CS232 Final - Application layer

need help in preparing study materials for cs232, I will give prof's selected topics and book text

2.6: architecture and motivation, bitTorrent (architecture and operations).

BitTorrent
BitTorrent is a popular P2P protocol for file distribution [Chao 2011]. In BitTorrent lingo, the collection of all peers participating in the distribution of a particular file is called a torrent. Peers in a torrent download equal-size chunks of the file from one another, with a typical chunk size of 256 KBytes. When a peer first joins a torrent, it has no chunks. Over time it accumulates more and more chunks. While it downloads chunks it also uploads chunks to other peers. Once a peer has acquired the entire file, it may (selfishly) leave the torrent, or (altruistically) remain in the torrent and continue to upload chunks to other peers. Also, any peer may leave the torrent at any time with only a subset of chunks, and later rejoin the torrent.
Let's now take a closer look at how BitTorrent operates. Since BitTorrent is a rather complicated protocol and system, we'll only describe its most important mechanisms, sweeping some of the details under the rug; this will allow us to see the forest through the trees. Each torrent has an infrastructure node called a tracker. When a peer joins a torrent, it registers itself with the tracker and periodically informs the tracker that it is still in the torrent. In this manner, the tracker keeps track of the peers that are participating in the torrent. A given torrent may have fewer than ten or more than a thousand peers participating at any instant of time.
As shown in Figure 2.26, when a new peer, Alice, joins the torrent, the tracker randomly selects a subset of peers (for concreteness, say 50) from the set of participating peers, and sends the IP addresses of these 50 peers to Alice. Possessing this list of peers, Alice attempts to establish concurrent TCP connections with all the peers on this list. Let's call all the peers with which Alice succeeds in establishing a TCP connection "neighboring peers." (In Figure 2.26, Alice is shown to have only three neighboring peers. Normally, she would have many more.) As time evolves, some of these peers may leave and other peers (outside the initial 50) may attempt to establish TCP connections with Alice. So a peer's neighboring peers will fluctuate over time.
At any given time, each peer will have a subset of chunks from the file, with different peers having different subsets. Periodically, Alice will ask each of her neighboring peers (over the TCP connections) for the list of the chunks they have. If Alice has L different neighbors, she will obtain L lists of chunks. With this knowledge, Alice will issue requests (again over the TCP connections) for chunks she currently does not have. So at any given instant of time, Alice will have a subset of chunks and will know which chunks her neighbors have. With this information, Alice will have two important decisions to make. First, which chunks should she request first from her neighbors? And second, to which of her neighbors should she send requested chunks? In deciding which chunks to request, Alice uses a technique called rarest first. The idea is to determine, from among the chunks she does not have, the chunks that are the rarest among her neighbors (that is, the chunks that have the fewest repeated copies among her neighbors) and then request those rarest chunks first. In this manner, the rarest chunks get more quickly redistributed, aiming to

(roughly) equalize the numbers of copies of each chunk in the torrent.
2.6 • PEER-TO-PEER APPLICATIONS 149
To determine which requests she responds to, BitTorrent uses a clever trading
algorithm. The basic idea is that Alice gives priority to the neighbors that are currently supplying
her data at the highest rate. Specifically, for each of her neighbors,
Alice continually measures the rate at which she receives bits and determines the four
peers that are feeding her bits at the highest rate. She then reciprocates by sending
chunks to these same four peers. Every 10 seconds, she recalculates the rates and possibly
modifies the set of four peers. In BitTorrent lingo, these four peers are said to
be unchoked. Importantly, every 30 seconds, she also picks one additional neighbor
at random and sends it chunks. Let's call the randomly chosen peer Bob. In BitTorrent lingo, Bob
is said to be optimistically unchoked. Because Alice is sending data
to Bob, she may become one of Bob's top four uploaders, in which case Bob would
start to send data to Alice. If the rate at which Bob sends data to Alice is high enough,
Bob could then, in turn, become one of Alice's top four uploaders. In other words,
every 30 seconds, Alice will randomly choose a new trading partner and initiate trading with that
partner. If the two peers are satisfied with the trading, they will put each
other in their top four lists and continue trading with each other until one of the peers
finds a better partner. The effect is that peers capable of uploading at compatible rates
tend to find each other. The random neighbor selection also allows new peers to get
150 CHAPTER 2 • APPLICATION LAYER
Tracker
Trading chunks
Peer
Obtain
list of
peers
Alice
Figure 2.26 ␂ File distribution with BitTorrent
chunks, so that they can have something to trade. All other neighboring peers besides
these five peers (four "top" peers and one probing peer) are "choked," that is, they do
not receive any chunks from Alice. BitTorrent has a number of interesting mechanisms that are
not discussed here, including pieces (mini-chunks), pipelining, random
first selection, endgame mode, and anti-snubbing [Cohen 2003].
The incentive mechanism for trading just described is often referred to as tit-for-tat
[Cohen 2003]. It has been shown that this incentive scheme can be circumvented
[Liogkas 2006; Locher 2006; Piatek 2007]. Nevertheless, the BitTorrent ecosystem is
wildly successful, with millions of simultaneous peers actively sharing files in hundreds of
thousands of torrents. If BitTorrent had been designed without tit-for-tat (or a
variant), but otherwise exactly the same, BitTorrent would likely not even exist now, as
the majority of the users would have been freeriders [Saroiu 2002].
Interesting variants of the BitTorrent protocol are proposed [Guo 2005; Piatek
2007]. Also, many of the P2P live streaming applications, such as PPLive and
ppstream, have been inspired by BitTorrent [Hei 2007]

Perfect—this is a **high-yield CS232 exam topic**, and your book excerpt is exactly what they test. I'll convert this into **clean, structured study notes** you can directly memorize.

✅ BitTorrent (Section 2.6): Architecture & Motivation

Peer-to-Peer (P2P) File Distribution

1️⃣ Motivation: Why BitTorrent Exists

Traditional **client–server distribution**:

- Server must upload the file to **every client**
- Server becomes the **bottleneck**
- Poor scalability

BitTorrent solves this by:

- Turning **every downloader into an uploader**
- Using **P2P chunk-based sharing**
- Scaling naturally with the number of users

✅ **Key Idea**:

> The more peers that join → the faster the system becomes.

2️⃣ Basic Terminology

Term	Meaning
Torrent	All peers sharing one file
Chunk	File divided into fixed-size pieces (≈256 KB)
Peer	Any user downloading/uploading
Seed	A peer with the full file
Leecher	A peer still downloading
Tracker	Central server that helps peers find each other

3️⃣ High-Level Architecture

Step 1: Join the Torrent

1. Peer (Alice) contacts the **tracker**
2. Tracker returns **IP addresses of ~50 random peers**

Step 2: Neighbor Connections

- Alice opens **TCP connections** to these peers

- Peers she successfully connects to = **neighboring peers**
- Neighbors **change dynamically**

Step 3: Chunk Knowledge Exchange

- Alice periodically asks each neighbor:

  > “Which chunks do you have?”

- She now knows:
  - What chunks she has
  - What chunks each neighbor has

4️⃣ Chunk Selection: Rarest First Strategy

Goal: Prevent missing chunks later.

How it works:

- Among all missing chunks,
- Alice requests the **chunk that appears in the fewest neighbors**

✅ **Why this works:**

- Prevents **chunk extinction**
- Ensures **uniform chunk distribution**
- Maximizes long-term throughput

5️⃣ Upload Decision: Tit-For-Tat Incentive Mechanism

This is the **core of BitTorrent’s success**.

Alice does two things:

✅ (A) Unchoking – Every 10 Seconds

- Alice measures:

  > “Who uploads to me the fastest?”

