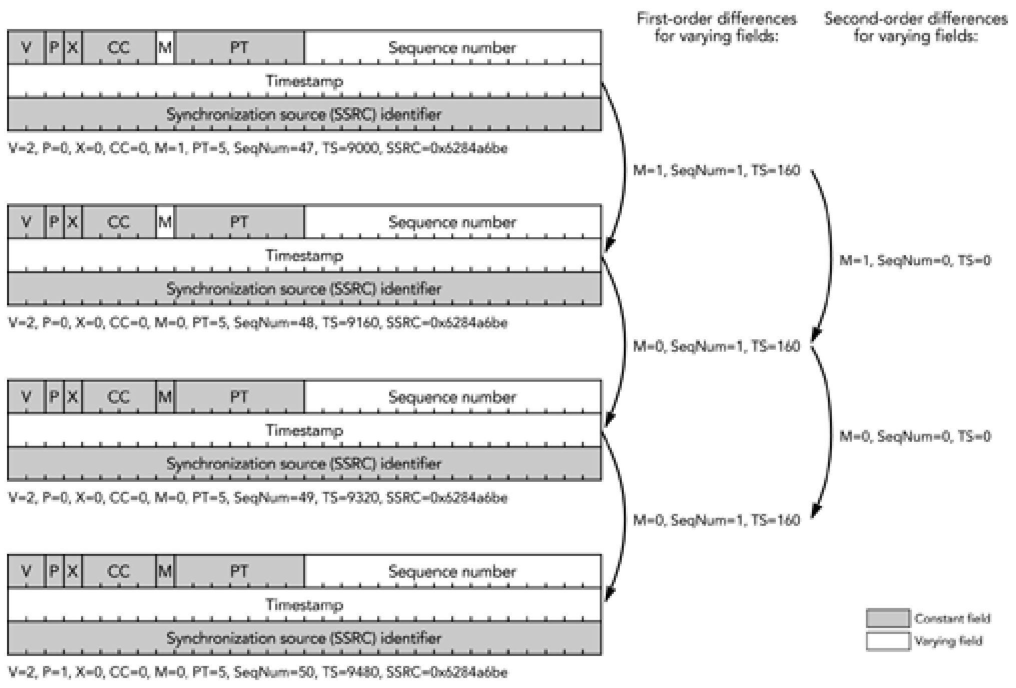
The standard for CRTP is specified in RFC 2508.[26] It was designed for operation over low-speed serial links, such as dial-up modems, that exhibit low bit-error rates. CRTP is a direct out-growth of the Van Jacobson header compression algorithm used for TCP,[4,25] having similar performance and limitations.

The principal gain of TCP header compression comes from the observation that half of the octets in the TCP/IP headers are constant between packets. These are sent once—in a full header packet—and then elided from the following update packets. The remaining gain comes from differential coding on the changing fields to reduce their size, and from eliminating the changing fields entirely for common cases by calculating the changes from the length of the packet.

RTP header compression uses many of the same techniques, extended by the observation that although several fields change in every packet, the difference from packet to packet is often constant and therefore the second-order difference is zero. The constant second-order difference allows the compressor to suppress the unvarying fields and the fields that change predictably from packet to packet.

Figure 11.1 shows the process in terms of just the RTP header fields. The shaded header fields are constant between packets—they have a first-order difference of zero—and do not need to be sent. The unshaded fields are varying fields; they have a nonzero first-order difference. However, their second-order difference is often constant and zero, making the varying fields predictable.

Figure 11.1. Principles of Header Compression



First-order differences for varying fields:
Second-order differences for varying fields:
V
P
X
CC
M
PT
Sequence number
Timestamp
Synchronization source (SSRC) identifier
V=2, P=0, X=0, CC=0, M=1, PT=5, SeqNum=47, TS=9000, SSRC=0x6284a6be
V=2, P=0, X=0, CC=0, M=0, PT=5, SeqNum=48, TS=9160, SSRC=0x6284a6be
V=2, P=0, X=0, CC=0, M=0, PT=5, SeqNum=49, TS=9320, SSRC=0x6284a6be
V=2, P=1, X=0, CC=0, M=0, PT=5, SeqNum=50, TS=9480, SSRC=0x6284a6be
M=1, SeqNum=1, TS=160
M=0, SeqNum=1, TS=160
M=0, SeqNum=1, TS=160
M=1, SeqNum=0, TS=0
M=0, SeqNum=0, TS=0
Constant field
Varying field

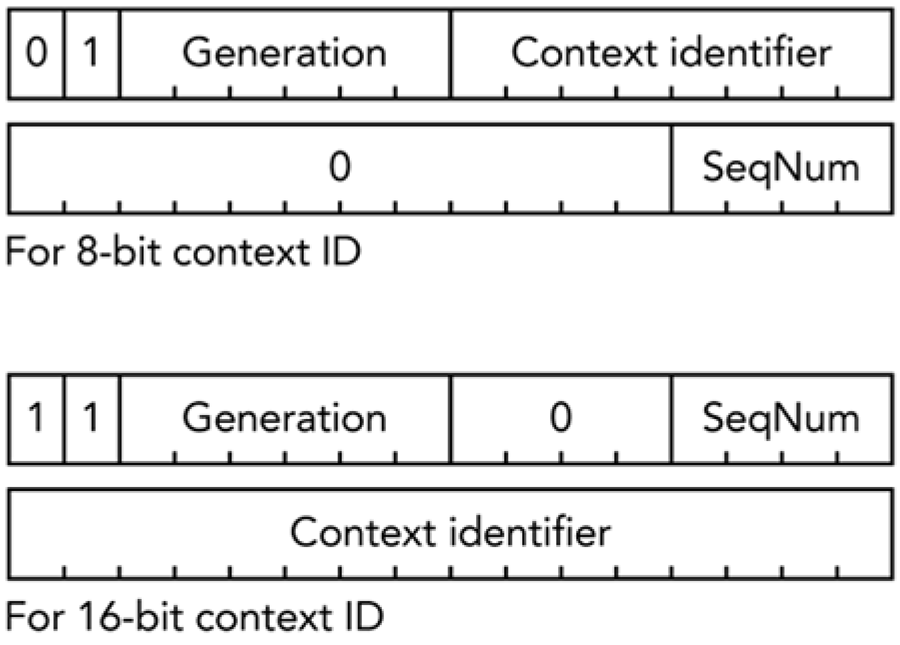
In Figure 11.1 all the fields except the M bit either are constant or have a zero second-order difference. Therefore, if the initial values for the predictable fields are known, only the change in the M bit needs to be communicated.

Operation of CRTP: Initialization and Context

Compressed RTP starts by sending an initial packet containing full headers, thereby establishing the same state in the compressor and decompressor. This state is the initial context of the compressed stream. Subsequent packets contain reduced headers, which either indicate that the decompressor should use the existing context to predict the headers, or contain updates to the context that must be used for the future packets. Periodically a full header packet can be sent to ensure that any loss of synchronization between compressor and decompressor is corrected.

Full header packets comprise the uncompressed original packet, along with two additional pieces of information—a context identifier and a sequence number—as shown in Figure 11.2. The context identifier is an 8- or 16-bit field that uniquely identifies this particular stream. The sequence number is 4 bits and is used to detect packet loss on the link. The additional fields replace the IP and UDP length fields in the original packet, with no loss of data because the length fields are redundant with the link-layer frame length. The full header packet format is common to several compression schemes—including IPv4 and IPv6, TCP, and UDP header compression—hence the inclusion of the generation field, which is not used with RTP header compression.

Figure 11.2. Additional Fields Added to a Full Header Packet



The link-layer protocol indicates to the transport layer that this is a CRTP (rather than IP) *full header* packet. This information allows the transport layer to route the packets to the CRTP decompressor instead of treating them as normal IP packets. The means by which the link layer provides this indication depends on the type of link in use. Operation over PPP links is specified in RFC 2509.[27]

Context identifiers are managed by the compressor, which generates a new context whenever it sees a new stream that it believes it can compress. On receiving a new context identifier, the decompressor allocates storage for the context. Otherwise it uses the context identifier as an index into a table of stored context state. Context identifiers are not required to be allocated sequentially; an implementation that expects relatively few contexts should use a hash table to reduce the amount of storage space needed for the state, and one that expects many contexts should simply use an array with the context identifier as an index.

The context contains the following state information, which is initialized when a full header packet is received:

- The full IP, UDP, and RTP headers, possibly including a CSRC list, for the last packet transferred.
- The first-order difference for the IPv4 ID field, initialized to one when a full header packet is received. (This information is not needed when

RTP-over-UDP/IPv6 is being compressed, because IPv6 does not have an ID field.)

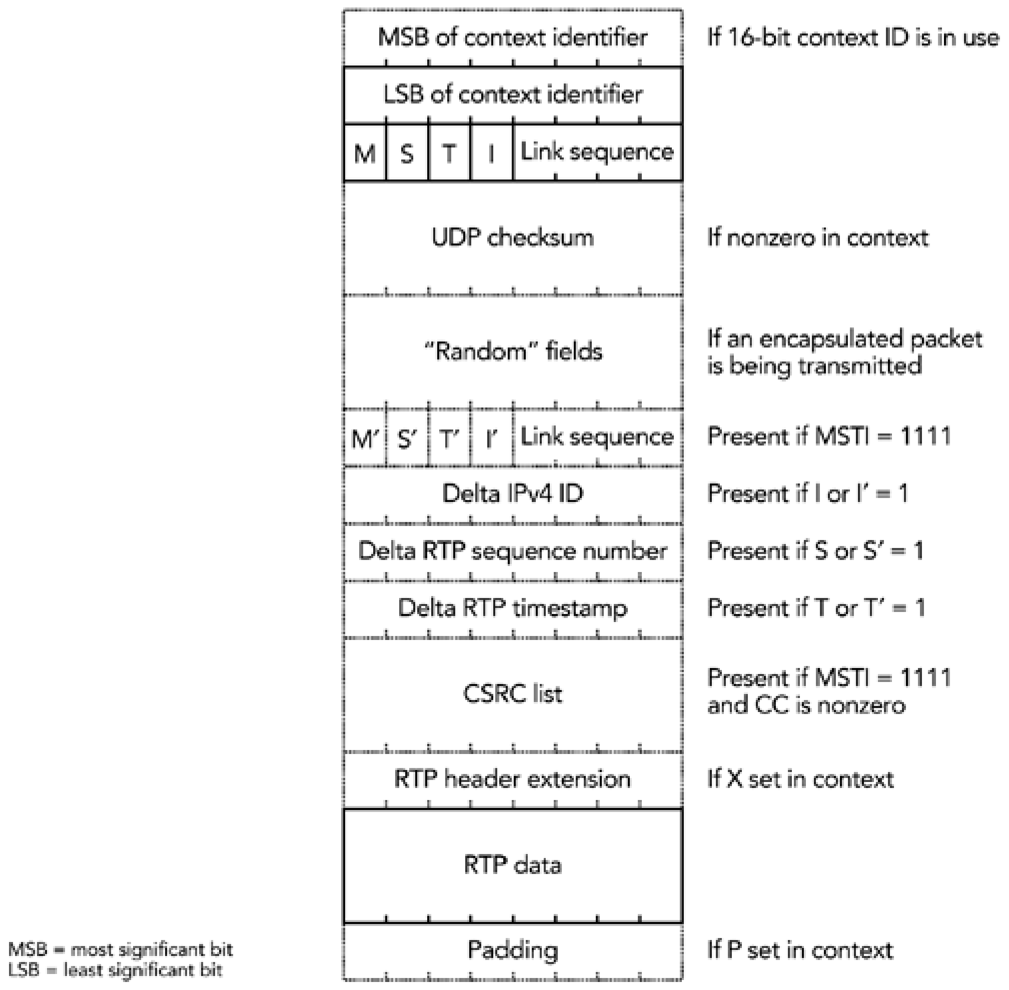
- The first-order difference for the RTP timestamp field, initialized to zero when a full header packet is received.
- The last value of the four-bit CRTP sequence number, used to detect packet loss on the compressed link.

Given the context information, the receiver can decompress each consecutive packet it receives. No additional state is needed, although the state stored in the context may need to be updated with each packet.

Operation of CRTP: Compression and Decompression

After a full header packet has been sent to establish the context, the transition to *Compressed RTP* packets may occur. Each Compressed RTP packet indicates that the decompressor may predict the headers of the next packet on the basis of the stored context. Compressed RTP packets may update that context, allowing for common changes in the headers to be communicated without full header packet being sent. The format of a Compressed RTP packet is shown in Figure 11.3. In most cases, only the fields with solid outline are present, indicating that the next header may be directly predicted from the context. Other fields are present as needed, to update the context, and their presence is either inferred from the context or signaled directly within the compressed header.

Figure 11.3. A Compressed RTP Packet



The compressor observes the characteristics of the stream and omits fields that are constant or that change in a predictable fashion. The compression algorithm will work on any stream of packets in which the bits in the position of the IPv4 ID, RTP sequence number, and RTP timestamp are predictable. In general, the compressor has no way of knowing whether a particular stream really is RTP; it must look for patterns in the headers and, if they are present, start compression. If the stream is not RTP, it is unlikely that the patterns will be present, so it will not be compressed (of course, if the stream is non-RTP but has the appropriate pattern, that the stream can be compressed). The compressor is expected to keep state to track which streams are compressible and which are not, to avoid wasting compression effort.

On receiving a Compressed RTP packet, the decompressor reconstructs the headers. You may reconstruct the IPv4 header by taking the header previously stored in the context, inferring the values of the header checksum and total length fields from the link-layer headers. If the I or I´ bit in the Compressed RTP packet is set, the IPv4 ID is incremented by the delta IPv4 ID field in the Compressed RTP packet, and the stored first-order difference in IPv4 ID in the context is updated. Otherwise the IPv4 ID is incremented by the first-order difference stored in the context.

If multiple IPv4 headers are present in the context—for example, because of IP-in-IP tunneling—their IPv4 ID fields are recovered from the Compressed RTP packet, where they are stored in order as the "random" fields. If IPv6 is being used, there are no packet ID or header checksum fields, so all fields are constant except the payload length, which may be inferred from the link-layer length.

You may reconstruct the UDP header by taking the header previously stored in the context and inferring the length field from the link-layer headers. If the checksum stored in the context is zero, it is assumed that the UDP checksum is not used. Otherwise the Compressed RTP packet will contain the new value of the checksum.

You may reconstruct the RTP header by taking the header previously stored in the context, modified as described here:

- If all of M, S, T, and I are set to one, the packet contains a CC field and CSRC list, along with M´, S´, T´, and I´ fields. In this case, the M´, S´, T´, and I´ fields are used in predicting the marker bit, sequence number, timestamp, and IPv4 ID; and the CC field and CSRC list are updated on the basis of the Compressed RTP packet. Otherwise, the CC field and CSRC list are reconstructed on the basis of the previous packet.
- The M bit is replaced by the M (or M´) bit in the Compressed RTP packet.
- If the S or S´ bit in the Compressed RTP packet is set, the RTP sequence number is incremented by the delta RTP sequence field in the Compressed RTP packet. Otherwise the RTP sequence number is incremented by one.
- If the T or T´ bit in the Compressed RTP packet is set, the RTP timestamp is incremented by the delta RTP timestamp field in the Compressed RTP packet, and the stored first-order difference in timestamp in the context is updated. Otherwise the RTP timestamp is incremented by the stored first-order difference from the context.
- If the X bit is set in the context, the header extension is recovered from the Compressed RTP packet.
- If the P bit is set in the context, the padding is recovered from the Compressed RTP packet.

The context is updated with the newly received headers, as well as any updates to first-order difference in IPv4 ID and RTP timestamp, and the link-layer sequence number. The reconstructed headers, along with the payload data, are then passed to the IP stack for processing in the usual manner.

Typically only the context identifier, link sequence number, M, S, T, and I fields are present, giving a 2-octet header (3 octets if 16-bit context identifiers are used). This compares well with the 40 octets of the uncompressed headers.

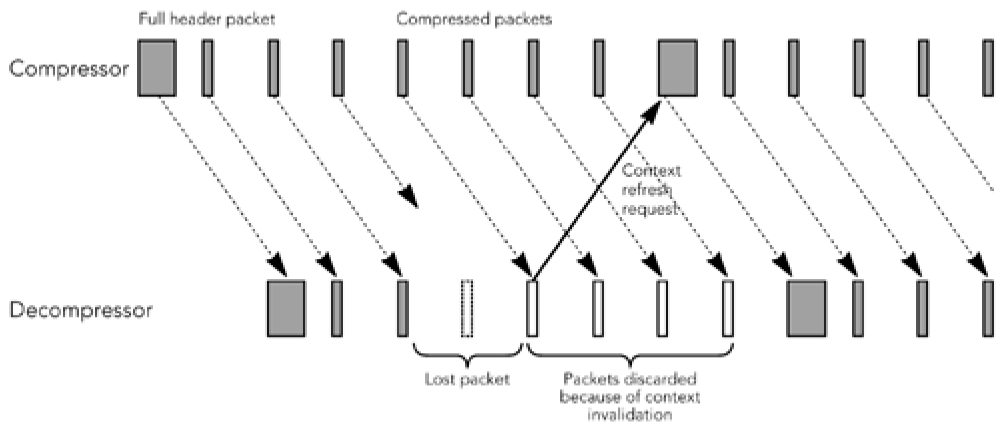
Effects of Packet Loss

An RTP receiver detects packet loss by a discontinuity in the RTP sequence number; the RTP timestamp will also jump. When packet loss occurs upstream of the compressor, new delta values are sent to the decompressor to communicate these discontinuities. Although the compression efficiency is reduced, the packet stream is communicated accurately across the link.