- Selects **Top 4 fastest peers**
- Sends chunks to **only these 4**
- These 4 peers are called:

  Unchoked peers

✅ (B) Optimistic Unchoking – Every 30 Seconds

- Alice randomly picks **1 new neighbor**
- Sends them chunks anyway
- This allows:
  - New peers to join trading
  - Discovery of better trading partners

✅ Final Upload Set = 5 total peers

- 4 best uploaders (tit-for-tat)
- 1 random probe (optimistic unchoke)

All others = **choked (receive nothing)**

6️⃣ Why Tit-For-Tat is Critical

With Tit-For-Tat	Without Tit-For-Tat
Fair sharing	Everyone freeloads
Upload rewarded	No incentive to upload
System scales	System collapses

✅ Without tit-for-tat → **free riders dominate** → **BitTorrent dies**

7️⃣ Key Properties You Must Remember for Exams

✅ **Chunk-based distribution**
✅ **Rarest-first chunk request**
✅ **Top-4 unchoked peers**
✅ **1 optimistically unchoked peer**
✅ **Tracker only helps with peer discovery (not data transfer)**
✅ **TCP used between peers**
✅ **Peers are both uploaders and downloaders**

8️⃣ One-Line Exam Summary (MEMORIZE THIS)

> *BitTorrent is a scalable P2P file distribution protocol where peers exchange file chunks using rarest-first request selection and tit-for-tat upload reciprocity with optimistic unchoking to incentivize fairness and maximize throughput.*

9️⃣ Typical CS232 Exam Questions

Here's what they usually ask:

1. ❓ Why is **rarest-first** used?
2. ❓ What is **tit-for-tat** in BitTorrent?
3. ❓ Difference between **unchoked** and **optimistically unchoked**
4. ❓ What does the **tracker do** and what it **does NOT do**
5. ❓ Why BitTorrent scales better than client–server
6. ❓ What happens if all users act selfishly

✅ If you want, I can now also give you:

- A **1-page final exam cheat sheet**
- Or **5–10 MCQs** exactly like your CS232 final
- Or convert this into a **LaTeX exam note**

Just tell me how you want to study it 🔥

entire 7.1, explain the stuffs in a detailed manner so that reading the notes are enough, not just bullet point

7.1 Multimedia Networking Applications
We define a multimedia network application as any network application that
employs audio or video. In this section, we provide a taxonomy of multimedia applications. We'll
see that each class of applications in the taxonomy has its own unique
set of service requirements and design issues. But before diving into an in-depth discussion of
Internet multimedia applications, it is useful to consider the intrinsic
characteristics of the audio and video media themselves.
7.1.1 Properties of Video
Perhaps the most salient characteristic of video is its high bit rate. Video distributed over the
Internet typically ranges from 100 kbps for low-quality video conferencing to over 3 Mbps for
streaming high-definition movies. To get a sense of how
video bandwidth demands compare with those of other Internet applications, let's
briefly consider three different users, each using a different Internet application. Our
first user, Frank, is going quickly through photos posted on his friends' Facebook
pages. Let's assume that Frank is looking at a new photo every 10 seconds, and that
photos are on average 200 Kbytes in size. (As usual, throughout this discussion we
make the simplifying assumption that 1 Kbyte = 8,000 bits.) Our second user,
Martha, is streaming music from the Internet ("the cloud") to her smartphone. Let's
assume Martha is listening to many MP3 songs, one after the other, each encoded at
a rate of 128 kbps. Our third user, Victor, is watching a video that has been encoded
at 2 Mbps. Finally, let's suppose that the session length for all three users is 4,000
seconds (approximately 67 minutes). Table 7.1 compares the bit rates and the total
bytes transferred for these three users. We see that video streaming consumes by far
588 CHAPTER 7 • MULTIMEDIA NETWORKING

the most bandwidth, having a bit rate of more than ten times greater than that of the Facebook and music-streaming applications. Therefore, when designing networked video applications, the first thing we must keep in mind is the high bit-rate requirements of video. Given the popularity of video and its high bit rate, it is perhaps not surprising that Cisco predicts [Cisco 2011] that streaming and stored video will be approximately 90 percent of global consumer Internet traffic by 2015.

Another important characteristic of video is that it can be compressed, thereby trading off video quality with bit rate. A video is a sequence of images, typically being displayed at a constant rate, for example, at 24 or 30 images per second. An uncompressed, digitally encoded image consists of an array of pixels, with each pixel encoded into a number of bits to represent luminance and color. There are two types of redundancy in video, both of which can be exploited by video compression. Spatial redundancy is the redundancy within a given image. Intuitively, an image that consists of mostly white space has a high degree of redundancy and can be efficiently compressed without significantly sacrificing image quality. Temporal redundancy reflects repetition from image to subsequent image. If, for example, an image and the subsequent image are exactly the same, there is no reason to re-encode the subsequent image; it is instead more efficient simply to indicate during encoding that the subsequent image is exactly the same. Today's off-the-shelf compression algorithms can compress a video to essentially any bit rate desired. Of course, the higher the bit rate, the better the image quality and the better the overall user viewing experience.

We can also use compression to create multiple versions of the same video, each at a different quality level. For example, we can use compression to create, say, three versions of the same video, at rates of 300 kbps, 1 Mbps, and 3 Mbps. Users can then decide which version they want to watch as a function of their current available bandwidth. Users with high-speed Internet connections might choose the 3 Mbps version; users watching the video over 3G with a smartphone might choose the 300 kbps version. Similarly, the video in a video conference application can be compressed "on-the-fly" to provide the best video quality given the available end-to-end bandwidth between conversing users.

7.1 • MULTIMEDIA NETWORKING APPLICATIONS 589

	Bit rate	Bytes transferred in 67 min
Facebook Frank	160 kbps	80 Mbytes
Martha Music	128 kbps	64 Mbytes
Victor Video	2 Mbps	1 Gbyte

Table 7.1 Comparison of bit-rate requirements of three Internet applications

7.1.2 Properties of Audio

Digital audio (including digitized speech and music) has significantly lower bandwidth requirements than video. Digital audio, however, has its own unique properties that must be considered when designing multimedia network applications. To understand these properties, let's first consider how analog audio (which humans and musical instruments generate) is converted to a digital signal:

- The analog audio signal is sampled at some fixed rate, for example, at 8,000 samples per second. The value of each sample is an arbitrary real number.
- Each of the samples is then rounded to one of a finite number of values. This operation is referred to as quantization. The number of such finite values—called quantization values—is typically a power of two, for example, 256 quantization values.
- Each of the quantization values is represented by a fixed number of bits. For example, if there are 256 quantization values, then each value—and hence each audio sample—is represented by one byte. The bit representations of all the samples are then

concatenated together to form the digital representation of the signal. As an example, if an analog audio signal is sampled at 8,000 samples per second and each sample is quantized and represented by 8 bits, then the resulting digital signal will have a rate of 64,000 bits per second. For playback through audio speakers, the digital signal can then be converted back—that is, decoded—to an analog signal. However, the decoded analog signal is only an approximation of the original signal, and the sound quality may be noticeably degraded (for example, high-frequency sounds may be missing in the decoded signal). By increasing the sampling rate and the number of quantization values, the decoded signal can better approximate the original analog signal. Thus (as with video), there is a trade-off between the quality of the decoded signal and the bit-rate and storage requirements of the digital signal.

The basic encoding technique that we just described is called pulse code modulation (PCM). Speech encoding often uses PCM, with a sampling rate of 8,000 samples per second and 8 bits per sample, resulting in a rate of 64 kbps. The audio compact disk (CD) also uses PCM, with a sampling rate of 44,100 samples per second with 16 bits per sample; this gives a rate of 705.6 kbps for mono and 1.411 Mbps for stereo.

PCM-encoded speech and music, however, are rarely used in the Internet. Instead, as with video, compression techniques are used to reduce the bit rates of the stream. Human speech can be compressed to less than 10 kbps and still be intelligible. A popular compression technique for near CD-quality stereo music is MPEG 1 layer 3, more commonly known as MP3. MP3 encoders can compress to many different rates; 128 kbps is the most common encoding rate and produces very little sound degradation. A related standard is Advanced Audio Coding (AAC), which has been popularized by Apple. As with video, multiple versions of a prerecorded audio stream can be created, each at a different bit rate.

Although audio bit rates are generally much less than those of video, users are generally much more sensitive to audio glitches than video glitches. Consider, for example, a video conference taking place over the Internet. If, from time to time, the video signal is lost for a few seconds, the video conference can likely proceed without too much user frustration. If, however, the audio signal is frequently lost, the users may have to terminate the session.

7.1.3 Types of Multimedia Network Applications

The Internet supports a large variety of useful and entertaining multimedia applications. In this subsection, we classify multimedia applications into three broad categories: (i) streaming stored audio/video, (ii) conversational voice/video-over-IP, and (iii) streaming live audio/video. As we will soon see, each of these application categories has its own set of service requirements and design issues.

Streaming Stored Audio and Video

To keep the discussion concrete, we focus here on streaming stored video, which typically combines video and audio components. Streaming stored audio (such as streaming music) is very similar to streaming stored video, although the bit rates are typically much lower.

In this class of applications, the underlying medium is prerecorded video, such as a movie, a television show, a prerecorded sporting event, or a prerecorded user-generated video (such as those commonly seen on YouTube). These prerecorded videos are placed on servers, and users send requests to the servers to view the videos on demand. Many Internet companies today provide streaming video, including YouTube (Google), Netflix, and Hulu. By some estimates, streaming stored video makes up over 50 percent of the downstream traffic in the Internet access networks today [Cisco 2011]. Streaming stored video has three key distinguishing features.

- Streaming. In a streaming stored video application, the client typically begins video playout within a few seconds after it begins receiving the video from the server. This means that the client will be playing out from one location in the video while at the same time receiving later parts of the video from the server. This technique, known as streaming, avoids having to download the entire video file (and incurring a potentially long delay) before playout begins.
- Interactivity. Because the media is prerecorded, the user may pause, reposition forward, reposition backward, fast-forward, and so on through the video content. The time from when the user makes such a request until the action manifests itself at the client should be less than a few seconds for acceptable responsiveness.
- Continuous playout. Once playout of the video begins, it should proceed according to the original timing of the recording. Therefore, data must be received from the server in time for its playout at the client; otherwise, users experience video frame freezing (when the client waits for the delayed frames) or frame skipping (when the client skips over delayed frames).

By far, the most important performance measure for streaming video is average throughput. In order to provide continuous playout, the network must provide an average throughput to the streaming application that is at least as large the bit rate of the video itself. As we will see in Section 7.2, by using buffering and prefetching, it is possible to provide continuous playout even when the throughput fluctuates, as long as the average throughput (averaged over 5–10 seconds) remains above the video rate [Wang 2008].

For many streaming video applications, prerecorded video is stored on, and streamed from, a CDN rather than from a single data center. There are also many P2P video streaming applications for which the video is stored on users' hosts (peers), with different chunks of video arriving from different peers that may spread around the globe. Given the prominence of Internet video streaming, we will explore video streaming in some depth in Section 7.2, paying particular attention to client buffering, prefetching, adapting quality to bandwidth availability, and CDN distribution.

Conversational Voice- and Video-over-IP

Real-time conversational voice over the Internet is often referred to as Internet telephony, since, from the user's perspective, it is similar to the traditional circuit-switched telephone service. It is also commonly called Voice-over-IP (VoIP). Conversational video is similar, except that it includes the video of the participants as well as their voices. Most of today's voice and video conversational systems allow users to create conferences with three or more participants. Conversational voice and video are widely used in the Internet today, with the Internet companies Skype, QQ, and Google Talk boasting hundreds of millions of daily users.

In our discussion of application service requirements in Chapter 2 (Figure 2.4), we identified a number of axes along which application requirements can be classified. Two of these axes—timing considerations and tolerance of data loss—are particularly important for conversational voice and video applications. Timing considerations are important because audio and video conversational applications are highly delay-sensitive. For a conversation with two or more interacting speakers, the delay from when a user speaks or moves until the action is manifested at the other end should be less than a few hundred milliseconds. For voice, delays smaller than 150 milliseconds are not perceived by a human listener, delays between 150 and 400 milliseconds can be acceptable, and delays exceeding 400 milliseconds can result in frustrating, if not completely unintelligible, voice conversations.

On the other hand, conversational multimedia applications are loss-tolerant—occasional loss only causes occasional glitches in audio/video playback, and these losses can often be partially or fully concealed. These delay-sensitive but loss-tolerant

592 CHAPTER 7 • MULTIMEDIA NETWORKING

characteristics are clearly different from those of elastic data applications such as Web browsing, e-mail, social networks, and remote login. For elastic applications, long delays are annoying but not particularly harmful; the completeness and integrity of the transferred data, however, are of paramount importance. We will explore conversational voice and video in more depth in Section 7.3, paying particular attention to how adaptive playout, forward error correction, and error concealment can mitigate against network-induced packet loss and delay.

Streaming Live Audio and Video

This third class of applications is similar to traditional broadcast radio and television, except that transmission takes place over the Internet. These applications allow a user to receive a live radio or television transmission—such as a live sporting event or an ongoing news event—transmitted from any corner of the world. Today, thousands of radio and television stations around the world are broadcasting content over the Internet.

Live, broadcast-like applications often have many users who receive the same audio/video program at the same time. Although the distribution of live audio/video to many receivers can be efficiently accomplished using the IP multicasting techniques described in Section 4.7, multicast distribution is more often accomplished today via application-layer multicast (using P2P networks or CDNs) or through multiple separate unicast streams. As with streaming stored multimedia, the network must provide each live multimedia flow with an average throughput that is larger than the video consumption rate. Because the event is live, delay can also be an issue, although the timing constraints are much less stringent than those for conversational voice. Delays of up to ten seconds or so from when the user chooses to view a live transmission to when playout begins can be tolerated. We will not cover streaming live media in this book because many of the techniques used for streaming live media—initial buffering delay, adaptive bandwidth use, and CDN distribution—are similar to those for streaming stored media.

Excellent—this is a **core, theory-heavy CS232 final topic**, and you're right to want it in a **fully explained, self-contained form**. Below is a **complete, clean, detailed explanation of ALL of Section 7.1**, written so that **you do NOT need to go back to the book** afterward.

You should treat this as your **primary exam study note** for Multimedia Networking Applications.

7.1 Multimedia Networking Applications — Complete Conceptual Explanation

A **multimedia networking application** is any networked application that transmits **audio, video, or both**. This includes everything from YouTube and Netflix to Zoom calls, Spotify, and live sports streaming.

Unlike traditional data applications (such as web browsing or email), multimedia applications have **strict timing, bandwidth, and quality requirements**, which makes their network design fundamentally different.

The goal of this section is to:

1. Understand the **inherent properties of video and audio**
2. Classify the **three major types of multimedia network applications**
3. Understand how their **network requirements differ**

Before designing networks for multimedia, we must first understand **what makes audio and video fundamentally different from ordinary data**.

7.1.1 Properties of Video

1. Video Has Extremely High Bit-Rate Requirements

The most important defining property of video is its **enormous bandwidth consumption**.

Typical video bit rates:

- **Low-quality video conferencing:** ~100 kbps
- **High-definition streaming:** > 3 Mbps

To understand how large this is, compare three users over the same session time (67 minutes):

User	Application	Bit Rate	Total Data
Frank	Facebook Photos	160 kbps	80 MB
Martha	Music Streaming	128 kbps	64 MB
Victor	Video Streaming	2 Mbps	**1 GB**

This shows that **video consumes more than 10× the bandwidth** of standard Internet apps. Because of this:

- Video dominates Internet traffic
- Cisco predicted as early as 2011 that **~90% of consumer Internet traffic would be video**

✅ **Exam takeaway:**

> Video networking is fundamentally constrained by **bandwidth availability**.