Similarly, if packet loss occurs on the compressed link, the loss is detected by the link-layer sequence number in the Compressed RTP packets. The decompressor then sends a message back to the compressor, indicating that it should send a full header packet to repair the state. If the UDP checksum is present, the decompressor may also attempt to apply the deltas stored in the context twice, to see whether doing so generates a correct packet. The twice algorithm is possible only if the UDP checksum is included; otherwise the decompressor has no way of knowing whether the recovered packet is correct.

The need to request a full header packet to recover the context when loss occurs makes CRTP highly susceptible to packet loss on the compressed link. In particular, if the link round-trip time is high, it is possible that many packets are being received while the context recovery request is being delivered. As shown in Figure 11.4, the result is a loss multiplier effect, in which a single loss affects multiple packets until a full header packet is delivered. Thus, packet loss can seriously affect the performance of CRTP.

Figure 11.4. CRTP Operation When Packet Loss Occurs



The other limiting factor of CRTP is packet reordering. If packets are reordered prior to the compressed link, the compressor is required to send packets containing both sequence number and timestamp updates to compensate. These packets are relatively large—commonly between two and four octets—and they have a significant effect on the compression ratio, at least doubling the size of the compressed headers.

Basic CRTP assumes that there is no reordering on the compressed link. Work is under way on an extension to make CRTP more robust in the face of packet loss and reordering.[43] It is expected that these enhancements will make CRTP suitable for environments with low to moderate amounts of loss or reordering over the compressed link. The Robust Header Compression scheme described in the next section is designed for more extreme environments.

Robust Header Compression

As noted earlier, CRTP does not work well over links with loss and long round-trip times, such as many cellular radio links. Each lost packet causes several subsequent packets to be lost because the context is out of sync during at least one link round-trip time.

In addition to reducing the quality of the media stream, the loss of multiple packets wastes bandwidth because some packets that have been sent are simply discarded, and because a full header packet must be sent to refresh the context. Robust Header Compression (ROHC)[37] was designed to solve these problems, providing compression suitable for use with third-generation cellular systems. ROHC gets significantly better performance than CRTP over such links, at the expense of additional complexity of implementation.

Observation of the changes in header fields within a media stream shows that they fall into three categories:

1. Some fields are static, or mostly static. Examples include the RTP SSRC, UDP ports, and IP addresses. These fields can be sent once when the connection is established, and either they never change or they change very infrequently.
2. Some fields change in a predictable manner with each packet sent, except for occasional sudden changes. Examples include the RTP timestamp and sequence number, and (often) the IPv4 ID field. During periods when these fields are predictable, there is usually a constant relation between them. When sudden changes occur, often only a single field changes unpredictably.
3. Some fields are unpredictable, having essentially random values, and have to be communicated as is, with no compression. The main example is the UDP checksum.

ROHC operates by establishing mapping functions between the RTP sequence number and the other predictable fields, then reliably transferring the RTP sequence number and the unpredictable header fields. These mapping functions form part of the compression context, along with the values of static fields that are communicated at startup or when those fields change.

The main differences between ROHC and CRTP come from the way they handle the second category: fields that usually change in a predictable manner. In CRTP, the value of the field is implicit and the packet contains an indication that it changed in the predictable fashion. In ROHC, the value of a single key field—the RTP sequence number—is explicitly included in all packets, and an implicit mapping function is used to derive the other fields.

Operation of ROHC: States and Modes

ROHC has three states of operation, depending on how much context has been transferred:

1. The system starts in initialization and refresh state, much like the full header mode of CRTP. This state conveys the necessary information to set up the context, enabling the system to enter first- or second-order compression state.
2. First-order compression state allows the system to efficiently communicate irregularities in the media stream—changes in the context—while still keeping much of the compression efficiency. In this state, only a compressed representation of the RTP sequence number, along with the context identifier, and a reduced representation of the changed fields are conveyed.
3. Second-order state is the highest compression level, when the entire header is predictable from the RTP sequence number and stored context. Only a compressed representation of the RTP sequence number and a (possibly implicit) context identifier are included in the packet, giving a header that can be as small as one octet.

If any unpredictable fields are present, such as the UDP checksum, then both first- and second-order compression schemes communicate those fields unchanged. As expected, the result is a significant reduction in the compression efficiency. For simplicity, this description omits further mention of these fields, although they will always be conveyed.

The compressor starts in initialization and refresh state, sending full headers to the decompressor. It will move to either first- or second-order state, sending compressed headers, after it is reasonably sure that the decompressor has correctly received enough information to set up the context.

The system can operate in one of three modes: unidirectional, bidirectional optimistic, and bidirectional reliable. Depending on the mode chosen, the compressor will transition from the initialization and refresh state to either first- or second-order compression state, according to a timeout or an acknowledgment:

1. **Unidirectional mode**. No feedback is possible, and the compressor transitions to first- or second-order state after a predetermined number of packets has been sent.
2. **Bidirectional optimistic mode**. The compressor transitions to the first- or second-order state after a predetermined number of packets has been sent, much as in unidirectional mode, or when an acknowledgment is received.
3. **Bidirectional reliable mode**. The compressor transitions to the first- or second-order state on receipt of an acknowledgment.

The choice of unidirectional or bidirectional feedback depends on the characteristics of the link between compressor and decompressor. Some network links may not support a (convenient) back channel for feedback messages, forcing unidirectional operation. In most cases, though, one of the bidirectional modes can be used, allowing the receiver to communicate its state to the sender.

The compressor starts by assuming unidirectional operation. The decompressor will choose to send feedback if the link supports it, depending on the loss patterns of the link. Receipt of feedback messages informs the compressor that bidirectional operation is desired. The choice between optimistic and reliable mode is made by the decompressor and depends on the capacity of the back channel and the loss characteristics of the link. Reliable mode sends more feedback but is more tolerant of loss.

It's important to keep the difference between ROHC states and modes clear. The state determines the type of information sent in each packet: full headers, partial updates, or fully compressed. The mode determines how and when feedback is sent from the decompressor: (1) never, (2) when there is a problem, or (3) always.

Typically the system transitions from initialization and refresh state to the second-order state after context has been established. It then remains in second-order state until loss occurs, or until a context update is needed because of changes in the stream characteristics.

If loss occurs, the system's behavior depends on the mode of operation. If one of the bidirectional modes was chosen, the decompressor will send feedback causing the compressor to enter the first-order state and send updates to repair the context. This process corresponds to the sending of a context refresh message in CRTP, causing the compressor to generate a full header packet. If unidirectional mode is used, the compressor will periodically transition to lower states, to refresh the context at the decompressor.

The compressor will also transition to the first-order state when it is necessary to convey a change in the mapping for one of the predictable fields, or an update to one of the static fields. This process corresponds to the sending of a compressed packet containing an updated delta field or a full header packet in CRTP. Depending on the mode of operation, the change to first-order state may cause the decompressor to send feedback indicating that it has correctly received the new context.

Operation of ROHC: Robustness and Compression Efficiency

If the compressed link is reliable, ROHC and CRTP have similar compression efficiency, although ROHC is somewhat more complex. For this reason, dial-up modem links do not typically use ROHC, because CRTP is less complex and yields comparable performance.

When there is packet loss on the compressed link, the performance of ROHC shows because of its flexibility in sending partial context updates, and its robust encoding of compressed values. The capability to send partial context updates allows ROHC to update the context in cases in which CRTP would have to send a full header packet. ROHC can also reduce the size of a context update when there is loss on the link. Both of these capabilities give improved performance, compared to CRTP.

The combination of robust encoding of compressed values and sequence number–driven operation is also a key factor. As noted earlier, the ROHC context contains a mapping between the RTP sequence number and the other predictable header fields. Second-order compressed packets convey the sequence number using a window-based least-significant bit (W-LSB) encoding, and the other fields are derived from this. It is largely the use of W-LSB encoding that gives ROHC its robustness to packet loss.

Standard LSB encoding transmits the k least-significant bits of the field value, instead of the complete field. On receiving these k bits, and given the previously transmitted value V_{ref}, the decompressor can derive the original value of the field, provided that it is within a range known as the *interpretation interval*.

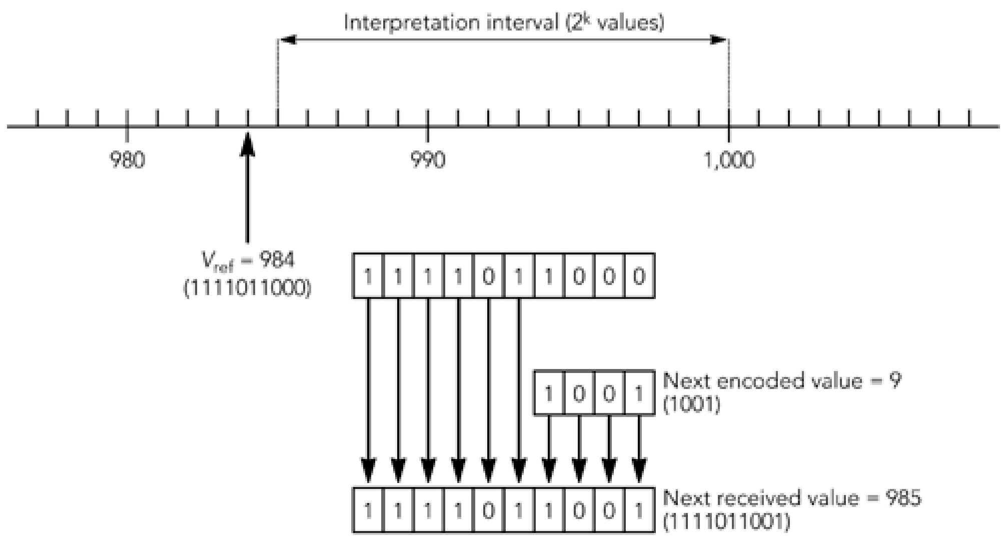
The interpretation interval is the 2^k values surrounding V_{ref}, offset by a parameter p so that it covers the range $V_{ref} - p$ to $V_{ref} + 2^k - 1 - p$. The parameter p is chosen on the basis of the characteristics of the field being transported and conveyed to the decompressor during initialization, forming part of the context. Possible choices include the following:

- If the field value is expected to increase, $p = -1$.
- If the field value is expected to increase or stay the same, $p = 0$.

- If the field value is expected to vary slightly from a fixed value, $p = 2^{(k-1)} + 1$.
- If the field value is expected to undergo small negative changes and large positive changes—for example, the RTP timestamp of a video stream using B-frames—then $p = 2^{(k-2)} - 1$.

As an example, consider the transport of sequence numbers, in which the last transmitted value $V_{ref} = 984$ and $k = 4$ least-significant bits are sent as the encoded form. Assume also that $p = -1$, giving an interpretation interval ranging between 985 and 1,000, as shown in Figure 11.5. The next value sent is 985 (in binary: 1111011001), which is encoded as 9 (in binary: 1001, the 4 least-significant bits of the original value). On receiving the encoded value, the decompressor takes V_{ref} and replaces the k least-signifi-cant bits with those received, restoring the original value.

Figure 11.5. An Example of LSB Encoding



LSB encoding will work provided that the encoded value is within the interpretation interval. If a single packet were lost in the preceding example and the next value received by the decoder were 10 (in binary: 1010), then the restored value would be 986, which is correct. If more than 2^k packets were lost, however, the decoder would have no way of knowing the correct value to decode.

The window-based variant of LSB encoding, W-LSB, maintains the interpretation interval as a sliding window, advancing when the compressor is reasonably sure that the decompressor has received a particular value. Confidence that the window can advance is obtained by various means: In bidirectional optimistic mode, the decompressor sends acknowledgments; in bidirectional optimistic mode, the window advances after a period of time, unless the decompressor sends a negative acknowledgment; and in unidirectional mode, the window simply advances after a period of time.

The advantage of W-LSB encoding is that loss of a small number of packets within the window will not cause the decompressor to lose synchronization. This robustness allows an ROHC decompressor to continue operation without requesting feedback in cases when a CRTP decompressor would fail and need a context update. The result is that ROHC is much less susceptible to the loss multiplier effect than CRTP: A single packet loss on the link will cause a single loss at the output of a ROHC decompressor, whereas a CRTP decompressor must often to wait for a context update before it can continue decompression.

Considerations for RTP Applications

RTP header compression—whether by CRTP or ROHC—is transparent to the application. When header compression is in use, the compressed link becomes a more efficient conduit for RTP packets, but aside from increased performance, an application should not be able to tell that compression is being used.

Nevertheless, there are ways in which an application can aid the operation of the compressor. The main idea is regularity: An application that sends packets with regular timestamp increments, and with a constant payload type, will produce a stream of RTP packets that compresses well, whereas variation in the payload type or interpacket interval will reduce the compression efficiency. Common causes of variation in the interpacket timing include silence suppression with audio codecs, reverse predicted video frames, and interleaving:

- Audio codecs that suppress packets during silent periods affect header compression in two ways: They cause the marker bit to be set, and they cause a jump in the RTP timestamp. These changes cause CRTP to send a packet containing an updated timestamp delta; ROHC will send a first-order packet containing the marker bit and a new timestamp mapping. Both approaches add at least one octet to the size of the compressed header. Despite this reduction in header compression efficiency, silence suppression almost always results in a net saving in bandwidth because some packets are not sent.
- Reverse predicted video frames—for example, MPEG B-frames—have a timestamp less than that of the previous frame.[12] As a result, CRTP sends multiple timestamp updates, seriously reducing compression efficiency. The effects on ROHC are less severe, although some reduction in compression efficiency occurs there also.
- Interleaving is often implemented within the RTP payload headers, with the format designed so that the RTP timestamp increment is constant. In this case, interleaving does not affect header compression, and it may even be

> beneficial. For example, CRTP has a loss multiplier effect when operating on high-delay links, which is less of an issue for inter-leaved streams than for noninterleaved streams. In some cases, though, interleaving can result in packets that have RTP timestamps with nonconstant offsets. Thus, interleaving will reduce the compression efficiency and is best avoided.

The use of UDP checksums also affects compression efficiency. When enabled, the UDP checksum must be communicated along with each packet. This adds two octets to the compressed header, which, because fully compressed RTP/UDP/IP headers are two octets for CRTP and one octet for ROHC, is a significant increase.

The implication is that an application intended to improve compression efficiency should disable the checksum, but this may not necessarily be appropriate. Disabling the checksum improves the compression ratio, but it may make the stream susceptible to undetected packet corruption (depending on the link layer; some links include a checksum that makes the UDP checksum redundant). An application designer must decide whether the potential for receiving corrupt packets outweighs the gain due to improved compression. On a wired network it is often safe to turn off the checksum because bit errors are rare. However, wireless networks often have a relatively high bit-error rate, and applications that may be used over a wireless link might want to enable the checksum.

The final factor that may affect the operation of header compression is generation of the IPv4 ID field. Some systems increment the IPv4 ID by one for each packet sent, allowing for efficient compression. Others use a pseudorandom sequence of IPv4 ID values, making the sequence unpredictable in an attempt to avoid certain security problems. The use of unpredictable IPv4 ID values significantly reduces the compression efficiency because it is necessary to convey the two-octet IPv4 ID in every packet, rather than allowing it to be predicted. It is recommended that IP implementations increment the IPv4 ID by one when sending RTP packets, although it is recognized that the IP layer will not typically know the contents of the packets (an implementation might provide a call to inform the system that the IPv4 ID should increment uniformly for a particular socket).

Summary

The use of RTP header compression—whether CRTP or ROHC—can significantly improve performance when RTP is operating over low-speed network links. When the payload is small—say, low-rate audio, with packets several tens of octets in length—the efficiency to be gained from CRTP with 2-octet compressed headers, compared to 40-octet uncompressed headers, is significant. The use of ROHC can achieve even greater gains in some environments, at the expense of additional complexity.

Header compression is beginning to be widely deployed. ROHC is an essential part of third-generation cellular telephony systems, which use RTP as their voice bearer channel. CRTP is widely implemented in routers and is beginning to be deployed in end hosts.

Chapter 12. Multiplexing and Tunneling

- The Motivation for Multiplexing
- Tunneling Multiplexed Compressed RTP
- Other Approaches to Multiplexing

Multiplexing and tunneling provide an alternative to header compression, improving the efficiency of RTP by bundling multiple streams together inside a single transport connection. The aim is to amortize the size of the headers across multiple streams, reducing the per-stream overhead. This chapter discusses the motivation for multiplexing and outlines the mechanisms that can be used to multiplex RTP streams.

The Motivation for Multiplexing

Header compression reduces the size of the headers of a single RTP stream, on a hop-by-hop basis. It provides very efficient transport but requires cooperation from the network (because header compression works hop-by-hop, rather than end-to-end). Header compression adds to the load on routers in terms of additional computation and flow-specific state, both of which may be unacceptable in systems that have to support many hundreds, or even thousands, of simultaneous RTP streams.