---

2. Video is Highly Compressible

A video is simply a **sequence of images (frames)** shown at a constant rate (usually 24–30 frames per second). Each frame consists of millions of pixels, and each pixel is encoded using multiple bits.

If sent without compression, video would be **astronomically large and infeasible to transmit**.

There are two major types of redundancy exploited by compression:

(a) Spatial Redundancy

This refers to **redundancy inside a single frame**.

- Large regions often have the same color or similar brightness
- Example: Blue sky, white walls
- Compression removes repeated pixel patterns

(b) Temporal Redundancy

This refers to **redundancy across consecutive frames**.

- Many frames are very similar
- Instead of sending the full image again, the encoder sends only the **difference**

By exploiting these two redundancies, modern video codecs can compress video to **almost any desired bit rate**.

However:

- **Higher bit rate** → **better quality**
- **Lower bit rate** → **poorer quality**

✅ This creates the fundamental **bit-rate vs quality tradeoff** in video streaming.

---

3. Multiple Versions of the Same Video Can Be Created

A video can be stored in **multiple compressed formats**, for example:

- 300 kbps → low quality (mobile)
- 1 Mbps → medium quality
- 3 Mbps → high-definition

Users then select the video **based on available bandwidth**:

- Smartphone on cellular → low-bit-rate version
- Home broadband → high-bit-rate version

This principle also applies to **live video calls**, where compression is adjusted dynamically to match network conditions.

✅ **Exam takeaway:**

> Video compression allows **adaptive quality based on bandwidth**.

7.1.2 Properties of Audio

Audio consumes **much less bandwidth than video**, but it has **stricter perceptual requirements**—humans are far more sensitive to audio failures than to video glitches.

1. How Analog Sound Becomes Digital

Sound in the real world is **analog (continuous)**. To transmit it digitally, three steps are used:

Step 1: Sampling

The analog wave is measured at regular time intervals.

- Example: 8,000 samples per second

Each sample is a real-number value.

Step 2: Quantization

Each sampled value is **rounded to a fixed number of discrete levels**.

- Example: 256 quantization levels

More quantization levels = better quality = higher bit rate.

Step 3: Encoding

Each quantized value is encoded as bits.

- 256 values → 8 bits per sample

If:

- Sampling rate = 8,000 samples/sec
- Bits per sample = 8
  Then:
- Data rate = 64 kbps

This method is called **Pulse Code Modulation (PCM)**.

2. PCM Usage in Practice

- **Telephone speech:**
  8,000 samples/sec × 8 bits → **64 kbps**
- **CD-quality music:**
  44,100 samples/sec × 16 bits →
    - **705 kbps (mono)**
    - **1.411 Mbps (stereo)**

However, PCM is **too bandwidth-heavy for modern Internet streaming**, so compression is always used.

---

3. Audio Compression

Human speech can be compressed to **< 10 kbps** and still be understandable.

Common compression formats:

- **MP3 (MPEG layer 3):** Typically 128 kbps
- **AAC (Advanced Audio Coding):** Similar quality at lower bit rates

Just like video:

- Audio can be stored in **multiple versions**
- The client selects based on bandwidth

---

4. Audio is More Sensitive than Video

Humans tolerate video loss surprisingly well:

- Frozen frames → slightly annoying
- Dropped video → still acceptable

But humans **do NOT tolerate audio loss well**:

- Missing words → breaks comprehension
- Frequent glitches → conversation becomes impossible

✅ **Critical exam fact:**

> Users are **more sensitive to audio glitches than video glitches**.

7.1.3 Types of Multimedia Network Applications

Multimedia applications fall into **three main categories**. Each has its own **delay tolerance, bandwidth needs, and loss tolerance**.

---

1. Streaming Stored Audio and Video

Examples:

- YouTube
- Netflix
- Spotify
- Hulu

These involve **pre-recorded media stored on servers**.

✅ Three Defining Properties

(a) Streaming (Not Full Download First)

The client:

- Starts playing the video **within seconds**
- While simultaneously **downloading future parts**

This avoids long wait times before playback begins.

(b) Interactivity

Because the media is stored:

- Users may pause
- Jump forward/backward
- Fast-forward

All these actions must respond **within a few seconds** or users get frustrated.

(c) Continuous Playout

Once started:

- Video must play at a fixed rate
- If data arrives late → frame freezing or skipping occurs
- This severely degrades user experience

✅ Key Performance Metric

The **most important performance requirement is average throughput**.

To sustain smooth video:

> The **average network throughput must be ≥ video bit rate**

Short-term drops can be handled using **buffering**, as long as:

- The **average over ~5–10 seconds** remains sufficient

✅ Network Infrastructure Used

Stored video is usually delivered via:

- **CDNs (Content Distribution Networks)** like Akamai
- Sometimes via **P2P video streaming**

---

2. Conversational Voice and Video over IP (VoIP)

Examples:

- Skype
- Zoom
- WhatsApp calls
- Google Meet

These involve **real-time two-way conversation**.

✅ Two Key Requirements

(1) Ultra Low Delay

Conversational voice/video is **extremely delay-sensitive**:

- < 150 ms → not noticeable
- 150–400 ms → tolerable
- 400 ms → frustrating / unusable

Unlike video streaming:

- Long buffering is **not allowed**
- Data must be delivered **immediately**

---

(2) Loss Tolerance

Some packet loss is tolerated:

- Small glitches are acceptable
- Error concealment can hide small losses

This is the opposite of web/email:

- Web/email → loss intolerant but delay tolerant
- VoIP → delay intolerant but loss tolerant

✅ Exam Rule-of-Thumb:

Application	Delay Tolerant	Loss Tolerant
Web / Email	✅ Yes	❌ No
VoIP	❌ No	✅ Yes

---

3. Streaming Live Audio and Video

Examples:

- Live sports streaming
- Live news broadcasts
- Online radio

✅ Key Characteristics

- Content is generated **in real time**
- Many users receive **the same stream simultaneously**
- Distribution is often done using:
  - CDNs
  - P2P networks
  - Application-layer multicast

✅ Network Requirements

- **Continuous throughput ≥ video rate**
- **Initial buffering delays of up to ~10 seconds are acceptable**
- Delay constraints are **less strict than VoIP**, but stricter than stored video

---

✅ Note: This book does NOT explain live streaming in detail because:

> It uses the **same buffering, adaptation, and CDN ideas as stored video streaming**.

✅ FINAL MASTER SUMMARY (Memorize This)

1. **Video** is bandwidth-heavy, highly compressible, and quality scales with bit rate.
2. **Audio** uses sampling, quantization, and compression—users are more sensitive to audio loss than video loss.
3. **Multimedia apps fall into three types:**
   - **Stored streaming:** throughput-dominated, interactive, buffer-based
   - **Conversational VoIP:** delay-sensitive, loss-tolerant

- **Live streaming:** moderate delay tolerance, high throughput demand

---

✅ What I Can Do Next for You

If you want, I can now:

✅ Convert this into:

- A **2-page exam cheat sheet**
- Or **10–15 CS232-style exam questions**
- Or a **side-by-side comparison table for all 3 types**
- Or continue with **Section 7.2 (Streaming Mechanisms)** in the same deep style

👉 Tell me what you want to tackle next.