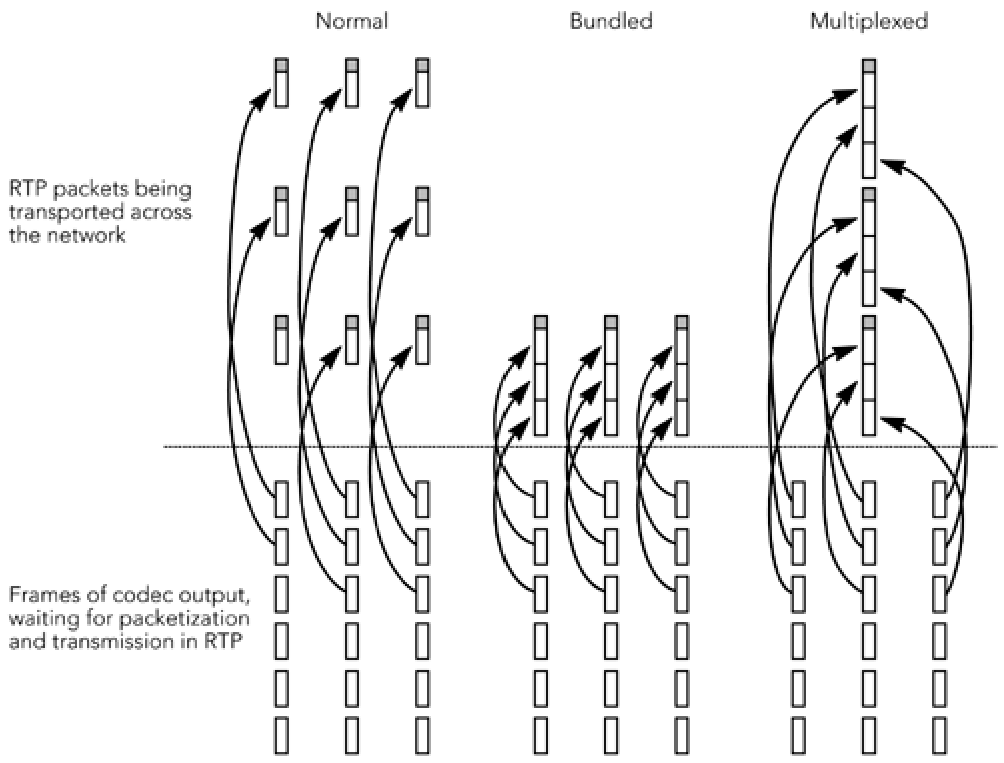
The traditional solution to the issues of computation and state within the network has been to push the complexity to the edge, simplifying the center of the network at the expense of additional complexity at the edges: the end-to-end argument. Application of this solution leads to the suggestion that headers should be reduced end-to-end if possible. You can reduce the header in this way by performing header compression end-to-end, thereby reducing the size of each header, or by placing multiple payloads within each packet to reduce the number of headers.

Applying RTP header compression end-to-end is possible, but unfortunately it provides little benefit. Even if the RTP headers were removed entirely, the UDP/IP headers would still be present. Thus a 28-octet overhead would remain for the typical IPv4 case, a size that is clearly unacceptable when the payload is, for example, a 14-octet audio frame. So there is only one possible end-to-end solution:

Place multiple payloads within each packet, to amortize the overhead due to UDP/IP headers.

The multiple frames of payload data may come from a single stream or from multiple streams, as shown in Figure 12.1. Placing multiple frames of payload data from a single stream into each RTP packet is known is *bundling*. As explained in Chapter 4, RTP Data Transfer Protocol, bundling is an inherent part of RTP and needs no special support. It is very effective at reducing the header overhead, but it imposes additional delay on the media stream because a packet cannot be sent until all the bundled data is present.

Figure 12.1. Bundling versus Multiplexing


Normal
Bundled
Multiplexed
RTP packets being transported across the network
Frames of codec output, waiting for packetization and transmission in RTP

The alternative, *multiplexing*, is the process by which multiple frames of payload data, from different streams, are placed into a single packet for transport. Multiplexing avoids the delay penalty inherent in bundling, and in some cases it can significantly improve efficiency. It is, however, not without problems, which may render it unsuitable in many cases:

- Multiplexing requires many streams, with similar characteristics, to be present at the multiplex point. If frames have nonuniform arrival times, the multiplexing device will have to delay some frames, waiting for others to arrive. Other problems arise if frames have unpredictable sizes because the multiplexing device will not know in advance how many frames can be multiplexed. This may mean that partially multiplexed packets will be sent when the frames are not of convenient sizes to fully multiplex. The results are inefficient transport and variable latency, neither of which is desirable.

- The quality-of-service mechanisms proposed for IP (Differentiated Services and Integrated Services) operate on the granularity of IP flows. Because a multiplex conveys several streams within a single IP layer flow, it is impossible to give those streams different quality of service. This limitation may restrict the usefulness of multiplexing in environments where QoS is used, because it requires all the flows to have identical QoS. On the other hand, if many flows require identical enhanced QoS, the multiplexing may help by reducing the number of flows that the QoS scheme must process.
- Similarly, multiplexing means that all streams within the multiplex have the same degree of error resilience. This is not necessarily appropriate, because some streams may be considered more important than others and would benefit from additional protection.

Despite these issues, in some cases it may still be desirable to multiplex RTP streams. The most common example is the case in which a large number of essentially identical flows are being transferred between two points, something that occurs often when voice-over-IP is being used as a backbone "trunk" to replace conventional telephone lines.

Multiplexing is not directly supported within RTP. If multiplexing streams within RTP is desired, one of the extensions described in this chapter must be used.

Tunneling Multiplexed Compressed RTP

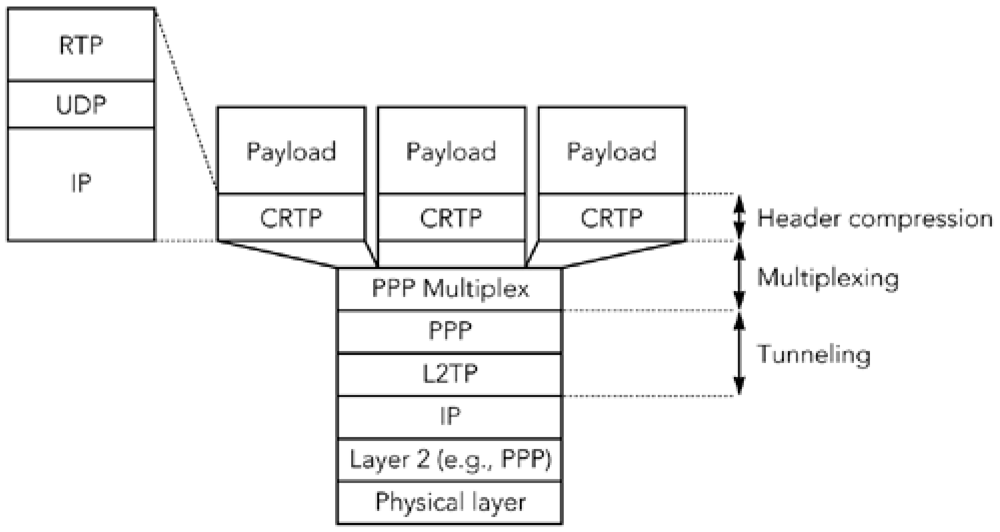
The IETF Audio/Video Transport working group received numerous proposals for RTP multiplexing solutions. Many of these were driven by the needs of specific applications, and although they may have solved the needs of those applications, they generally failed to provide a complete solution. One of the few proposals to keep the semantics of RTP was Tunneling Multiplexed Compressed RTP (TCRTP), which was adopted as the recommended "best current practice" solution.[52]

Basic Concepts of TCRTP

The TCRTP specification describes how existing protocols can be combined to provide a multiplex. It does not define any new mechanisms. The combination of RTP header compression, the Layer Two Tunneling Procotol (L2TP),[31] and PPP multiplexing[39] provides the TCRTP system, with the protocol stack shown in Figure 12.2. Header compression is used to reduce the header overhead of a single RTP payload. Tunneling is used to transport compressed headers and payloads through a multiple-hop IP network without having to compress and decompress at each link.

Multiplexing is used to reduce the overhead of tunnel headers by amortizing a single tunnel header over many RTP payloads.

Figure 12.2. TCRTP Protocol Stack



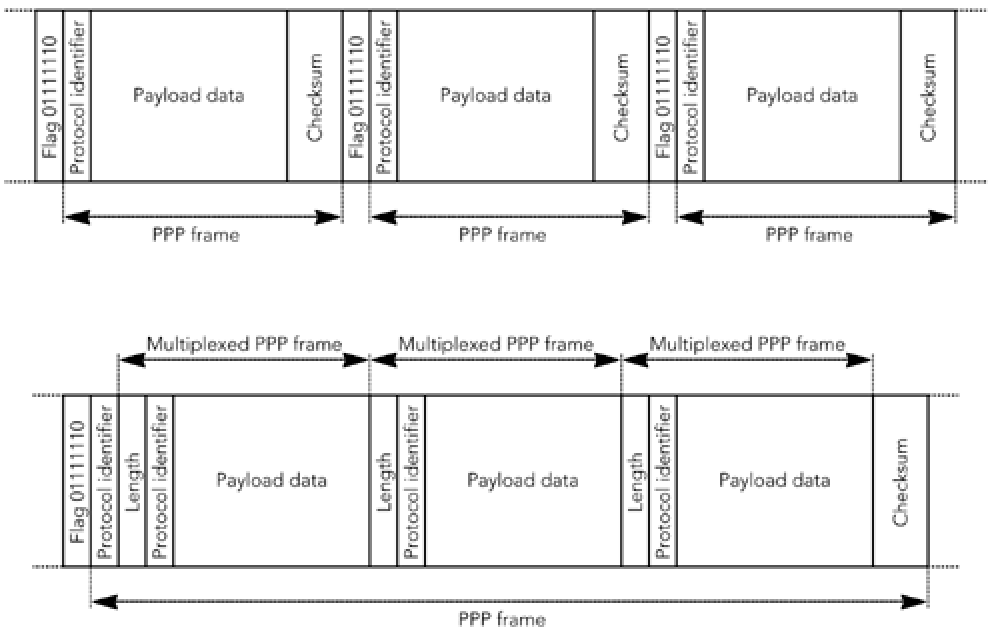
The first stage of TCRTP is to compress the header, which it does in the usual way, negotiating the use of either CRTP or ROHC over a PPP link. The difference is that the PPP link is a virtual interface representing a tunnel rather than a real link. The tunnel is essentially invisible to the header compression, making its presence known only because of the loss and delay characteristics imposed. The concept is much as if the RTP implementation were running over a virtual private network (VPN) link, except that the aim is to provide more efficient transport, rather than more secure transport.

Compared to a single physical link, a tunnel typically has a longer round-trip time, may have a higher packet loss rate, and may reorder packets. As discussed in Chapter 11, Header Compression, links with these properties have adverse effects on CRTP header compression and may lead to poor performance. There are enhancements to CRTP under development that will reduce this problem.[43] ROHC is not a good fit because it requires in-order delivery, which cannot be guaranteed with a tunnel.

The tunnel is created by means of L2TP providing a general encapsulation for PPP sessions over an IP network.[31] This is a natural fit because both CRTP and ROHC are commonly mapped onto PPP connections, and L2TP allows any type of PPP session to be negotiated transparently. If the interface to the PPP layer is correctly implemented, the CRTP/ROHC implementation will be unaware that the PPP link is a virtual tunnel.

Unfortunately, when the overhead of a tunnel header is added to a single compressed RTP payload, there is very little bandwidth savings compared to uncompressed transport of RTP streams. Multiplexing is required to amortize the overhead of the tunnel header over many RTP payloads. The TCRTP specification proposes the use of PPP multiplexing[39] for this purpose. PPP multiplexing combines consecutive PPP frames into a single frame for transport. It is negotiated as an option during PPP connection setup, and it supports multiplexing of variable sizes and types of PPP frames, as shown in Figure 12.3.

Figure 12.3. PPP Multiplexing



PPP adds framing to a point-to-point bit stream so that it can transport a sequence of upper-layer packets. At least four octets of framing are added to each upper-layer packet: a flag octet to signify the start of the PPP frame, followed by a protocol identifier, the mapped upper packet as payload data, and a two-octet check-sum (other framing headers may be present, depending on the options negotiated during channel setup). With several frames multiplexed into one, the PPP overhead is reduced from four octets per packet to two.

The need for TCRTP to include multiplexing becomes clear when the overhead of tunneling is considered. When PPP frames are tunneled over IP via L2TP, there is an overhead of 36 octets per frame (L2TP header compression[46] may reduce this number to 20 octets per frame). This amount of overhead negates the gain from header compression, unless frames are multiplexed before they are tunneled.

Implementing TCRTP

TCRTP has the advantage of being invisible to the upper-layer protocols. An application generating RTP packets cannot tell whether those packets are multiplexed, and it should be possible to add a multiplex to an existing application without changing that application.

The versatility of multiplexing allows great flexibility in the use of TCRTP. For example, TCRTP may be implemented as a virtual network interface on a host to multiplex locally generated packets, or on a router to multiplex packets that happen to flow between two common intermediate points, or as part of a standalone gateway from PSTN (Public Switched Telephone Network) to IP, multiplexing voice calls in a telephony system.

Depending on the scenario, TCRTP may be implemented in many ways. One possible implementation is for the TCRTP stack to present itself to the rest of the system as a standard PPP network interface, which allows the negotiation of RTP header compression. Internally, it will implement PPP multiplexing and L2TP tunneling, but this implementation is transparent to the applications.

The transparency of the TCRTP interface depends primarily on the operating system. If the IP protocol implementation is only loosely coupled to the layer-two interface, it should be possible to add a new interface—TCRTP—relatively easily and transparently. If the IP layer is tightly coupled to the layer-two interface, as may occur in an embedded system in which the TCP/IP implementation is tuned to the characteristics of a particular link, then the process may be more difficult.

A more serious problem may be the interaction between layer-two interfaces and other parts of the network stack. TCRTP is a tunneling protocol, in which compressed RTP/UDP/IP is layered above multiplexed PPP and L2TP, and then layered above UDP/IP. If the operating system does not support the concept of tunnel interfaces, this layering of IP-over-something-over-IP can be problematic and require extensive work. It is also helpful if the system hides tunnel interfaces within the abstraction of a normal network interface because otherwise the different API for tunnel interfaces raises the possibility of application incompatibility with TCRTP.

Within a TCRTP interface, the degree of multiplexing must be carefully controlled to bound the delay, while ensuring that sufficient packets are included in the multiplex to keep the header overhead within acceptable limits. If too few packets are multiplexed together, the per-packet headers become large and negate the effects of the multiplex. We can avoid this problem by delaying sending multiplexed packets until they have accumulated sufficient data to make the header overhead acceptable; however, because interactive applications need a total end-to-end delay of less than approximately 150 milliseconds, the multiplex cannot afford to insert much delay.

Non-RTP traffic can be sent through a TCRTP tunnel but will cause a significant reduction in compression efficiency, so it is desirable to separate it from the RTP traffic. The destination address can be used to separate the two types of traffic if the implementations cooperate to ensure that only RTP packets are sent to a particular destination, or a more extensive examination of the packet headers can be used (for example, checking that the packets are UDP packets destined for a particular port range). Because RTP does not use a fixed port, there is no direct way of distinguishing an RTP flow from a non-RTP flow; so the multiplex cannot be certain that only RTP packets are transported, unless the applications generating those packets cooperate with the multiplex in some manner.

Performance

The bandwidth saved by the use of TCRTP depends on several factors, including the multiplexing gain, the expected packet loss rate within the tunnel, and the rate of change of fields within the multiplexed RTP and IP headers.

Multiplexing reduces the overhead due to the PPP and L2TP headers, and the reduction is greater as an increasing number of streams are multiplexed together into each PPP frame. Performance always increases as more flows are multiplexed together, although the incremental gain per flow is less as the total number of flows in the multiplex increases.

The packet loss rate, and the rate of change of the header fields, can have an adverse effect on the header compression efficiency. Packet loss will cause context invalidation, which will cause compression to switch to a less efficient mode of operation while the context is refreshed. The problem is particularly acute if standard CRTP is used; enhanced CRTP performs much better. Changes in the header fields may also cause the compression to transition to a less efficient mode of operation, sending first-order deltas instead of fully compressed second-order deltas. Little can be done about this, except to note that the section titled Considerations for RTP Applications, in Chapter 11, Header Compression, is also relevant to TCRTP.

The TCRTP specification includes a simple performance model, which attempts to predict the bandwidth used by a voice-over-IP stream (given enhanced CRTP compression), the packet size and duration, the average talk spurt length, the number of packets that can be multiplexed, and estimates of the overhead due to changes in the compressed RTP and IP headers. This model predicts that TCRTP will achieve a rate of 14.4 Kbps for a G.729 stream with 20-millisecond packets, three packets multiplexed into one, and an average talk spurt length of 1,500 milliseconds. This compares well with the 12 Kbps achieved by hop-by-hop CRTP, and the 25.4 Kbps of standard RTP with no header compression or multiplexing (all numbers include the layer-two overhead due to PPP-in-HDLC [High-level Data Link Control] framing).

Performance will, of course, depend heavily on the characteristics of the media and network, but it is believed that the relative performance seen in the example is not unrealistic. Provided that there is sufficient traffic to multiplex, TCRTP will perform significantly better than standard RTP but slightly worse than hop-by-hop header compression.

Other Approaches to Multiplexing

Multiplexing has been an area of some controversy, and considerable discussion, within the IETF. Although TCRTP is the recommended best current practice, there are other proposals that merit further discussion. These include Generic RTP Multiplexing (GeRM), which is one of the few alternatives to TCRTP that maintains RTP semantics, and several application-specific multiplexes.

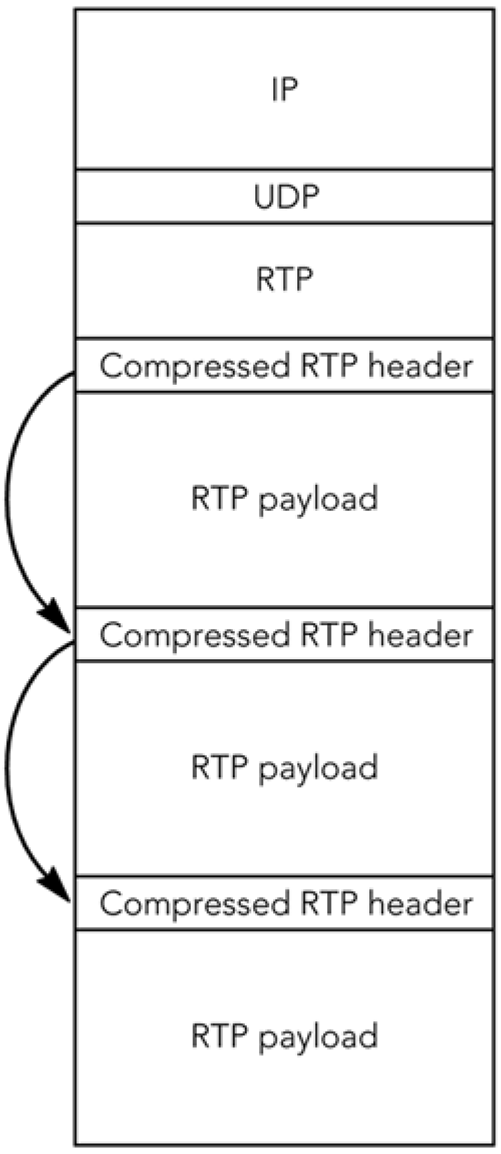
GeRM

Generic RTP Multiplexing (GeRM) was proposed at the IETF meeting in Chicago in August 1998 but was never developed into a complete protocol specification.[45] GeRM uses the ideas of RTP header compression, but instead of compressing the headers between packets, it applies compression to multiple payloads multiplexed within a single packet. All compression state is reinitialized in each new packet, and as a result, GeRM can function effectively end-to-end.