7.2.1 and 7.2.2

7.2 Streaming Stored Video
For streaming video applications, prerecorded videos are placed on servers, and
users send requests to these servers to view the videos on demand. The user may
watch the video from beginning to end without interruption, may stop watching the
video well before it ends, or interact with the video by pausing or repositioning to a
future or past scene. Streaming video systems can be classified into three categories:
UDP streaming, HTTP streaming, and adaptive HTTP streaming. Although all
three types of systems are used in practice, the majority of today's systems employ
HTTP streaming and adaptive HTTP streaming.
7.2 • STREAMING STORED VIDEO 593
A common characteristic of all three forms of video streaming is the extensive
use of client-side application buffering to mitigate the effects of varying end-to-end
delays and varying amounts of available bandwidth between server and client. For
streaming video (both stored and live), users generally can tolerate a small several␂second initial
delay between when the client requests a video and when video play␂out begins at the client.
Consequently, when the video starts to arrive at the client,
the client need not immediately begin playout, but can instead build up a reserve of
video in an application buffer. Once the client has built up a reserve of several sec␂onds of
buffered-but-not-yet-played video, the client can then begin video playout.
There are two important advantages provided by such client buffering. First, client␂side buffering
can absorb variations in server-to-client delay. If a particular piece of
video data is delayed, as long as it arrives before the reserve of received-but-not␂yet-played video
is exhausted, this long delay will not be noticed. Second, if the
server-to-client bandwidth briefly drops below the video consumption rate, a user
can continue to enjoy continuous playback, again as long as the client application
buffer does not become completely drained.
Figure 7.1 illustrates client-side buffering. In this simple example, suppose that
video is encoded at a fixed bit rate, and thus each video block contains video frames
that are to be played out over the same fixed amount of time, . The server trans␂mits the first

video block at , the second block at , the third block at
, and so on. Once the client begins playout, each block should be played out
time units after the previous block in order to reproduce the timing of the original
recorded video. Because of the variable end-to-end network delays, different video
blocks experience different delays. The first video block arrives at the client at t
1
and the second block arrives at . The network delay for the ith block is the hori␂zontal distance
between the time the block was transmitted by the server and the
t2
␂
t0 + 2␂
t t0 0 + ␂
␂
594 CHAPTER 7 • MULTIMEDIA NETWORKING
Variable
network
delay
Client
playout
delay
Constant bit
rate video
transmission
by server
1
2
3
4
5
6
7
8
9
10
11
12
Constant bit
rate video
playout
by client
Time
Video block number
t0 t1 t2 t3 t0+2Δ
t0+Δ t1+Δ t3+Δ
Video
reception
at client
Figure 7.1 ␂ Client playout delay in video streaming
time it is received at the client; note that the network delay varies from one video
block to another. In this example, if the client were to begin playout as soon as the
first block arrived at , then the second block would not have arrived in time to be

played out at out at . In this case, video playout would either have to stall (waiting for block 1 to arrive) or block 1 could be skipped—both resulting in undesirable playout impairments. Instead, if the client were to delay the start of playout until , when blocks 1 through 6 have all arrived, periodic playout can proceed with all blocks having been received before their playout time.

7.2.1 UDP Streaming

We only briefly discuss UDP streaming here, referring the reader to more in-depth discussions of the protocols behind these systems where appropriate. With UDP streaming, the server transmits video at a rate that matches the client's video consumption rate by clocking out the video chunks over UDP at a steady rate. For example, if the video consumption rate is 2 Mbps and each UDP packet carries 8,000 bits of video, then the server would transmit one UDP packet into its socket every (8000 bits)/(2 Mbps) = 4 msec. As we learned in Chapter 3, because UDP does not employ a congestion-control mechanism, the server can push packets into the network at the consumption rate of the video without the rate-control restrictions of TCP. UDP streaming typically uses a small client-side buffer, big enough to hold less than a second of video.

Before passing the video chunks to UDP, the server will encapsulate the video chunks within transport packets specially designed for transporting audio and video, using the Real-Time Transport Protocol (RTP) [RFC 3550] or a similar (possibly proprietary) scheme. We delay our coverage of RTP until Section 7.3, where we discuss RTP in the context of conversational voice and video systems.

Another distinguishing property of UDP streaming is that in addition to the server-to-client video stream, the client and server also maintain, in parallel, a separate control connection over which the client sends commands regarding session state changes (such as pause, resume, reposition, and so on). This control connection is in many ways analogous to the FTP control connection we studied in Chapter 2. The Real-Time Streaming Protocol (RTSP) [RFC 2326], explained in some detail in the companion Web site for this textbook, is a popular open protocol for such a control connection.

Although UDP streaming has been employed in many open-source systems and proprietary products, it suffers from three significant drawbacks. First, due to the unpredictable and varying amount of available bandwidth between server and client, constant-rate UDP streaming can fail to provide continuous playout. For example, consider the scenario where the video consumption rate is 1 Mbps and the server-to-client available bandwidth is usually more than 1 Mbps, but every few minutes the available bandwidth drops below 1 Mbps for several seconds. In such a scenario, a UDP streaming system that transmits video at a constant rate of 1 Mbps over RTP/UDP would likely provide a poor user experience, with freezing or skipped frames soon after the available bandwidth falls below 1 Mbps. The second drawback of UDP streaming is that it requires a media control server, such as an RTSP server, to process client-to-server interactivity requests and to track client state (e.g., the client's playout point in the video, whether the video is being paused or played, and so on) for each ongoing client session. This increases the overall cost and complexity of deploying a large-scale video-on-demand system. The third drawback is that many firewalls are configured to block UDP traffic, preventing the users behind these firewalls from receiving UDP video.

7.2.2 HTTP Streaming

In HTTP streaming, the video is simply stored in an HTTP server as an ordinary file with a specific URL. When a user wants to see the video, the client establishes a TCP connection with the server and issues an HTTP GET request for that URL. The server then sends the video file, within an HTTP response message, as quickly as possible, that is, as quickly as TCP congestion control and flow control will allow.

On the client side, the bytes are collected in a client application buffer. Once the number of bytes in this buffer exceeds a predetermined threshold, the client application begins playback—specifically, it periodically grabs video frames from the client application buffer, decompresses the frames, and displays them on the user's screen.

We learned in Chapter 3 that when transferring a file over TCP, the server-to-client transmission rate can vary significantly due to TCP's congestion control mechanism. In particular, it is not uncommon for the transmission rate to vary in a "saw-tooth" manner (for example, Figure 3.53) associated with TCP congestion control. Furthermore, packets can also be significantly delayed due to TCP's retransmission mechanism. Because of these characteristics of TCP, the conventional wisdom in the 1990s was that video streaming would never work well over TCP. Over time, however, designers of streaming video systems learned that TCP's congestion control and reliable-data transfer mechanisms do not necessarily preclude continuous playout when client buffering and prefetching (discussed in the next section) are used.

The use of HTTP over TCP also allows the video to traverse firewalls and NATs more easily (which are often configured to block most UDP traffic but to allow most HTTP traffic). Streaming over HTTP also obviates the need for a media control server, such as an RTSP server, reducing the cost of a large-scale deployment over the Internet. Due to all of these advantages, most video streaming applications today—including YouTube and Netflix—use HTTP streaming (over TCP) as its underlying streaming protocol.

Prefetching Video

We just learned, client-side buffering can be used to mitigate the effects of varying end-to-end delays and varying available bandwidth. In our earlier example in Figure 7.1, the server transmits video at the rate at which the video is to be played out. However, for streaming stored video, the client can attempt to download the video at a rate higher than the consumption rate, thereby prefetching video frames that are to be consumed in the future. This prefetched video is naturally stored in the client application buffer. Such prefetching occurs naturally with TCP streaming, since TCP's congestion avoidance mechanism will attempt to use all of the available bandwidth between server and client.