CONCEPTS AND PACKET FORMAT

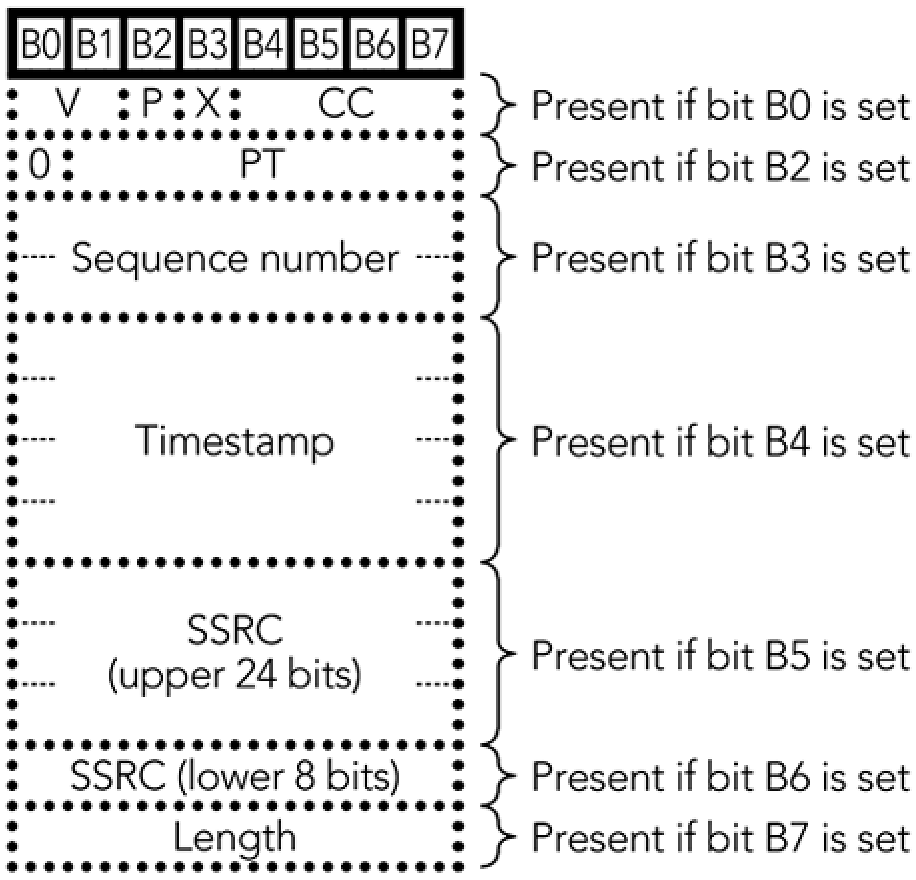
Figure 12.4 shows the basic operation of GeRM. A single RTP packet is created, and multiple RTP packets—known as *subpackets*—are multiplexed inside it. Each GeRM packet has an outer RTP header that contains the header fields of the first subpacket, but the RTP payload type field is set to a value indicating that this is a GeRM packet.

Figure 12.4. A GeRM Packet Containing Three Subpackets



The first subpacket header will compress completely except for the payload type field and length because the full RTP header and the subpacket header differ only in the payload type. The second subpacket header will then be encoded on the basis of predictable differences between the original RTP header for that subpacket and the original RTP header for the first subpacket. The third subpacket header is then encoded off of the original RTP header for the second subpacket, and so forth. Each subpacket header comprises a single mandatory octet, followed by several extension octets, as shown in Figure 12.5.

Figure 12.5. GeRM Subpacket Header



The meanings of the bits in the mandatory octet are as detailed here:

- **B0**: Zero indicates that the first octet of the original RTP header remains unchanged from the original RTP header in the previous subpacket (or outer RTP header if there's no previous subpacket in this packet). That is, V, CC, and P are unchanged. One indicates that the first octet of the original RTP header immediately follows the GeRM header.
- **B1**: This bit contains the marker bit from the subpacket's RTP header.
- **B2**: Zero indicates that the payload type remains unchanged. One indicates that the payload type field follows the GeRM header and any first-octet header that may be present. Although PT is a seven-bit field, it is added as an eight-bit field. Bit 0 of this field is always zero.
- **B3**: Zero indicates that the sequence number remains unchanged. One indicates that the 16-bit sequence number field follows the GeRM header and any first-octet or PT header that may be present.
- **B4**: Zero indicates that the timestamp remains unchanged. One indicates that the 32-bit timestamp field follows the GeRM header and any first-octet, PT, or sequence number header that may be present.

- **B5**: Zero indicates that the most significant 24 bits of the SSRC remain unchanged. One indicates that the most significant 24 bits of the SSRC follow the GeRM header and any first-octet, PT, sequence number, or timestamp field that may be present.
- **B6**: Zero indicates that the least significant eight bits of the SSRC are one higher than the preceding SSRC. One indicates that the least significant eight bits of the SSRC follow the GeRM header and any first-octet, PT, sequence number, timestamp, or MSB SSRC header fields that may be present.
- **B7**: Zero indicates that the subpacket length in bytes (ignoring the subpacket header) is unchanged from the previous subpacket. One indicates that the subpacket length (ignoring the subpacket header) follows all the other GeRM headers as an eight-bit unsigned integer length field. An eight-bit length field is sufficient because there is little to be gained by multiplexing larger packets.

Any CSRC fields present in the original RTP header then follow the GeRM headers. Following this is the RTP payload.

APPLICATION SCENARIOS

The bandwidth saving due to GeRM depends on the similarity of the headers between the multiplexed packets. Consider two scenarios: arbitrary packets and packets produced by cooperating applications.

If arbitrary RTP packets are to be multiplexed, the multiplexing gain is small. If there is no correlation between the packets, all the optional fields will be present and the subpacket header will be 14 octets in length. Compared to nonmultiplexed RTP, there is still a gain here because a 14-octet subheader is smaller than the 40-octet RTP/UDP/IP header that would otherwise be present, but the bandwidth saving is relatively small compared to the saving from standard header compression.

If the packets to be multiplexed are produced by cooperating applications, the savings due to GeRM may be much greater. In the simplest case, all the packets to be multiplexed have the same payload type, length, and CSRC list; so three octets are removed in all but the first subpacket header. If the applications generating the packets cooperate, they can collude to ensure that the sequence numbers and timestamps in the subpackets match, saving an additional six octets. Even more saving can be achieved if the applications generate packets with consecutive synchronization source identifiers, allowing the SSRC to be removed also.

Of course, such collusion between implementations is stretching the bounds of what is legal RTP. In particular, an application that generates nonrandom SSRC identifiers can cause serious problems in a session with standard RTP senders. Such nonrandom SSRC use is acceptable in two scenarios:

1. When RTP and GeRM are used to convey media data between two gateways. In this case the originators and receivers of the data are blissfully unaware that RTP and GeRM have been used to transfer data. An example might be a system that generates voice-over-IP packets as part of a gateway between two PSTN exchanges.
2. When the multiplexing device remaps the SSRC before inclusion in GeRM, with the demultiplexing device regenerating the original SSRC. In this case, the SSRC identifier mapping must be signaled out of band, but that may be possible as part of the call setup procedure.

At best, GeRM can produce packets with a two-octet header per multiplexed packet, which is a significant saving compared to nonmultiplexed RTP. GeRM will always reduce the header overheads, compared to nonmultiplexed RTP.

THE FUTURE OF GERM

GeRM is not a standard protocol, and there are currently no plans to complete its specification. There are several reasons for this, primary among them being concern that the requirements for applications to collude in their production of RTP headers will limit the scope of the protocol and cause interoperability problems if GeRM is applied within a network. In addition, the bandwidth saving is relatively small unless such collusion occurs, which may make GeRM less attractive.

The concepts of GeRM are useful as an application-specific multiplex, between two gateways that source and sink multiple RTP streams using the same codec, and that are willing to collude in the generation of the RTP headers for those streams. The canonical example is IP-to-PSTN gateways, in which the IP network acts as a long-distance trunk circuit between two PSTN exchanges. GeRM allows such systems to maintain most RTP semantics, while providing a multiplex that is efficient and can be implemented solely at the application layer.

Application-Specific Multiplexing

In addition to the general-purpose multiplexing protocols such as TCRTP and GeRM, various application-specific multiplexes have been proposed. The vast majority of these multiplexes have been targeted toward IP-to-PSTN gateways, in which the IP network acts as a long-distance trunk circuit between two PSTN exchanges. These gateways have many simultaneous voice connections between them, which can be multiplexed to improve the efficiency, enabling the use of low bit-rate voice codecs, and to improve scalability.

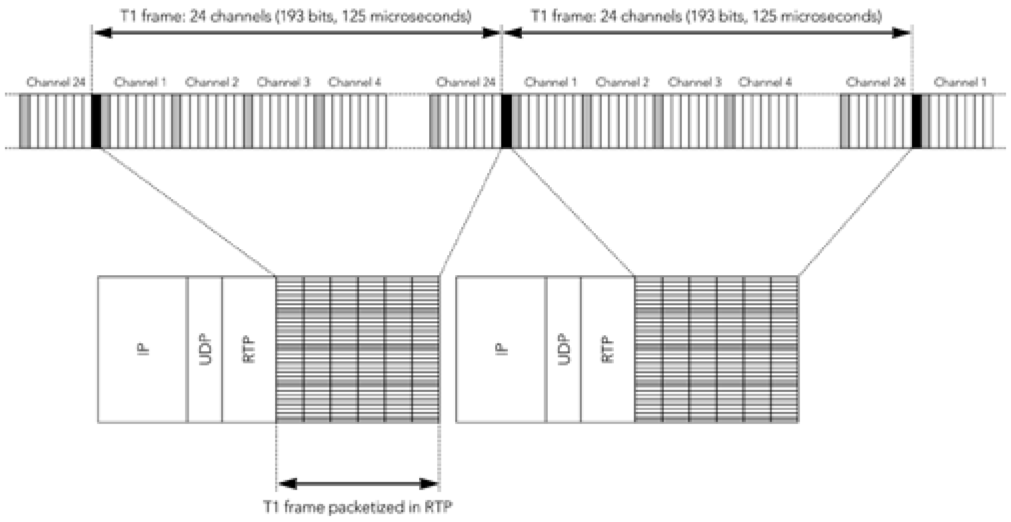
Such gateways often use a very restricted subset of the RTP protocol features. All the flows to be multiplexed commonly use the same payload format and codec, and it is likely that they do not employ silence suppression. Furthermore, each flow represents a single conversation, so there is no need for the mixer functionality of RTP. The result is that the CC, CSRC, M, P, and PT fields of the RTP header are redundant, and the sequence number and timestamp have a constant relation, allowing one of them to be elided. After these fields are removed, the only things left are the sequence number/timestamp and synchronization source (SSRC) identifier. Given such a limited use of RTP, there is a clear case for using an application-specific multiplex in these scenarios.

A telephony-specific multiplex may be defined as an operation on the RTP packets, transforming several RTP streams into a single multiplex with reduced headers. At its simplest, such a multiplex may concatenate packets with only the sequence number and a (possibly reduced) synchronization source into UDP packets, with out-of-band signaling being used to define the mapping between these reduced headers and the full RTP headers. Depending on the application, the multiplex may operate on real RTP packets, or it may be a logical operation with PSTN packets being directly converted into multiplexed packets. There are no standard solutions for such application-specific multiplexing.

As an alternative, it may be possible to define an RTP payload format for TDM (Time Division Multiplexing) payloads, which would allow direct transport of PSTN voice without first mapping it to RTP. The result is a "circuit emulation" format, defined to transport the complete circuit without caring for its contents.

In this case the RTP header will relate to the circuit. The SSRC, sequence number, and timestamp relate to the circuit, not to any of the individual conversations being carried on that circuit; the payload type identifies, for example, "T1 emulation"; the mixer functionality (CC and CSRC list) is not used, nor are the marker bit and padding. Figure 12.6 shows how the process might work, with each T1 frame forming a single RTP packet.

Figure 12.6. Voice Circuit Emulation



Of course, direct emulation of a T1 line gains little because the RTP overhead is large. However, it is entirely reasonable to include several consecutive T1 frames in each RTP packet, or to emulate a higher-rate circuit, both of which reduce the RTP overhead significantly.

The IETF has a Pseudo-Wire Edge-to-Edge Emulation working group, which is developing standards for circuit emulation, including PSTN (Public Switched Telephone Network), SONET (Synchronous Optical Network), and ATM (Asynchronous Transfer Mode) circuits. These standards are not yet complete, but an RTP payload format for circuit emulation is one of the proposed solutions.

The circuit emulation approach to IP-to-PSTN gateway design is a closer fit with the RTP philosophy than are application-specific multiplexing solutions. Circuit emulation is highly recommended as a solution for this particular application.

Summary

Multiplexing is not usually desirable. It forces all media streams to have a single transport, preventing the receiver from prioritizing them according to its needs, and making it difficult to apply error correction. In almost all cases, header compression provides a more appropriate and higher-performance solution.

Nevertheless, in some limited circumstances multiplexing can be useful, primarily when many essentially identical flows are being transported between two points, something that occurs often when voice-over-IP is being used as a backbone "trunk" to replace conventional telephone lines.

Chapter 13. Security Considerations

Until now we have considered an RTP implementation only in isolation. In the real world, however, there are those who seek to eavesdrop on sessions, impersonate legitimate users, and "hack into" systems for malicious purposes. A correctly implemented RTP system is secure against such attackers and can provide privacy, confidentiality, and authentication services to its users. This chapter describes the features of RTP that enhance user privacy, and outlines various potential attacks and how they may be prevented.

Privacy

The obvious privacy issue is how to prevent unauthorized third parties from eavesdropping on an RTP session. This issue is discussed in the next section, Confidentiality. Confidentiality is not the only privacy issue, though; users of an RTP implementation may want to limit the amount of personal information they give out during the session, or they may want to keep the identities of their communication partners secret.

Information regarding a source, possibly including personal details, is communicated in RTCP source description packets, as noted in Chapter 5, RTP Control Protocol. This information has several uses, most of which are legitimate, but some may be regarded as inappropriate by some users. An example of legitimate use might be a teleconferencing application that uses source description packets to convey the names and affiliations of participants, or to provide other caller identification information. Inappropriate use might include providing personal details via RTCP while the user is listening to an online radio station, hence allowing the station and its advertisers to track their audience. Because of these concerns, it is recommended that applications not send source description packets without first informing the user that the information is being made available.

The exception is the canonical name (CNAME), which is mandatory to send. The canonical name includes the IP address of the participant, but this should not represent a privacy issue, because the same information is available from the IP header of the packet (unless the packets pass through an RTCP-unaware Network Address Translation (NAT) device, in which case CNAME packets will expose the internal address, which some sites might want to hide). The canonical name also exposes the user name of the participant, which may be a greater privacy issue. Applications may omit or rewrite the user name to obscure this information, provided that this is done consistently by any set of applications that will associate sessions via the CNAME.

The CNAME provides a mostly unique identifier, which could be used to track participants. This issue is avoided if the participant is joining from a machine with a temporary IP address (for example, a dial-up user whose IP address is assigned dynamically). If the system has a static IP address, little can be done to protect that address from being tracked, but the CNAME provides little more information than can be obtained from the IP headers (especially if the user name portion of the CNAME is obscured).

Some receivers may not want their presence to be visible at all. It may be acceptable for those receivers not to send RTCP, although doing so prevents the sender from using the reception quality information to adapt its transmission to match the receiver (and it may cause a source to assume that the receiver has failed, and stop transmission). Furthermore, some content providers may require RTCP reports, to gauge the size of their audience.

A related privacy concern involves being able to keep the identities of the partners in a conversation secret. Even if the confidentiality measures described in the next section are used, it is possible for an eavesdropper to observe the RTP packets in transit and gain some knowledge of the communication by looking at the IP headers (for example, your boss might be interested in knowing that your voice-over-IP call is destined for an IP address owned by a recruitment agency). This problem cannot easily be solved with IP, but it can be mitigated if traffic is routed through a trusted third-party gateway that acts as an intermediary with a neutral IP address. (Traffic analysis of the gateway might still reveal communication patterns, unless the gateway is both busy and designed to withstand such analysis.)

Confidentiality

One of the key security requirements for RTP is confidentiality, ensuring that only the intended receivers can decode your RTP packets. RTP content is kept confidential by encryption, either at the application level—encrypting either the entire RTP packet or just the payload section—or at the IP layer.

Application-level encryption has advantages for RTP, but also some disadvantages. The key advantage is that it allows header compression. If only the RTP payload is encrypted, then header compression will work normally, which is essential for some applications (for example, wireless telephony using RTP). If the RTP header is encrypted too, the operation of header compression is disrupted to some extent, but it is still possible to compress the UDP/IP headers.