To gain some insight into prefetching, let's take a look at a simple example. Suppose the video consumption rate is 1 Mbps but the network is capable of delivering the video from server to client at a constant rate of 1.5 Mbps. Then the client will not only be able to play out the video with a very small playout delay, but will also be able to increase the amount of buffered video data by 500 Kbits every second. In this manner, if in the future the client receives data at a rate of less than 1 Mbps for a brief period of time, the client will be able to continue to provide continuous playback due to the reserve in its buffer. [Wang 2008] shows that when the average TCP throughput is roughly twice the media bit rate, streaming over TCP results in minimal starvation and low buffering delays.

Client Application Buffer and TCP Buffers

Figure 7.2 illustrates the interaction between client and server for HTTP streaming. At the server side, the portion of the video file in white has already been sent into the server's socket, while the darkened portion is what remains to be sent. After "passing through the socket door," the bytes are placed in the TCP send buffer before being transmitted into the Internet, as described in Chapter 3. In Figure 7.2,

Video file

Web server
Client
TCP send
buffer
TCP receive
buffer
TCP application
buffer
Frames read
out periodically
from buffer,
decompressed,
and displayed
on screen

Figure 7.2 Streaming stored video over HTTP/TCP

because the TCP send buffer is shown to be full, the server is momentarily prevented from sending more bytes from the video file into the socket. On the client side, the client application (media player) reads bytes from the TCP receive buffer (through its client socket) and places the bytes into the client application buffer. At the same time, the client application periodically grabs video frames from the client application buffer, decompresses the frames, and displays them on the user's screen. Note that if the client application buffer is larger than the video file, then the whole process of moving bytes from the server's storage to the client's application buffer is equivalent to an ordinary file download over HTTP—the client simply pulls the video off the server as fast as TCP will allow!

Consider now what happens when the user pauses the video during the streaming process. During the pause period, bits are not removed from the client application buffer, even though bits continue to enter the buffer from the server. If the client application buffer is finite, it may eventually become full, which will cause "back pressure" all the way back to the server. Specifically, once the client application buffer becomes full, bytes can no longer be removed from the client TCP receive buffer, so it too becomes full. Once the client receive TCP buffer becomes full, bytes can no longer be removed from the client TCP send buffer, so it also becomes full. Once the TCP send buffer becomes full, the server cannot send any more bytes into the socket. Thus, if the user pauses the video, the server may be forced to stop transmitting, in which case the server will be blocked until the user resumes the video.

In fact, even during regular playback (that is, without pausing), if the client application buffer becomes full, back pressure will cause the TCP buffers to become full, which will force the server to reduce its rate. To determine the resulting rate, note that when the client application removes f bits, it creates room for f bits in the client application buffer, which in turn allows the server to send f additional bits. Thus, the server send rate can be no higher than the video consumption rate at the client. Therefore, a full client application buffer indirectly imposes a limit on the rate that video can be sent from server to client when streaming over HTTP.

Analysis of Video Streaming

Some simple modeling will provide more insight into initial playout delay and freezing due to application buffer depletion. As shown in Figure 7.3, let B denote the size (in bits) of the client's application buffer, and let Q denote the number of bits that must be buffered before the client application begins playout. (Of course, Q < B.) Let r denote the video consumption rate—the rate at which the client draws bits out of the client application buffer during playback. So, for example, if the video's frame rate is 30 frames/sec, and each (compressed) frame is 100,000 bits, then r = 3 Mbps. To see the forest through the trees, we'll ignore TCP's send and receive buffers Let's assume that the server sends bits at a constant rate x whenever the client buffer is not full. (This is a gross simplification, since TCP's send rate varies due to congestion control; we'll examine more realistic time-dependent rates x(t) in the problems at the end of this chapter.) Suppose at time t = 0, the application buffer is empty and video begins arriving to the client application buffer. We now ask at what time does playout begin? And while we are at it, at what time does the client application buffer become full?

First, let's determine , the time when Q bits have entered the application buffer and playout begins. Recall that bits arrive to the client application buffer at rate x and no bits are removed from this buffer before playout begins. Thus, the amount of time required to build up Q bits (the initial buffering delay) is Q/x.

Now let's determine , the point in time when the client application buffer becomes full. We first observe that if x < r (that is, if the server send rate is less than the video consumption rate), then the client buffer will never become full! Indeed, starting at time , the buffer will be depleted at rate r and will only be filled at rate x < r. Eventually the client buffer will empty out entirely, at which time the video will freeze on the screen while the client buffer waits another seconds to build up Q bits of video. Thus, when the available rate in the network is less than the video rate, playout will alternate between periods of continuous playout and periods of freezing. In a homework problem, you will be asked to determine the length of each continuous playout and freezing period as a function of Q, r, and x. Now let's determine for when x > r. In this case, starting at time , the buffer increases from Q to B at rate x r since bits are being depleted at rate r but are arriving at rate x, as shown in Figure 7.3. Given these hints, you will be asked in a homework problem to determine , the time the client buffer becomes full. Note that when the available rate in the network is more than the video rate, after the initial buffering delay, the user will enjoy continuous playout until the video ends.

tf
-
tp tf
tp
tp
tf
tp =
tp
t = tf t = tp
7.2 • STREAMING STORED VIDEO 599
Fill rate = x Depletion rate = r
Video
server
Internet
Q
B
Client application buffer
Figure 7.3 Analysis of client-side buffering for video streaming

Early Termination and Repositioning the Video

HTTP streaming systems often make use of the HTTP byte-range header in the

HTTP GET request message, which specifies the specific range of bytes the client currently wants to retrieve from the desired video. This is particularly useful when the user wants to reposition (that is, jump) to a future point in time in the video.

When the user repositions to a new position, the client sends a new HTTP request, indicating with the byte-range header from which byte in the file should the server send data. When the server receives the new HTTP request, it can forget about any earlier request and instead send bytes beginning with the byte indicated in the byte-range request.

While we are on the subject of repositioning, we briefly mention that when a user repositions to a future point in the video or terminates the video early, some prefetched-but-not-yet-viewed data transmitted by the server will go unwatched—a waste of network bandwidth and server resources. For example, suppose that the client buffer is full with B bits at some time t 0 into the video, and at this time the user repositions to some instant t > t 0 + B/r into the video, and then watches the video to completion from that point on. In this case, all B bits in the buffer will be unwatched and the bandwidth and server resources that were used to transmit those B bits have been completely wasted. There is significant wasted bandwidth in the Internet due to early termination, which can be quite costly, particularly for wireless links [Ihm 2011]. For this reason, many streaming systems use only a moderate-size client application buffer, or will limit the amount of prefetched video using the byte-range header in HTTP requests [Rao 2011].

Repositioning and early termination are analogous to cooking a large meal, eating only a portion of it, and throwing the rest away, thereby wasting food. So the next time your parents criticize you for wasting food by not eating all your dinner, you can quickly retort by saying they are wasting bandwidth and server resources when they reposition while watching movies over the Internet! But, of course, two wrongs do not make a right—both food and bandwidth are not to be wasted!

Nice, this is one of the densest but most “exam-payoff” sections in the book. I’ll walk through **all the ideas in 7.2.1 and 7.2.2**, including the surrounding buffering/prefetching stuff they tie into, in a way where **this alone should be enough to revise from**.

I’ll keep the structure close to the book, but explain everything in plain language.

---

0. Big Picture: Streaming Stored Video

We’re talking about **stored** (pre-recorded) video: movies, shows, YouTube clips, lectures, etc. The file is on a server; the client requests it and watches with possible interactions:

- Watch straight through
- Stop early
- Pause / seek forward / backward

The book divides streaming systems into three families:

1. UDP streaming

2. HTTP streaming
3. Adaptive HTTP streaming

You only asked for **7.2.1 (UDP streaming)** and **7.2.2 (HTTP streaming)**, but they rely on some shared ideas: especially **client-side buffering**. So I'll start there.

---

1. Client-Side Buffering (Figure 7.1 idea)

Why do we need buffering?