The other advantage of application-level encryption is that it is simple to implement and deploy, requiring no changes to host operating systems or routers. Unfortunately, this is also a potential disadvantage because it spreads the burden of correct implementation to all applications. Encryption code is nontrivial, and care must be taken to ensure that security is not compromised through poor design or flaws in implementation.

It's worth saying again: Encryption code is nontrivial, and care must be taken to ensure that security is not compromised through poor design or flaws in implementation. I strongly recommend that you study the relevant standards in detail before building a system using encryption, and make sure that you use well-known, publicly analyzed, cryptographic techniques.

Another potential disadvantage of application-level encryption is that it leaves some header fields unencrypted. In some cases, the lack of encryption might reveal sensitive information. For example, knowledge of the payload type field may allow an attacker to ascertain the values of parts of the encrypted payload data, perhaps because each frame starts with a payload header with a standard format. This should not be a problem, provided that an appropriate encryption algorithm is chosen, but it has the potential to compromise an already weak solution.

As an alternative, encryption can be performed at the IP layer—for example, using the IP security (IPsec) protocols. This approach has the advantage of being transparent to RTP, and of providing a single—presumably well-tested—suite of encryption code that can be trusted to be correct. The disadvantages of IP-layer encryption are that it disrupts the operation of RTP header compression and its deployment requires extensive changes to host operating systems.

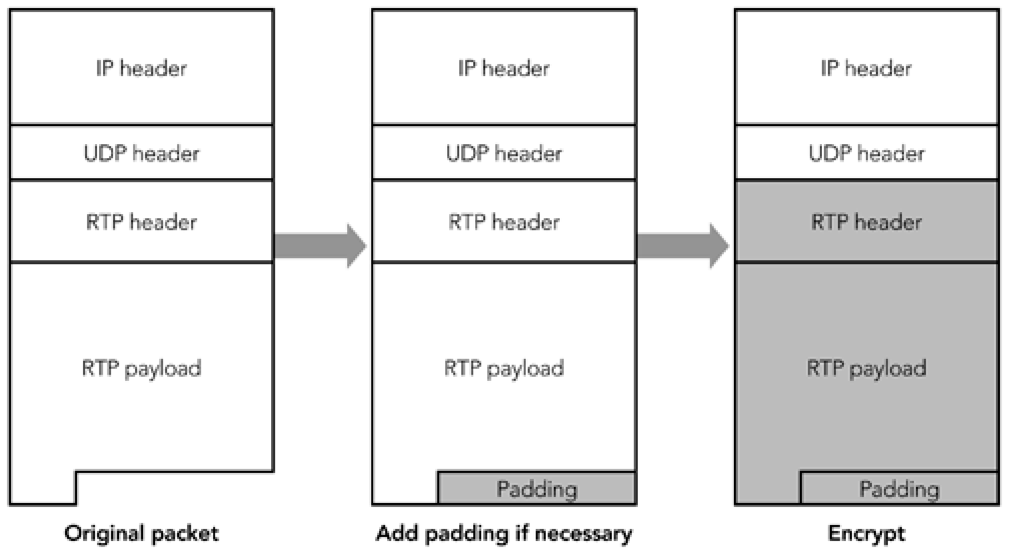
Confidentiality Features in the RTP Specification

The RTP specification provides support for encryption of both RTP data packets (including headers) and RTCP packets.

All octets of RTP data packets—including the RTP header and the payload data—are encrypted. Implementations have a choice of the encryption schemes they support. Depending on the encryption algorithm used, it may be necessary to append padding octets to the payload before encryption can be performed. For example, DES encryption[56] operates on blocks of 64 bits, so payloads will need to be padded if they are not multiples of eight octets in length. Figure 13.1 illustrates the process.

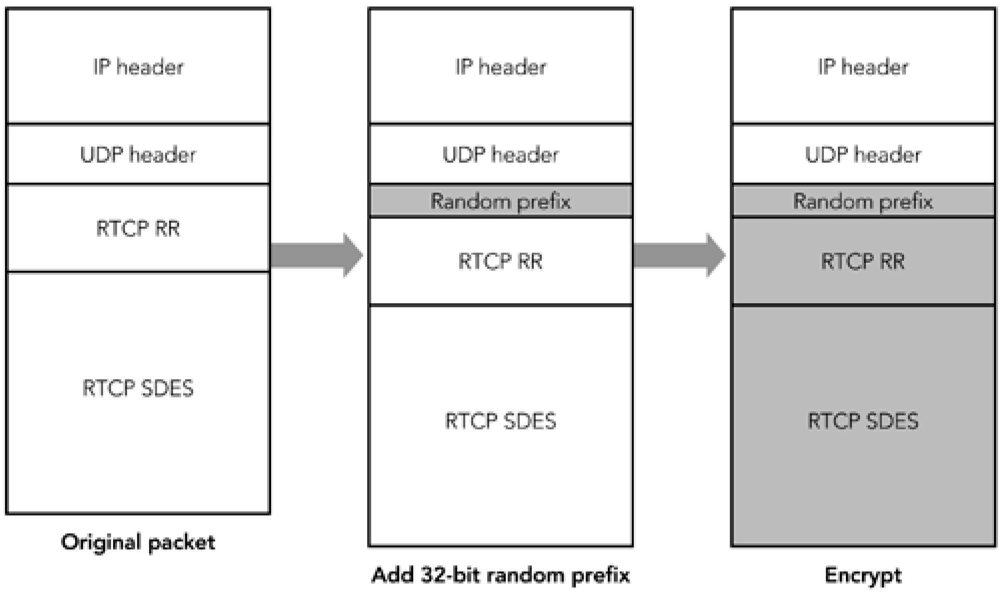
When padding is used, the P bit in the RTP header is set, and the last octet of the padding indicates the number of padding octets that have been appended to the payload.

Figure 13.1. Standard RTP Encryption of a Data Packet



When RTCP packets are encrypted, a 32-bit random number is inserted before the first packet, as shown in Figure 13.2. This is done to prevent known plain-text attacks. RTCP packets have a standard format with many fixed octets; knowledge that these fixed octets exist makes a wily cracker's work easier because he knows part of what he is looking for in a decrypted packet. The cracker could employ a brute-force key guessing, using the fixed octet values in the decryption attempt to determine when to stop.

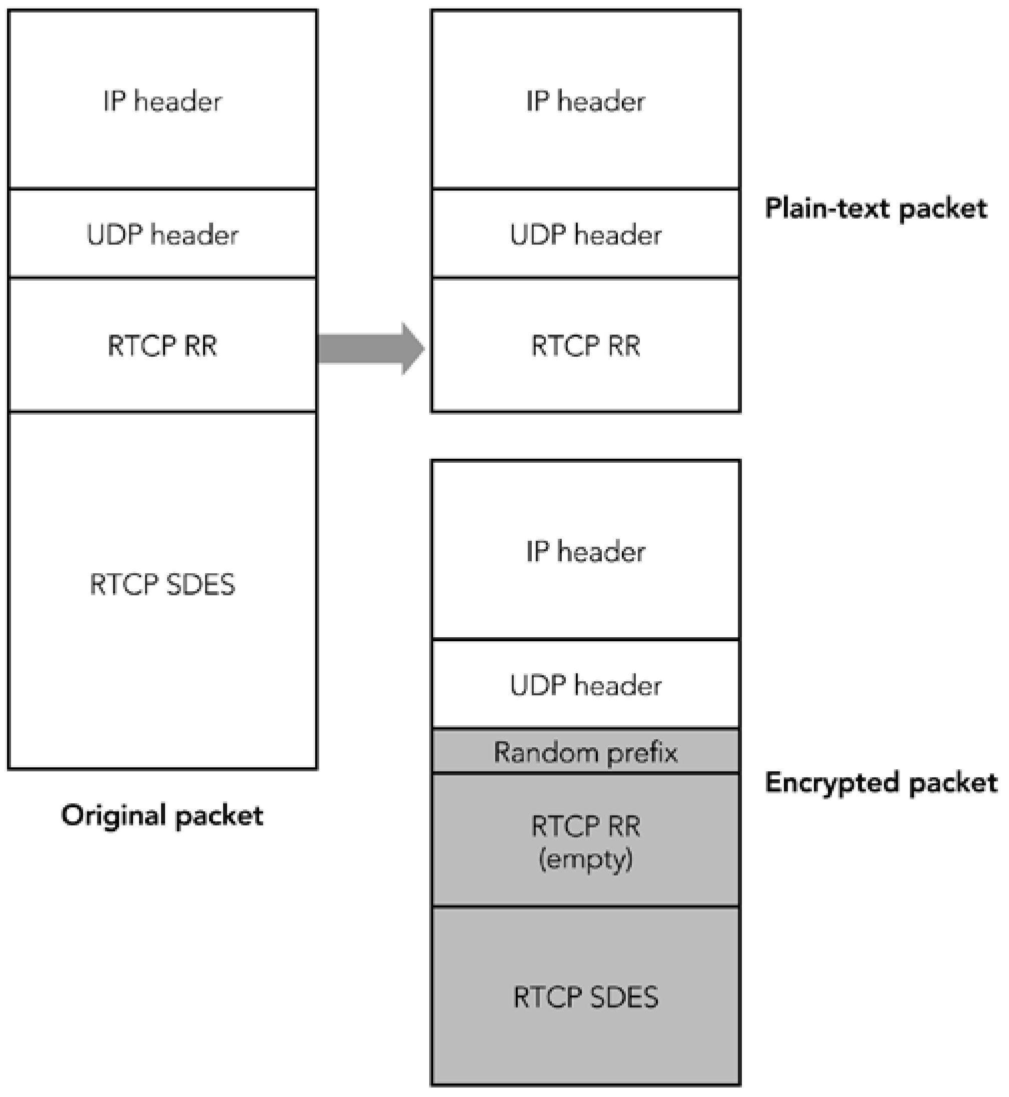
Figure 13.2. Standard RTP Encryption of a Control Packet



The insertion of the prefix provides initialization for the cipher, which effectively prevents known plain-text attacks. No prefix is used with data packets because there are fewer fixed header fields: The synchronization source is randomly chosen, and the sequence number and timestamp fields have random offsets.

In some cases it is desirable to encrypt only part of the RTCP packets while sending other parts in the clear. The typical example would be to encrypt the SDES items, but leave reception quality reports unencrypted. We can do this by splitting a compound RTCP packet into two separate compound packets. The first includes the SR/RR packets; the second includes an empty RR packet (to satisfy the rule that all compound RTCP packets start with an SR or RR) and the SDES items. (For a review of RTCP packet formats, see Chapter 5, RTP Control Protocol.) Figure 13.3 illustrates the process.

Figure 13.3. Using Standard RTP Encryption to Partially Encrypt a Control Packet



There is an exception to the rule that all compound RTCP packets must contain both an SR/RR packet and an SDES packet: When a compound packet is split into two separate compound packets for encryption, the SDES packet is included in only one compound packet, not both.

The default encryption algorithm is the Data Encryption Standard (DES) in cipher block chaining mode.[56] When RTP was designed, DES provided an appropriate level of security. However, advances in processing capacity have rendered it weak, so it is recommended that implementations choose a stronger encryption algorithm where possible. Suitable strong encryption algorithms include Triple DES[57] and the Advanced Encryption Standard (AES).[58] To maximize interoperability, all implementations that support encryption should support DES, despite its weakness.

The presence of encryption and the use of the correct key are confirmed by the receiver through header or payload validity checks, such as those described in the Packet Validation sections of Chapter 4, RTP Data Transfer Protocol, and Chapter 5, RTP Control Protocol.

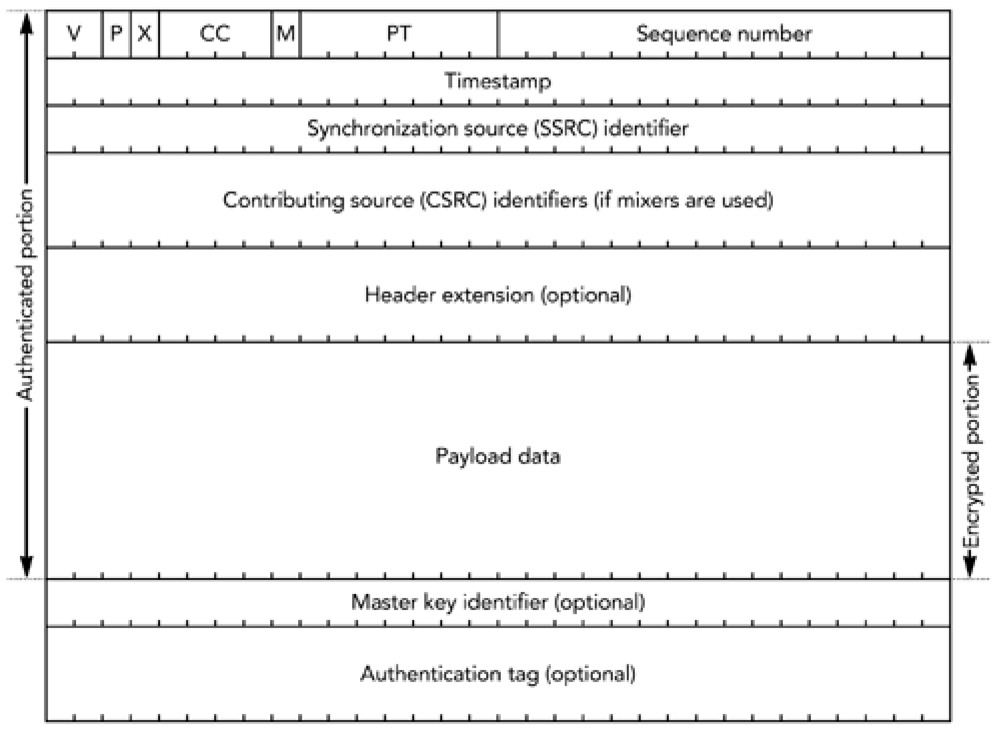
The RTP specification does not define any mechanism for the exchange of encryption keys. Nevertheless, key exchange is an essential part of any system, and it must be performed during session initiation. Call setup protocols such as SIP[28] and RTSP[14] are expected to provide key exchange, in a form suitable for RTP.

Confidentiality Using the Secure RTP Profile

An alternative to the mechanisms in the RTP specification is provided by the Secure RTP (SRTP) profile.[55] This new profile, designed with the needs of wireless telephony in mind, provides confidentiality and authentication suitable for use with links that may have relatively high loss rate, and that require header compression for efficient operation. SRTP is a work in progress, with the details of the protocol still evolving at the time of this writing. After the specification is complete, readers should consult the final standard to ensure that the details described here are still accurate.

SRTP provides confidentiality of RTP data packets by encrypting just the payload section of the packet, as shown in Figure 13.4.

Figure 13.4. Secure RTP Encryption of a Data Packet


V
P
X
CC
M
PT
Sequence number
Timestamp
Synchronization source (SSRC) identifier
Contributing source (CSRC) identifiers (if mixers are used)
Header extension (optional)
Payload data
Master key identifier (optional)
Authentication tag (optional)
Authenticated portion
Encrypted portion

The RTP header, as well as any header extension, is sent without encryption. If the RTP payload format uses a payload header within the payload section of the RTP packet, that payload header will be encrypted along with the payload data. The authentication header is described in the section titled Authentication Using the Secure RTP Profile later in this chapter. The optional master key identifier may be used by the key management protocol, for the purpose of rekeying and identifying a particular master key within the cryptographic context.

When using SRTP, the sender and receiver are required to maintain a cryptographic context, comprising the encryption algorithm, the master and salting keys, a 32-bit rollover counter (which records how many times the 16-bit RTP sequence number has wrapped around), and the session key derivation rate. The receiver is also expected to maintain a record of the sequence number of the last packet received, as well as a replay list (when using authentication). The transport address of the RTP session, together with the SSRC, is used to determine which cryptographic context is used to encrypt or decrypt each packet.

The default encryption algorithm is the Advanced Encryption Standard in either counter mode or f8 mode,[58],[59] with counter mode being mandatory to implement. Other algorithms may be defined in the future.

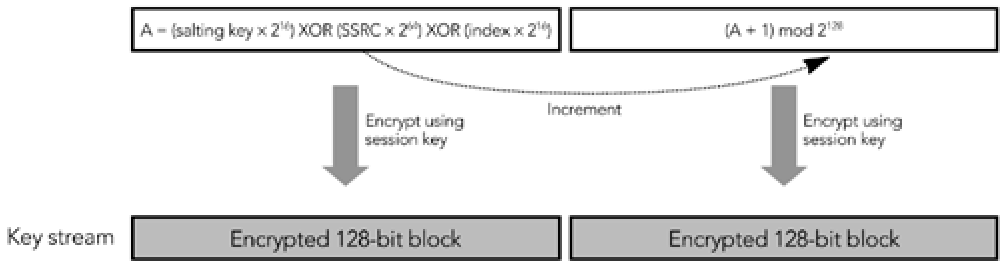
The encryption process consists of two steps:

1. The system is supplied with one or more master keys via a non-RTP-based key exchange protocol, from which ephemeral session keys are derived. Each session key is a sampling of a pseudorandom function, redrawn after a certain number of packets have been sent, with the master key, packet index, and key derivation rate as inputs. The session key can potentially change for each packet sent, depending on the key derivation rate in the cryptographic context. The master key identifier may be used by the key management protocol, to signal which of the preexchanged master keys is to be used, allowing changes in the master key to be synchronized.
2. The packet is encrypted via the generation of a key stream based on the packet index and the salting and session keys, followed by computation of the bitwise exclusive-OR (XOR) of that key stream with the payload section of the RTP packet.

In both steps, the packet index is the 32-bit extended RTP sequence number. The details of how the key stream is generated depend on the encryption algorithm and mode of operation.

If AES in counter mode is used, the key stream is generated in this way: A 128-bit integer is calculated as follows: (2^{16} x the packet index) XOR (the salting key x 2^{16}) XOR (the SSRC x 2^{64}). The integer is encrypted with the session key, resulting in the first output block of the key stream. The integer then is incremented modulo 2^{128}, and the block is again encrypted with the session key. The result is the second output block of the key stream. The process repeats until the key stream is at least as long as the payload section of the packet to be encrypted. Figure 13.5 shows this key-stream generation process.

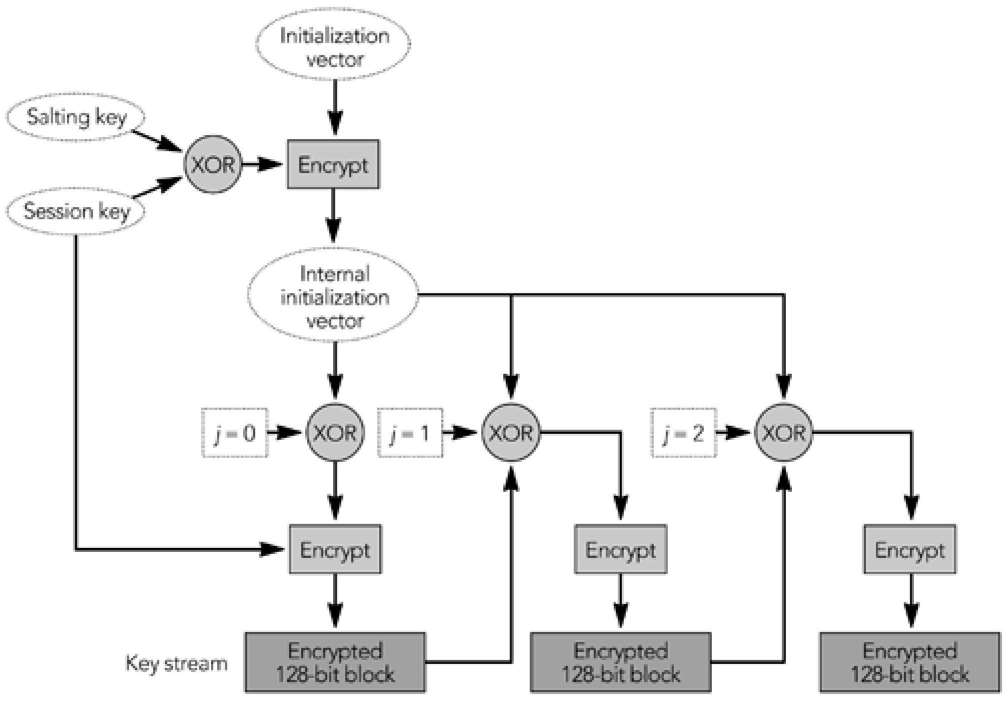
Figure 13.5. Key-Stream Generation for SRTP: AES in Counter Mode


A = (salting key × 2^16) XOR (SSRC × 2^64) XOR (index × 2^16)
(A + 1) mod 2^128
Increment
Encrypt using session key
Encrypt using session key
Key stream
Encrypted 128-bit block
Encrypted 128-bit block

When implementing AES in counter mode, you *must* ensure that each packet is encrypted with a unique key stream (the presence of the packet index and SSRC in the key stream derivation function ensures this). If you accidentally encrypt two packets using the same key stream, the encryption becomes trivial to break: You simply XOR the two packets together, and the plain text becomes available (remember from the discussion of parity FEC in Chapter 9, Error Correction, that `A XOR B XOR B = A`).

If AES in f8 mode is used, the key stream is generated in this way: The XOR of the session key and a salting key is generated, and it is used to encrypt the initialization vector. If the salting key is less than 128 bits in length, it is padded with alternating zeros and ones (0x555...) to 128 bits. The result is known as the *internal initialization vector*. The first block of the key stream is generated as the XOR of the internal initialization vector and a 128-bit variable (*j* = 0), and the result is encrypted with the session key. The variable *j* is incremented, and the second block of the key stream is generated as the XOR of the internal initialization vector, the variable *j*, and the previous block of the key stream. The process repeats, with *j* incrementing each time, until the key stream is at least as long as the payload section of the packet to be encrypted. Figure 13.6 shows this key-stream generation process.

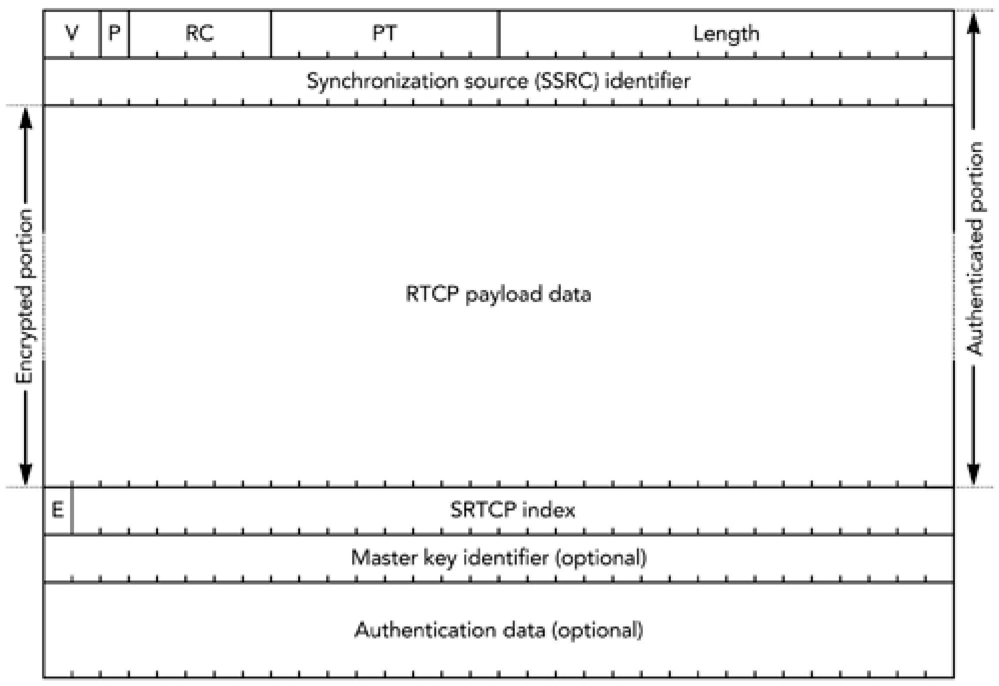
Figure 13.6. Key-Stream Generation for SRTP: AES in f8 Mode



The default encryption algorithm and mode is AES in counter mode; use of AES f8 mode can be negotiated during session initiation.

SRTP also provides confidentiality of RTP control packets. The entire RTCP packet is encrypted, excluding the initial common header (the first 64 bits of the packet) and several additional fields that are added to the end of each RTCP packet, as shown in Figure 13.7. The additional fields are an SRTCP (Secure RTCP) index, a bit to indicate if the payload is encrypted (the E bit), an optional master key identifier, and an authentication header (described in the section titled Authentication Using the Secure RTP Profile later in this chapter).

Figure 13.7. Secure RTP Encryption of a Control Packet


V
P
RC
PT
Length
Synchronization source (SSRC) identifier
RTCP payload data
Encrypted portion
Authenticated portion
E
SRTCP index
Master key identifier (optional)
Authentication data (optional)

Encryption of RTCP packets proceeds in much the same way as encryption of RTP data packets does, but using the SRTCP index in place of the extended RTP sequence number.

The encryption prefix applied during standard RTCP encryption is not used with SRTP (the differences in encryption algorithm mean that the prefix offers no benefit). It is legal to split RTCP packets into encrypted and unencrypted packets, as can be done with standard RTCP encryption, indicated by the E bit in the SRTCP packet.

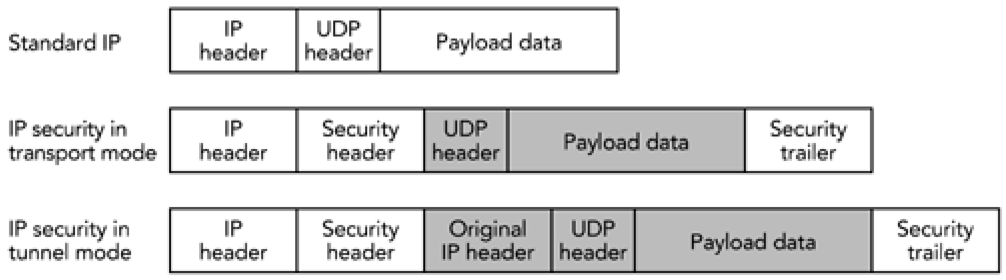
As with the RTP specification, the SRTP profile defines no mechanism for exchange of encryption keys. Keys must be exchanged via non-RTP means—for example, within SIP or RTSP. The master key identifier may be used to synchronize changes of master keys.

Confidentiality Using IP Security

In addition to the application-level security provided by standard RTP and Secure RTP, it is possible to use IP-level security[17], [110] with RTP. IPsec is implemented as part of the operating system network stack or in a gateway, not by applications. It provides security for all communications from a host, and it is not RTP-specific.

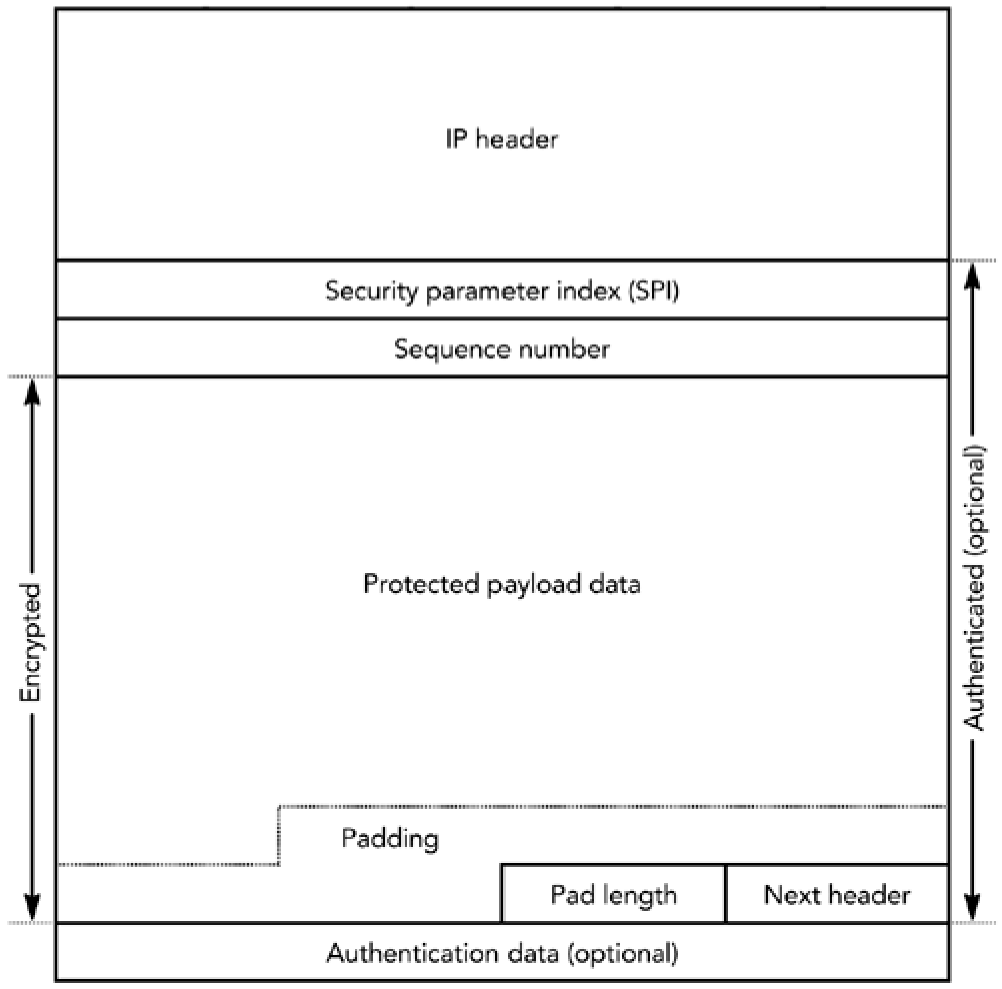
IP security (IPsec) has two modes of operation: transport mode and tunnel mode. *Transport mode* inserts an additional header between IP and the transport header, providing confidentiality of the TCP or UDP header and payload, but leaving the IP header untouched. *Tunnel mode* encapsulates the entire IP datagram inside a security header. Figure 13.8 illustrates the differences between the two modes of operation. IP security in tunnel mode is most commonly used to build virtual private networks, tunneling between two gateway routers to securely extend an intranet across the public Internet. Transport mode is used when end-to-end security between individual hosts is desired.

Figure 13.8. Transport Mode versus Tunnel Mode IPsec (Shaded Data Is Protected)



Both tunnel mode and transport mode support confidentiality and authentication of packets. Confidentiality is provided by a protocol known as the Encapsulating Security Payload (ESP).[21] ESP comprises an additional header and trailer added to each packet. The header includes a security parameter index and sequence number; the trailer contains padding and an indication of the type of the encapsulated data (TCP or UDP if transport mode is used, IP-in-IP if tunnel mode is used). Encapsulated between the header and trailer is the protected data, including the remaining headers. Figure 13.9 shows the encapsulation process, header, and trailer.

Figure 13.9. An Encapsulating Security Payload Packet



The security parameter index (SPI) and sequence number are 32-bit fields. The SPI is used to select the cryptographic context, and hence the decryption key to be used. The sequence number increments by one with each packet sent and is used to provide replay protection (see the section titled Replay Protection later in this chapter). Following these two header fields is the encapsulated payload: a UDP header followed by an RTP header and payload if transport mode is used; IP/UDP/RTP headers and payload if tunnel mode is used.

Following the payload data is padding, if required, and a "next header" field. This last field determines the type of the encapsulated header. Its name is somewhat misleading, given that the header to which this field refers is actually sent earlier in the packet. Finally, optional authentication data completes the packet (see the section titled Authentication Using IP Security later in this chapter).

ESP encrypts the protected data section of the packet, using a symmetric algorithm (DES is mandatory to implement; other algorithms may be negotiated). If ESP is used with RTP, the entire RTP header and payload will be encrypted, along with the UDP headers—and IP headers if tunnel mode is used.

It is not possible to use header compression with IP security in transport mode. If tunnel mode is used, the inner IP/UDP/RTP headers may be compressed before encryption and encapsulation. Doing so largely removes the bandwidth penalty due to the IPsec headers, but it does not achieve the efficiency gains expected of header compression. If bandwidth efficiency is a goal, application-level RTP encryption should be used.

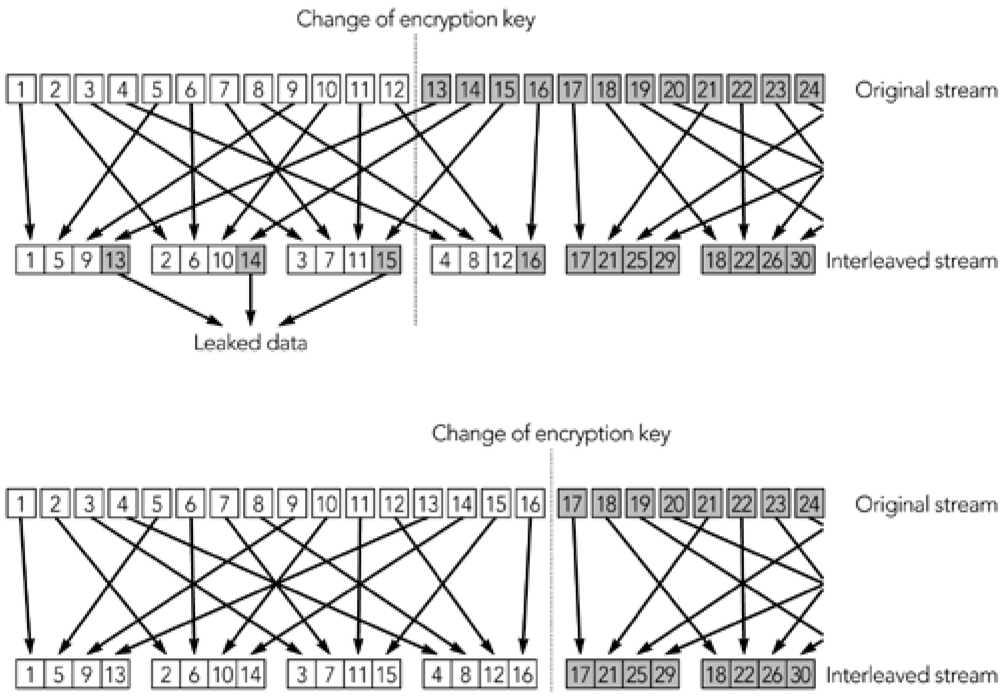
IP security may also cause difficulty with some firewalls and Network Address Translation (NAT) devices. In particular, IP security hides the TCP or UDP headers, replacing them with an ESP header. Firewalls are typically configured to block all unrecognized traffic, in many cases including IPsec (the firewall has to be configured to allow ESP [IP protocol 50], in addition to TCP and UDP). Related problems occur with NAT because translation of TCP or UDP port numbers is impossible if they are encrypted in an ESP packet. If firewalls and NAT boxes are present, application-level RTP encryption may be more successful.

The IP security protocol suite includes an extensive signaling protocol, the Internet Key Exchange (IKE), used to set up the necessary parameters and encryption keys. The details of IKE are beyond the scope of this book.

Other Considerations

Several RTP payload formats provide coupling between packets—for example, when interleaving or forward error correction is being used. Coupling between packets may affect the operation of encryption, restricting the times when it is possible to change the encryption key. Figure 13.10 shows an example of interleaving that illustrates this problem.