The Internet has:

- **Variable delay**: some packets arrive fast, some slow
- **Fluctuating bandwidth**: sometimes the path is fast, sometimes congested

For video, we want **smooth, continuous playout**. Users are okay with:

- A small **initial delay** (a few seconds of "loading...")
- But **not okay** with the video freezing every few seconds

So the idea:

> Let the client **wait a bit**, gather some video into a **buffer**, and only then start playing.

That way, if the network is temporarily slow, the player can keep showing frames from its buffer instead of freezing.

How the timeline works in the example

Assume:

- Video is encoded at a **fixed rate**
- The server sends the video in **blocks** (block 1, block 2, block 3, ...)
- Each block contains exactly enough frames for a fixed time interval, say **Δ seconds**

Server sends:

- Block 1 at time t_0
- Block 2 at time $t_0 + \Delta$
- Block 3 at time $t_0 + 2\Delta$, etc.

But the **network delay is variable**, so:

- Block 1 arrives at the client at time t_1
- Block 2 arrives at time t_2, etc.
- Delays are different for different blocks

Now, suppose the client **starts playout immediately** when block 1 arrives (time t_1). Then:

- Block 1 needs to be played during $[t_1, t_1 + \Delta]$
- Block 2 must arrive before $t_1 + \Delta$ to be played on time

If, in reality, **block 2 arrives late** (after $t_1 + \Delta$), then:

- At the time we need to show those frames, they aren't there
- The player must either:
  - **Freeze** (stall until block 2 comes), or
  - **Skip** block 2 and jump to block 3

Both are bad for user experience.

Fix: start playout later

Instead of starting at the first arrival time (t_1), the client waits until it has **several blocks** accumulated. For example:

- Wait until time $t_0 + 2\Delta$ or later when **blocks 1–6** have arrived
- Then start playing block 1, followed by 2, 3, …

Because you already have a **reserve of content** ("buffered but not yet played"), later fluctuations in delay won't cause immediate freezes – as long as the network isn't slow for too long.

This buffering gives two advantages:

1. **Absorbs variable delays**: late packets are okay as long as they arrive before their turn comes up in the buffer.
2. **Smooths bandwidth dips**: if network bandwidth briefly drops below the video rate, the player can keep playing from its buffer.

This buffering idea is common to **all** streaming methods: UDP streaming, HTTP streaming, and adaptive streaming.

---

2. 7.2.1 – UDP Streaming

Now we look at the **old-school way**: streaming over UDP.

How UDP streaming works

- The **server sends video at the playback rate** using UDP.
- Example: video consumption rate is 2 Mbps.
  - Each UDP packet carries 8,000 bits.
  - To sustain 2 Mbps, server sends:

$$\text{packet interval} = \frac{8000 \text{ bits}}{2\,000\,000 \text{ bits/s}} = 0.004 \text{ seconds} = 4 \text{ ms}$$

  - So, one UDP packet every 4 ms.
- **No TCP congestion control** here:
  UDP allows the server to push packets at exactly the rate it wants, unconstrained by TCP's additive-increase/multiplicative-decrease behavior.
- The client typically uses a **small buffer** (maybe < 1 second of video) – enough to smooth minor variations in delay, but not big like YouTube's multi-second buffer.

Adding RTP: carrying audio/video over UDP

Before sending video chunks as UDP, the server usually wraps them in **RTP (Real-Time Transport Protocol)** packets:

- RTP is designed specifically for **real-time audio/video**.
- It adds:
  - Sequence numbers (to detect losses, reorder packets)
  - Timestamps (to synchronize playout, especially for audio + video together)

You'll see RTP more in **conversational multimedia (VoIP)** later, but here it's just "the typical way" to packetize video over UDP.

RTSP: separate control connection

Streaming video isn't only "play from start"; users want to:

- Pause
- Resume
- Seek (reposition) forward/back

To handle this, UDP streaming systems have:

- One **media path**: the UDP flow carrying the video (RTP/UDP)
- One **control channel**: usually over TCP, using RTSP (Real-Time Streaming Protocol)

RTSP is like:

- The **FTP control channel**:
  - Client sends commands: PLAY, PAUSE, TEARDOWN, SEEK, etc.
  - Server keeps track of session state for each client: where in the video they are, whether paused, etc.

So UDP streaming typically involves:

- Video: **RTP over UDP**
- Control: **RTSP over TCP**

Why UDP streaming is problematic (three big drawbacks)

Even though this seems “ideal” (no TCP overhead, direct control over rate), UDP streaming has **three important practical problems**:

---

1. No congestion awareness → freezes when bandwidth drops

UDP doesn’t do congestion control. The server **always sends at the video rate**, say 1 Mbps.

Imagine:

- Video rate: 1 Mbps
- Available bandwidth:
  - Typically >1 Mbps
  - But every few minutes, it drops <1 Mbps for a few seconds

When bandwidth < 1 Mbps:

- The stream can’t keep up
- Packets are delayed or dropped
- The client’s small buffer drains and empties
- Result: **freezes or skipped frames** during those periods

Because UDP streaming uses:

- A constant sending rate
- Small buffer
  It’s not very robust to normal Internet fluctuations.

---

2. Needs extra infrastructure: media control server

UDP streaming usually requires a **special media server**:

- That speaks RTSP
- Tracks per-client state (current position, paused/playing, etc.)
- Responds to user commands like pause, resume, seek

For a **large video-on-demand service** (millions of users), running such stateful media servers is more complex and more expensive than simple HTTP servers hosting files.

---

3. Firewalls and NAT often block UDP

Many networks:

- Block arbitrary UDP traffic

- Allow HTTP (TCP port 80/443) freely

So if you rely on UDP streaming:

- A large fraction of users behind corporate or home firewalls **won't be able to receive the video**.
- This is a huge deployment obstacle.

Because of these:

> UDP streaming is less attractive for large, global video platforms.

That's one major reason **modern services moved to HTTP-based streaming** over TCP.

---

3. 7.2.2 – HTTP Streaming

Now, the mainstream approach: **HTTP streaming over TCP**.

Basic idea

In HTTP streaming, the video is just:

- An **ordinary file** sitting on a web server
- Identified by a **URL**

When the user wants to watch:

1. Client opens a **TCP connection** to the web server.
2. Client sends an **HTTP GET** for the video's URL.
3. Server responds with the **video file as the HTTP response body**.
4. The server sends data as fast as TCP allows (subject to congestion control and flow control).
5. At the client:
    - Bytes arrive into the **TCP receive buffer**
    - The application (media player) reads from this and fills the **client's application buffer**
    - When enough data is buffered, the player starts playout: decode & display frames.

So: from TCP's point of view, this is **just a file download**; from the user's point of view, it feels like streaming.

But wait… doesn't TCP variability ruin streaming?

TCP has features that historically made people doubt that video streaming over TCP would work:

- **Congestion control**:
    - Send rate varies in a "saw-tooth" pattern: grows, then drops when loss is detected.
- **Retransmissions**:

    - Lost packets are re-sent
    - Retransmitted packets can be delayed significantly

The concern in the 1990s was:

> “This makes the sending rate too jerky and delayed for smooth video playout.”

But in practice, if we:

- Use **client buffering**
- Allow **prefetching**
  We can still achieve **continuous playout**, as long as the **average throughput** is good enough.

That’s the key insight:
Streaming over TCP works fine with smart buffering strategies.

Advantages of HTTP streaming

1. **Firewall/NAT friendly**
    - Uses ordinary HTTP over TCP (port 80 or 443)
    - Almost always allowed through firewalls and NATs
2. **No special media server required**
    - Just store videos on a standard HTTP server (or CDN edge server)
    - No need to maintain per-client state like RTSP
3. **Leverages CDNs easily**
    - CDNs are naturally built around HTTP/TCP
    - Scaling to many users is straightforward

Because of all this:

> Most real-world streaming platforms (YouTube, Netflix, etc.) rely on **HTTP streaming over TCP**.