Figure 13.10. Interactions between Encryption and Interleaving



The confidentiality mechanisms available for RTP can use a range of encryption algorithms, but they define a "must implement" algorithm to ensure interoperability. In many cases the mandatory algorithm is the Data Encryption Standard (DES). Advances in computational power have made DES seem relatively weak—recent systems have demonstrated brute-force cracking of DES in less than 24 hours—so it is appropriate to negotiate a stronger algorithm if possible. Triple DES[57] and the recently announced Advanced Encryption Standard (AES)[58] are suitable possibilities.

The use of end-to-end encryption to ensure confidentiality in RTP is effective at preventing unauthorized access to content, whether that content is pay-per-view TV or private telephone conversations. This protection is generally desirable, but it does have some wider implications. In particular, there is an ongoing debate in some jurisdictions regarding the ability of law enforcement officials to wiretap communications. Widespread use of encryption as a confidentiality measure in RTP makes such wiretaps—along with other forms of eavesdropping—more difficult. In addition, some jurisdictions restrict the use, or distribution, of products that include encryption. You should understand the legal and regulatory issues regarding use of encryption in your jurisdiction before implementing the confidentiality measures described in this chapter.

Authentication

There are two types of authentication: proof that the packets have not been tampered with, known as *integrity protection*, and proof that the packets came from the correct source, known as *source origin authentication*.

Integrity protection is achieved through the use of message authentication codes. These codes take a packet to be protected, and a key known only to the sender and receivers, and use these to generate a unique signature. Provided that the key is not known to an attacker, it is impossible to change the contents of the packet without causing a mismatch between the packet contents and the message authentication code. The use of a symmetric shared secret limits the capability to authenticate the source in a multiparty group: All members of the group are able to generate authenticated packets.

Source origin authentication is a much harder problem for RTP applications. It might first be thought equivalent to the problem of generating message authentication codes, but it is more difficult because a shared secret between sender and receiver is not sufficient. Rather, it is necessary to identify the sender in the signature, meaning that the signature is larger and more expensive to compute (in much the way public key cryptography is more expensive than symmetric cryptography). This requirement often makes it infeasible to authenticate the source of each packet in an RTP stream.

Like confidentiality, authentication can be applied at either the application level or the IP level, with much the same set of advantages and disadvantages. Both alternatives have been developed for use with RTP.

Authentication Using Standard RTP

Standard RTP provides no support for integrity protection or source origin authentication. Implementations that require authentication should either implement secure RTP or use a lower-layer authentication service such as that provided by the IP security extensions.

Authentication Using the Secure RTP Profile

SRTP supports both message integrity protection and source origin authentication. For integrity protection, a message authentication tag is appended to the end of the packet, as was shown in Figure 13.4. The message authentication tag is calculated over the entire RTP packet and is computed after the packet has been encrypted. At the time of this writing, the HMAC-SHA-1 integrity protection algorithm is specified for use with SRTP. Other integrity protection algorithms may be defined in the future, and negotiated for use with SRTP.

Source origin authentication using the TESLA (Timed Efficient Stream Loss-tolerant Authentication) authentication algorithm has been considered for use with SRTP, but TESLA is not fully defined at the time of this writing. You are advised to consult the SRTP specification, when it is completed, for details.

The authentication mechanisms of SRTP are not mandatory, but I *strongly* recommend that all implementations use them. In particular, you should be aware that it is trivially possible for an attacker to forge data encrypted using AES in counter mode unless authentication is used. The secure RTP profile specification describes this issue in detail.

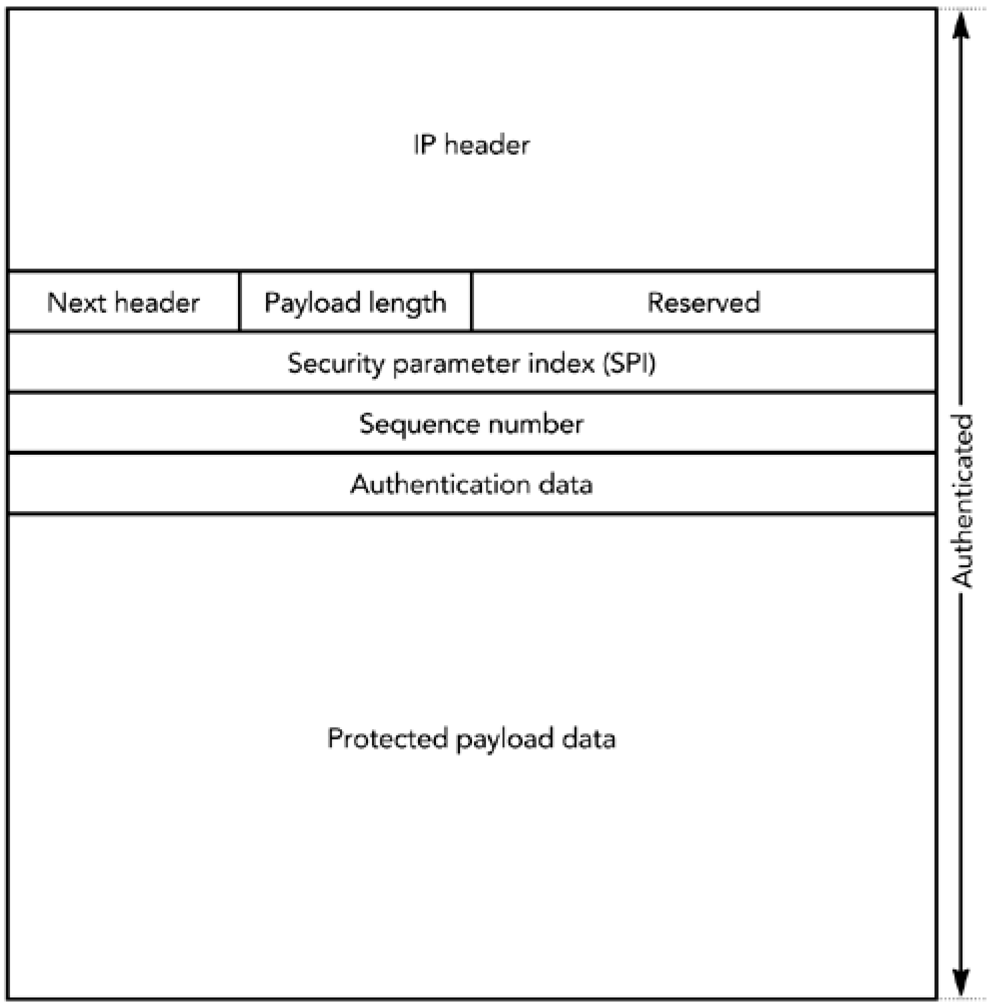
Authentication Using IP Security

The IP security extensions can provide integrity protection and authentication for all packets sent from a host. There are two ways this can be done: as part of the encapsulating security payload, or as an authentication header.

The Encapsulating Security Payload was described earlier, in the section titled Confidentiality Using IP Security. As shown in Figure 13.9, ESP includes an optional authentication data section as part of the trailer. If present, the authentication provides a check on the entire encapsulated payload, plus the ESP header and trailer. The outer IP header is not protected.

An alternative to ESP is the Authentication Header (AH) defined in RFC 2402[18] and shown in Figure 13.11. The fields in this header are much like their counterparts in ESP, and the AH can be used in both tunnel mode and transport mode. The key difference is that the entire packet is authenticated, including the outer IP header.

Figure 13.11. The IPsec Authentication Header



If the requirement is to provide confidentiality as well as authentication, then ESP is appropriate (using ESP to encrypt and AH to authenticate would lead to excessive bandwidth overheads). If the only requirement is authentication—not confidentiality—then the choice between ESP and AH depends on whether it is considered necessary to authenticate the outer IP header. Authentication of the outer header provides some additional security, by ensuring that the source IP address is not spoofed. It also has the side effect of rendering Network Address Translation impossible.

A range of authentication algorithms may be used with IP security. If authentication is used, the mandatory algorithms for both ESP and AH are HMAC-MD5-96 and HMAC-SHA-96,[19], [20] which provide integrity protection only. Algorithms for source origin authentication may be defined in the future.

The earlier section titled Confidentiality Using IP Security noted potential interactions between IP security and header compression, firewalls, and NAT boxes. These issues also apply when IP security is used to provide authentication.

The IP security protocol suite includes an extensive signaling protocol, the Internet Key Exchange (IKE), which is used to set up the necessary parameters and authentication keys. The details of IKE are beyond the scope of this book.

Replay Protection

Related to authentication is the issue of replay protection: stopping an attacker from recording the packets of an RTP session and reinjecting them into the network later for malicious purposes. The RTP timestamp and sequence number provide limited replay protection because implementations are supposed to discard old data. However, an attacker can observe the packet stream and modify the recorded packets before playback such that they match the expected timestamp and sequence number of the receiver.

To provide replay protection, it is necessary to authenticate messages for integrity protection. Doing so stops an attacker from changing the sequence number of replayed packets, making it impossible for old packets to be replayed into a session.

Denial of Service

A potential denial-of-service threat exists with payload formats using compression that has nonuniform receiver-end computational load. If the payload format has such properties, an attacker might be able to inject pathological packets into a media stream, which are complex to decode and cause the receiver to be overloaded. There is little a receiver can do to avoid this problem, short of stopping processing packets in overload situations.

Another problem that can cause denial of service is failure to implement the RTCP timing rules correctly. If participants send RTCP at a constant rate rather than increasing the interval between packets as the size of the group increases, the RTCP traffic will grow linearly with the size of the group. For large groups, this growth can result in significant network congestion. The solution is to pay close attention to the correct implementation of RTCP.

Similarly, failure to implement the RTCP reconsideration algorithms can result in transient congestion when rapid changes in membership occur. This is a secondary concern because the effect is only transitory, but it may still be an issue.

Network congestion can also occur if RTP implementations respond inappropriately to packet loss. Chapter 10, Congestion Control, describes the issues in detail; for our purposes here, suffice it to say that congestion can cause significant denial of service.

Mixers and Translators

It is not possible to use an RTP mixer to combine encrypted media streams unless the mixer is trusted and has access to the encryption key. This requirement typically prevents mixers from being used if the session needs to be kept confidential.

Translation is also difficult, although not necessarily impossible, if the media streams are encrypted. Translation that involves recoding of the media stream is typically impossible unless the translator is trusted. Some translations, however, do not require recoding of the media stream—for example, translation between IPv4 and IPv6 transport—and these may be provided without affecting the encryption, and without having to trust the translator.

Source origin authentication is similarly difficult if the mixer or translator modifies the media stream. Again, if the mixer or translator is trusted, it may re-sign the modified media stream; otherwise the source cannot be authenticated.

Active Content

Most audiovisual content is passive: It comprises encoded data that is passed through a static decryption algorithm to produce the original content. With the exception of the issues due to nonuniform processing requirements noted earlier, such content is not dangerous.

There is, however, another class of content: that which contains executable code—for example, Java applets or ECMAScript, which can be included with MPEG-4 streams. Such *active content* has the potential to execute arbitrary operations on the receiving system, unless carefully restricted.

Other Considerations

The text of SDES items is not null-terminated, and manipulating SDES items in languages that assume null-terminated strings requires care. This is a particular problem with C-based implementations, which must take care to ensure that they use lengthchecking string manipulation functions—for example, `strncpy()` rather than `strcpy()`. Careless implementations may be vulnerable to buffer overflow attacks.

The text of SDES items is entered by the user, and thus it cannot be trusted to have safe values. In particular, it may contain metacharacters that have undesirable side effects. For example, some user interface scripting languages allow command substitution to be triggered by metacharacters, potentially giving an attacker the means to execute arbitrary code.

Implementations should not assume that packets are well formed. For example, it might be possible for an attacker to produce packets whose actual length does not correspond to the expected length. Again, there is a potential for buffer overflow attacks against careless implementations.

Summary

This chapter has outlined the features of RTP that provide privacy and confidentiality of communication, as well as authentication of participants and the media stream. It has also described various potential attacks on a system, and how they can be prevented.

These are not the only issues that a secure implementation needs to consider, though; it is also necessary to secure the session initiation and control protocols. This chapter has highlighted some issues relating to the security of these initiation and control protocols, but a full treatment of this topic is outside the scope of this book.

References

This book includes various references to Internet RFC (Request for Comments) standards and informational documents. These can be retrieved directly from the RFC Editor (at http://www.rfc-editor.org) or from the Internet Engineering Task Force (IETF, at http://www.ietf.org). RFC documents do not change after they've been published, but they can be made obsolete by new standards. The RFC Editor maintains a list giving the current status of each RFC (http://www.rfc-editor.org/rfc-index.html); it is advisable to check that an RFC is still current before beginning implementation (for example, the RTP specification RFC 1889 is under revision, and the revision will result in a new RFC that makes RFC 1889 obsolete).

In some cases it has been necessary to reference work in progress, in the form of Internet-Drafts. These are the working documents of the IETF; some are eventually published as RFCs, and others are discarded. The Internet-Drafts referenced here are expected to be available in RFC form shortly. The RFC Editor Web site has a search function that allows you to find the RFC form of these documents when published. The IETF maintains an archive of Internet-Drafts online for up to six months after their submission; at that time they must be published as an RFC, revised, or discarded.

It is important to remember that Internet-Drafts are not reference material and may change at any time. Each draft carries the following note as a reminder:

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as work in progress.

Implementing from an Internet-Draft is not wise, unless you are willing to face the possibility that the draft will change while you are writing code. In particular, it is not appropriate to ship products based on Internet-Drafts, because they may be made obsolete by the final standard.

IETF RFC Standards

1. D. Cohen. "Specifications for the Network Voice Protocol," Network Working Group, RFC 741, November 1977.

2. D. Crocker. "Standard for the Format of ARPA Internet Text Messages," Network Working Group, RFC 822, August 1982.

3. J. Nagle. "Congestion Control in IP/TCP Internetworks," Network Working Group, RFC 896, January 1984.

4. V. Jacobson. "Compressing TCP/IP Headers for Low-Speed Serial Links," Internet Engineering Task Force, RFC 1144, February 1990.

5. D. L. Mills. "Network Time Protocol (Version 3): Specification, Implementation and Analysis," Internet Engineering Task Force, RFC 1305, March 1992.

6. H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. "RTP: A Transport Protocol for Real-Time Applications," Internet Engineering Task Force, RFC 1889, January 1996.

7. H. Schulzrinne. "RTP Profile for Audio and Video Conferences with Minimal Control," Internet Engineering Task Force, RFC 1890, January 1996.

8. S. Bradner. "The Internet Standards Process—Revision 3," Internet Engineering Task Force, RFC 2026, October 1996.

9. T. Turletti and C. Huitama. "RTP Payload Format for H.261 Video Streams," Internet Engineering Task Force, RFC 2032, October 1996.

10. C. Perkins, I. Kouvelas, O. Hodson, V. Hardman, M. Handley, J. C. Bolot, A. Vega-Garcia, and S. Fosse-Parisis. "RTP Payload for Redundant Audio Data," Internet Engineering Task Force, RFC 2198, September 1997.

11. B. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. "Resource ReSerVation Protocol (RSVP)—Version 1 Functional Specification," Internet Engineering Task Force, RFC 2205, September 1997.

12. D. Hoffman, G. Fernando, V. Goyal, and M. Civanlar. "RTP Payload Format for MPEG1/MPEG2 Video," Internet Engineering Task Force, RFC 2250, January 1998.

13. F. Yergeau. "UTF-8, a Transformation Format of ISO 10646," Internet Engineering Task Force, RFC 2279, January 1998.

14. H. Schulzrinne, A. Rao, and R. Lanphier. "Real Time Streaming Protocol (RTSP)," Internet Engineering Task Force, RFC 2326, April 1998.

15. M. Handley and V. Jacobson. "SDP: Session Description Protocol," Internet Engineering Task Force, RFC 2327, April 1998.

16. R. Hinden and S. Deering. "IP Version 6 Addressing Architecture," Internet Engineering Task Force, RFC 2373, July 1998.

17. S. Kent and R. Atkinson. "Security Architecture for the Internet Protocol," Internet Engineering Task Force, RFC 2401, November 1998.

18. S. Kent and R. Atkinson. "IP Authentication Header," Internet Engineering Task Force, RFC 2402, November 1998.

19. C. Madson and R. Glenn. "The Use of HMAC-MD5-96 Within ESP and AH," Internet Engineering Task Force, RFC 2403, November 1998.

20. C. Madson and R. Glenn. "The Use of HMAC-SHA-1-96 Within ESP and AH," Internet Engineering Task Force, RFC 2404, November 1998.

21. S. Kent and R. Atkinson. "IP Encapsulating Security Payload (ESP)," Internet Engineering Task Force, RFC 2406, November 1998.

22. C. Bormann, L. Cline, G. Deisher, T. Gardos, C. Maciocco, D. Newell, J. Ott, G. Sullivan, S. Wenger, and C. Zhu. "RTP Payload Format for the 1998 Version of ITU-T Rec. H.263 Video (H.263+)," Internet Engineering Task Force, RFC 2429, October 1998.

23. K. Nichols, S. Blake, F. Baker, and D. Black. "Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers," Internet Engineering Task Force, RFC 2474, December 1998.

24. S. Blake, D. Black, M. Carlson, E. Davis, Z. Wang, and W. Weiss. "An Architecture for Differentiated Services," Internet Engineering Task Force, RFC 2475, December 1998.

25. M. Degermark, B. Nordgren, and S. Pink. "IP Header Compression," Internet Engineering Task Force, RFC 2507, February 1999.

26. S. Casner and V. Jacobson. "Compressing IP/UDP/RTP Headers for Low-Speed Serial Links," Internet Engineering Task Force, RFC 2508, February 1999.

27. M. Engan, S. Casner, and C. Bormann. "IP Header Compression over PPP," Internet Engineering Task Force, RFC 2509, February 1999.

28. J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, R. Sparks, M. Handley, and E. Schooler. "SIP: Session Initiation Protocol," Internet Engineering Task Force, RFC 3261, June 2002.

29. M. Allman, V. Paxson, and W. Stevens. "TCP Congestion Control," Internet Engineering Task Force, RFC 2581, April 1999.

30. K. McKay. "RTP Payload Format for PureVoice Audio," Internet Engineering Task Force, RFC 2658, August 1999.

31. W. Townsley, A. Valencia, A. Rubens, G. Pall, G. Zorn, and B. Palter. "Layer Two Tunneling Protocol (L2TP)," Internet Engineering Task Force, RFC 2661, August 1999.

32. J. Rosenberg and H. Schulzrinne. "An RTP Payload Format for Generic Forward Error Correction," Internet Engineering Task Force, RFC 2733, December 1999.

33. M. Handley and C. Perkins. "Guidelines for Writers of RTP Payload Format Specifications," Internet Engineering Task Force, RFC 2736, December 1999.

34. H. Schulzrinne and S. Petrack. "RTP Payload for DTMF Digits, Telephony Tones and Telephony Signals," Internet Engineering Task Force, RFC 2833, May 2000.

35. M. Handley, C. Perkins, and E. Whelan. "Session Announcement Protocol," Internet Engineering Task Force, RFC 2974, October 2000.

36. P. Luthi. "RTP Payload Format for ITU-T Recommendation G.722.1," Internet Engineering Task Force, RFC 3047, January 2001.

37. C. Bormann (Editor). "Robust Header Compression (ROHC): Framework and Four Profiles: RTP, UDP, ESP and Uncompressed," Internet Engineering Task Force, RFC 3095, July 2001.

38. R. Finlayson. "A More Loss-Tolerant RTP Payload Format for MP3 Audio," Internet Engineering Task Force, RFC 3119, June 2001.

39. R. Pazhyannur, I. Ali, and C. Fox. "PPP Multiplexed Frame Option," Internet Engineering Task Force, RFC 3153, August 2001.

40. C. Perkins, J. Rosenberg, and H. Schulzrinne. "RTP Testing Strategies," Internet Engineering Task Force, RFC 3158, August 2001.

41. J. Sjoberg, M. Westerlund, A. Lakaniemi, and Q. Xie. "Real-Time Transport Protocol (RTP) Payload Format and File Storage Format for the Adaptive Multi-Rate (AMR) and Adaptive Multi-Rate Wideband (AMR-WB) Audio Codecs," Internet Engineering Task Force, RFC 3267, June 2002.

42. R. Zopf. "Real-time Transport Protocol (RTP) Payload for Comfort Noise (CN)," Internet Engineering Task Force, RFC 3389, September 2002.

IETF Internet-Drafts

43. T. Koren, S. Casner, J. Geevarghese, B. Thompson, and P. Ruddy. "Enhanced Compressed RTP (CRTP) for Links with High Delay, Packet Loss and Reordering," Internet Engineering Task Force, Work in Progress (Update to RFC 2508), February 2003.

44. J. Ott, S. Wenger, N. Sato, C. Burmeister, and J. Rey. "Extended RTP Profile for RTCP-Based Feedback (RTP/AVPF)," Internet Engineering Task Force, Work in Progress, February 2003.

45. M. Handley. "GeRM: Generic RTP Multiplexing," Internet Engineering Task Force, Work in Progress, November 1998.

46. A. Valencia and T. Koren. "L2TP Header Compression ("L2TPHC")," Internet Engineering Task Force, Work in Progress, November 2002.

47. A. Li, F. Liu, J. Villasenor, J-H. Park, D-S. Park, Y. L. Lee, J. Rosen-berg, and H. Schulzrinne. "An RTP Payload Format for Generic FEC with Uneven Level Protection," Internet Engineering Task Force, Work in Progress, November 2002.

48. G. Liebl, M. Wagner, J. Pandel, and W. Weng. "An RTP Payload Format for Erasure-Resilient Transmission of Progressive Multimedia Streams," Internet Engineering Task Force, Work in Progress, March 2003.

49. H. Schulzrinne and S. Casner. "RTP Profile for Audio and Video Conferences with Minimal Control," Internet Engineering Task Force, Work in Progress (Update to RFC 1890), March 2003.

50. H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. "RTP: A Transport Protocol for Real-Time Applications," Internet Engineering Task Force, Work in Progress (Update to RFC 1889), March 2003.

51. S. Casner and P. Hoschka. "MIME Type Registration of RTP Payload Formats," Internet Engineering Task Force, Work in Progress, November 2001.

52. B. Thompson, T. Koren, and D. Wing. "Tunneling Multiplexed Compressed RTP ('TCRTP')," Internet Engineering Task Force, Work in Progress, November 2002.

53. L-A. Larzon, M. Degermark, S. Pink, L-E. Jonsson (Editor), and G. Fairhurst (Editor). . "The UDP Lite Protocol," Internet Engineering Task Force, Work in Progress, December 2002.

54. S. Bhattacharyya, C. Diot, L. Giuliano, R. Rockell, J. Meylor, D. Meyer, G. Shepherd, and B. Haberman. "An Overview of Source-Specific Multicast (SSM)," Internet Engineering Task Force, Work in Progress, November 2002.

55. M. Baugher, D. McGrew, D. Oran, R. Blom, E. Carrara, M. Naslund, and K. Norrman. "The Secure Real Time Transport Protocol," Internet Engineering Task Force, Work in Progress, June 2002.

Other Standards

56. NIST. "Data Encryption Standard (DES)," Federal Information Processing Standard, FIPS 46-2, December 1993.

57. NIST. "Data Encryption Standard (DES)," Federal Information Processing Standard, FIPS 46-3, October 1999.

58. NIST. "Advanced Encryption Standard (AES)," Federal Information Processing Standard, FIPS-197, 2001 (http://www.nist.gov/aes).

59. ETSI SAGE 3GPP Standard Algorithms Task Force. "Security Algorithms Group of Experts (SAGE); General Report on the Design, Specification and Evaluation of 3GPP Standard Confidentiality and Integrity Algorithms," Public Report, Draft Version 1.0, December 1999.

60. ETSI Recommendation GSM 6.11 "Substitution and Muting of Lost Frames for Full Rate Speech Channels," 1992.

61. ITU-T Recommendation G.114. "One-way Transmission Time," May 2000.

62. ITU-T Recommendation H.323. "Packet-Based Multimedia Communications Systems," November 2000.

63. ITU-T Recommendation P.861. "Objective Quality Measurement of Telephone-Band (300-3400 Hz) Speech Codecs," February 1998.

64. ITU-T Recommendation P.862. "Perceptual Evaluation of Speech Quality (PESQ), an Objective Method for End-to-End Speech Quality Assessment of Narrowband Telephone Networks and Speech Codecs," January 2001.

Conference and Journal Papers

65. D. D. Clark and D. L. Tennenhouse. "Architectural Considerations for a New Generation of Protocols," Proceedings of the SIGCOMM Symposium on Communications Architectures and Protocols, *Computer Communications Review*, Volume 20, Number 4, September 1990.

66. J. C. Bolot and A. Vega Garcia. "The Case for FEC-Based Error Control for Packet Audio in the Internet," to appear in *ACM Multimedia Systems*.

67. J. C. Bolot and A. Vega Garcia. "Control Mechanisms for Packet Audio in the Internet," Proceedings of IEEE Infocom '96, San Francisco, CA, March 1996.

68. D. Chiu and R. Jain. "Analysis of the Increase/Decrease Algorithms for Congestion Avoidance in Computer Networks," *Journal of Computer Networks and ISDN Systems*, Volume 17, Number 1, June 1989.

69. B. J. Dempsey, J. Liebeherr, and A. C. Weaver. "On Retransmission-Based Error Control for Continuous Media Traffic in Packet Switched Networks," *Computer Networks and ISDN Systems*, Volume 28, 1996, pages 719–736.

70. J. H. Saltzer, D. P. Reed, and D. D. Clark. "End-to-End Arguments in System Design," *ACM Transactions on Computer Systems*, Volume 2, Number 4, November 1984.

71. S. Floyd and V. Paxson. "Difficulties in Simulating the Internet," *IEEE/ACM Transactions on Networking*, Volume 9, Number 4, August 2001. An earlier version appeared in the Proceedings of the 1997 Winter Simulation Conference, Atlanta, GA, December 1997.

72. S. Floyd, M. Handley, J. Padhye, and J. Widmer. "Equation-Based Congestion Control for Unicast Applications," Proceedings of ACM SIGCOMM 2000, Stockholm, August 2000.

73. S. Floyd and K. Fall. "Promoting the Use of End-to-End Congestion Control in the Internet," *IEEE/ACM Transactions on Networking*, Volume 7, Number 4, August 1999.

74. D. J. Goodman, G. B. Lockhart, and O. J. Wasem. "Waveform Substitution Techniques for Recovering Missing Speech Segments in Packet Voice Communications," *IEEE Transactions on Acoustics, Speech and Signal Processing*, Volume 34, Number 6, December 1986.

75. J. G. Gruber and L. Strawczynski. "Subjective Effects of Variable Delay and Speech Clipping in Dynamically Managed Voice Systems," *IEEE Transactions on Communications*, Volume 33, Number 8, August 1985.

76. M. Handley, J. Crowcroft, C. Bormann, and J. Ott. "Very Large Conferences on the Internet: The Internet Multimedia Conferencing Architecture," *Journal of Computer Networks and ISDN Systems*, Volume 31, Number 3, February 1999.

77. V. Hardman, M. A. Sasse, M. Handley, and A. Watson. "Reliable Audio for Use over the Internet," Proceedings of INET '95, Honolulu, HI, June 1995.

78. M. Handley, J. Padhye, S. Floyd, and J. Widmer. "TCP Friendly Rate Control (TFRC): Protocol Specification," Internet Engineering Task Force, RFC 3448, January 2003.

79. O. Hodson, C. Perkins, and V. Hardman. "Skew Detection and Compensation for Internet Audio Applications," Proceedings of the IEEE International Conference on Multimedia and Expo, New York, July 2000.

80. C. Perkins, O. Hodson, and V. Hardman. "A Survey of Packet Loss Recovery Techniques for Streaming Media", IEEE Network Magazine, September/October 1998.

81. V. Jacobson. "Congestion Avoidance and Control," Proceedings of ACM SIGCOMM '88, Stanford, CA, August 1988.

82. N. S. Jayant and S. W. Christensen. "Effects of Packet Losses in Waveform Codec Speech and Improvements Due to an Odd-Even Sample Interpolation Procedure," *IEEE Transactions on Communications*, Volume 29, Number 2, February 1981.

83. I. Kouvelas and V. Hardman. "Overcoming Workstation Scheduling Problems in a Real-Time Audio Tool," Proceedings of the USENIX Annual Technical Conference, Anaheim, CA, January 1997.

84. I. Kouvelas, V. Hardman, and A. Sasse. "Lip Synchronization for Use over the Internet: Analysis and Implementation," Proceedings of the IEEE Global Internet Symposium, London, November 1996.

85. J. Nonnenmacher, E. Biersack, and D. Towsley. "Parity-Based Loss Recovery for Reliable Multicast Transmission," Proceedings of ACM SIGCOMM '97, Cannes, France, September 1997.

86. S. McCanne, V. Jacobson, and M. Vetterli. "Receiver-Driven Layered Multicast," Proceedings of ACM SIGCOMM '96, Stanford, CA, August 1996.

87. S. McCanne and V. Jacobson. "vic: A Flexible Framework for Packet Video," Proceedings of ACM Multimedia '95, San Francisco, CA, November 1995.

88. G. A. Miller and J. C.R. Licklider. "The Intelligibility of Interrupted Speech," *Journal of the Acoustical Society of America*, Volume 22, Number 2, 1950.

89. S. Moon, J. Kurose, P. Skelly, and D. Towsley. "Correlation of Packet Delay and Loss in the Internet," Technical Report 98-11, Department of Computer Science, University of Massachusetts, Amherst, 1998.

90. S. Moon, P. Skelly, and D. Towsley. "Estimation and Removal of Clock Skew from Network Delay Measurements," Proceedings of IEEE Infocom '99, New York, March 1999.

91. S. Moon, J. Kurose, and D. Towsley. "Packet Audio Playout Delay Adjustment: Performance Bounds and Algorithms," *ACM/Springer Multimedia Systems*, Volume 6, Number 1, January 1998.

92. W. A. Montgomery. "Techniques for Packet Voice Synchronization," *IEEE Journal on Selected Areas in Communications*, Volume 6, Number 1, December 1983.

93. J. D. Day and H. Zimmermann. "The OSI Reference Model," *Proceedings of the IEEE*, Volume 71, December 1983.

94. J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. "Modeling TCP Throughput: A Simple Model and its Empirical Validation," Proceedings of ACM SIGCOMM '98, Vancouver, Canada, August 1998.

95. V. Paxson. "End-to-End Internet Packet Dynamics," *IEEE/ACM Transactions on Networking*, Volume 7, Number 3, June 1999. An earlier version appeared in the Proceedings of ACM SIGCOMM '97, Cannes, France, September 1997.

96. R. Ramjee, J. Kurose, D. Towsley, and H. Schulzrinne. "Adaptive Playout Mechanisms for Packetized Audio Applications in Wide-Area Networks," Proceedings of IEEE Infocom '94, Toronto, June 1994.

97. J. L. Ramsey. "Realization of Optimum Interleavers," *IEEE Transactions on Information Theory*, Volume 16, Number 3, May 1970.

98. I. S. Reed and G. Solomon. "Polynomial Codes over Certain Finite Fields," *Journal of the Society of Industrial and Applied Mathematics*, Volume 9, Number 2, June 1960.

99. R. Rejaie, M. Handley, and D. Estrin. "RAP: An End-to-End Rate-Based Congestion Control Mechanism for Realtime Streams in the Internet," Proceedings of IEEE Infocom '99, New York, March 1999.

100. J. Rosenberg and H. Schulzrinne. "Timer Reconsideration for Enhanced RTP Scalability," Proceedings of IEEE Infocom '98, San Francisco, CA, March 1998.

101. J. Rosenberg, L. Qiu, and H. Schulzrinne. "Integrating Packet FEC into Adaptive Voice Playout Buffer Algorithms on the Internet," Proceedings of IEEE Infocom 2000, Tel Aviv, March 2000.

102. H. Sanneck, A. Stenger, K. B. Younes, and B. Girod. "A New Technique for Audio Packet Loss Concealment," Proceedings of the IEEE Global Internet Symposium, London, December 1996.

103. J. Stone. "When the CRC and TCP Checksum Disagree," Proceedings of ACM SIGCOMM 2000, Stockholm, September 2000.

104. L. Vicisano, L. Rizzo, and J. Crowcroft. "TCP-Like Congestion Control for Layered Multicast Data Transfer," Proceedings of IEEE Infocom '98, San Francisco, CA, March 1998.

105. Y. Wang, S. Wenger, J. Wen, and A. K. Katsaggelos. "Error Resilient Video Coding Techniques," *IEEE Signal Processing Magazine*, Volume 19, Number 4, July 2000.

106. Y. Wang and Q-F. Zhu. "Error Control and Concealment for Video Communication: A Review," *Proceedings of the IEEE*, Volume 86, Number 5, May 1998.

107. O. J. Wasem, D. J. Goodman, C. S. Dvorak, and H. G. Page. "The Effect of Waveform Substitution on the Quality of PCM Packet Communications," *IEEE Transactions on Acoustics, Speech and Signal Processing*, Volume 36, Number 3, March 1988.

108. M. Yajnik, J. Kurose, and D. Towsley. "Packet Loss Correlation in the Mbone Multicast Network," Proceedings of IEEE Global Internet Symposium, London, November 1996. Also Technical Report 96-32, Department of Computer Science, University of Massachusetts, Amherst, 1996.

109. M. Yajnik, S. Moon, J. Kurose, and D. Towsley. "Measurement and Modeling of the Temporal Dependence in Packet Loss," Proceedings of IEEE Infocom '99, New York, March 1999. Also Technical Report 98-78, Department of Computer Science, University of Massachusetts, Amherst, 1998.

Books

110. N. Doraswamy and D. Harkins, *IPSec: The New Security Standard for the Internet, Intranets, and Virtual Private Networks*, Prentice Hall, Upper Saddle River, NJ, 1999.

111. A. B. Johnston. *SIP: Understanding the Session Initiation Protocol*, Artech House, Norwood, MA, 2001.

112. W. R. Stevens. *TCP/IP Illustrated, Volume 1*, Addison Wesley, Reading, MA, 1994.

113. A. S. Tanenbaum. *Computer Networks*, 3rd Edition, Prentice Hall, Upper Saddle River, NJ, 1996.

114. R. M. Warren. *Auditory Perception—A New Analysis and Synthesis*, Cambridge University Press, Cambridge, England, 1999.

Web Sites

115. The 3rd Generation Partnership Project, http://www.3gpp.org.

116. The Internet Traffic Archive, http://www.acm.org/sigcomm/ITA.

117. Cooperative Association of Internet Data Analysis, http://www.caida.org.

118. Telcordia Internet Sizer, Internet Growth Forecasting Tool, 2000, http://www.telcordia.com/research/netsizer.

119. National Laboratory for Applied Network Research, http://www.nlanr.net.

120. The Internet Weather Report, http://www.internetweather.com.

Other References

121. M. Eubanks. Personal communication and discussion on the Mbone@isi.edu mailing list, April 2001.

122. M. Handley. "An Examination of Mbone Performance," University of Southern California, Information Sciences Institute, Research Report ISI/RR-97-450, 1997.

123. S. Leinen. "UDP vs. TCP Distribution," message to the end2end-interest mailing list, February 2001.

124. V. Paxson. "Measurements and Analysis of End-to-End Internet Dynamics," Ph.D. dissertation, University of California, Berkeley, 1997.

125. D. Sisalem, H. Schulzrinne, and F. Emanuel. "The Direct Adjustment Algorithm: A TCP-Friendly Adaptation Scheme," Technical Report, GMD-FOKUS, August 1997.