The book then dives into **prefetching**, **buffer interactions**, and **behavior when the user pauses or seeks**. Those are part of understanding how HTTP streaming really behaves.

---

4. Prefetching Video over HTTP/TCP

Recall:

- Client can buffer data before starting playout.
- With TCP, the server **tries to send as fast as the network allows**.

That means:

- If the **network is faster than the video rate**, the client will accumulate a **growing reserve of future frames** in its application buffer.
- This is **prefetching**.

Example

Video consumption rate: $r = 1 \text{ Mbps}$
Network can deliver: $x = 1.5 \text{ Mbps}$ (constant, for now)

So each second:

- Client **plays 1 Mbit** (consumes)
- Network **delivers 1.5 Mbits**

Net gain in the buffer:

$$1.5 \text{ Mbps} - 1 \text{ Mbps} = 0.5 \text{ Mbps} = 500 \text{ Kbps}$$

So the buffer grows by **500 Kbits per second**.

Later, if the network slows down and delivers less than 1 Mbps for a while, the client can survive that dip by using its **stored buffer**.

Empirical studies (as the book notes) show:

- If **average TCP throughput** is about **twice** the media bit rate, playout starvation is rare and initial buffering can be small.

So with HTTP/TCP:

> Prefetching + buffering smooth out TCP's variability and allow good user experience.

---

5. Client Application Buffer and TCP Buffers (Figure 7.2)

Now let's look at the **buffer chain** in HTTP streaming.

On the server side

- The video file is on disk.
- Part of it has already been **sent into the socket**.
- TCP places these bytes in the **TCP send buffer**.
  - If that buffer is full, the server **cannot push more data** until some is transmitted and ACKed.

On the client side

There are two layers of buffering:

1. **TCP receive buffer**
   - Managed by the OS/TCP stack

    - Holds bytes received but not yet read by the application
2. **Client application buffer**
    - Managed by the media player
    - Holds data that is available for decoding/playing

Flow:

- Bytes arrive → TCP receive buffer
- Application reads bytes → copies to **client app buffer**
- Player periodically reads frames from client app buffer → decodes → displays

If the **client application buffer is very large** (big enough to hold entire file), then:

- From a data-transfer point of view:
    - The process is just **full file download over HTTP**.
- The streaming behavior is purely handled at the **client side** (when it starts reading and playing the buffered video).

What happens when user hits "pause"?

During pause:

- The player stops **removing bits** from the client application buffer.
- But if the server keeps sending, new bits keep entering.

If buffers are finite, a **back-pressure chain** occurs:

1. Client application buffer fills up.
2. Because it's full, the app **stops reading** from TCP receive buffer.
3. TCP receive buffer fills up.
4. Since receive buffer is full, TCP flow control tells the sender to **stop sending** (zero window advertised).
5. Sender's TCP send buffer becomes full.
6. Server application is blocked from writing more data to the socket.

So, effectively:

> Once the client buffer is full (due to pause), the **server is forced to stop transmitting** until the user resumes.

The same thing can happen even **without pausing**, if:

- The client buffer is relatively small, and
- Prefetching fills it up fully

Once the buffer is full:

- The effective send rate from server is limited:
  - The server can only send new bits when the client consumes some (frames played).
  - When the client consumes f bits, that frees up space for f new bits.
- So, in steady state:

  > Server's send rate cannot exceed the **video consumption rate** at the client.

This is how a **full client buffer indirectly throttles the server** over TCP.

---

6. Simple Analytical Model of Streaming Behavior (Figure 7.3)

Let's formalize some of this with a simple model (ignoring TCP internals):

Definitions:

- B: size of client application buffer (in bits)
- Q: number of bits the client waits to accumulate before starting playout (so $Q < B$)
- r: video consumption rate (bits per second)
  – how fast the player consumes bits during playback
- x: server send rate / effective network throughput (bits per second)
  – assume constant for now

Timeline:

- At time $t = 0$: buffer is empty
- Bits start arriving at rate x
- Client starts playing only when Q **bits have arrived**

6.1. When does playout start?

Playout begins when the buffer has accumulated Q bits.

Since bits arrive at rate x and none are removed before starting:

$$t_p = \frac{Q}{x}$$

This is the **initial buffering delay** (startup delay).

6.2. What happens if $x < r$? (Network slower than video rate)

Once playout starts at time t_p:

- Bits arrive at rate x
- Bits are consumed at rate r

Net rate of change of buffer = $x - r$, which is **negative**.

So the buffer **shrinks** over time and will eventually hit zero:

- When the buffer empties → **video freezes**.
- Client must pause playout and wait to re-buffer Q bits again.
- This creates a cycle:
  1. Buffering period (no playout)
  2. Playout period (buffer drains)
  3. Freeze again, and so on

The book leaves the exact durations as an exercise, but intuitively:

> If the network is **permanently slower** than the video rate, you cannot have uninterrupted streaming, no matter how clever buffering is.

6.3. What happens if $x > r$? (Network faster than video rate)

Once playout begins at t_p:

- Bits arrive at rate x
- Bits are consumed at rate r
- Net change in buffer = $x - r > 0$

So:

- The buffer **fills up** over time, until it reaches capacity B bits.
- At that point, the client will cause back-pressure and throttle the server.

In this case:

> After the initial buffering delay, the user can enjoy **continuous playout until the end of the video**, because on average the network is fast enough to "stay ahead" of playback.

The exact time when the buffer becomes full t_f depends on Q, B, x, r, and again is done in exercises.

---

7. Early Termination and Repositioning (HTTP byte-range)

User behavior matters too:

- Users may **seek forward** (jump ahead) or **stop early**.
- But the server may have already **prefetched** a lot of data into the client's buffer, which the user will never watch.

How repositioning works in HTTP streaming

HTTP has a header: **Range:** (also called byte-range).

Instead of always doing:

```http
GET /video.mp4 HTTP/1.1
```

Client can say:

```http
GET /video.mp4 HTTP/1.1
Range: bytes=20000000-
```

Meaning:

> “Send me the file starting from byte 20,000,000 onward.”

When the user seeks to a future point in the video:

- The media player can map the desired time position to a **byte offset** in the file.
- It then sends a new HTTP GET with a Range header.
- The server stops worrying about the old transfer and starts sending from the new byte position.

Wasted bandwidth

Example:

- At time t_0, the buffer is completely full with B bits of **future content**.
- At that moment, the user:
  - Jumps to a much later time in the video, or
  - Stops watching entirely

All those B bits are:

- **Downloaded**
- **Paid for** (bandwidth, server resources)
- But **never actually viewed**

This is **wasted bandwidth**, particularly problematic on:

- Mobile data
- Wireless links
- Large-scale CDNs

To reduce this waste, systems often:

- Limit how far ahead they prefetch

- Use moderate-sized buffers
- Control prefetching behavior using HTTP Range requests (only request more chunks when needed).

---

8. Mental Summary for Exams

If you're summarizing 7.2.1–7.2.2 in your own words:

- **UDP streaming**:
  - Server sends at video rate over UDP (often RTP).
  - RTSP over TCP for control (pause/seek).
  - Problems: no congestion control (freezes under fluctuating bandwidth), needs special media server, UDP often blocked by firewalls/NAT.
- **HTTP streaming over TCP**:
  - Video stored as a regular file, requested via HTTP GET.
  - Uses client-side buffering and TCP's natural "send as fast as possible" behavior to prefetch.
  - Firewall/NAT-friendly, no special media server, easy to deploy with CDNs.
  - Prefetching and buffering smooth TCP's variable rate and retransmissions.
  - Back-pressure from full buffers throttles the server to roughly the video consumption rate.
  - HTTP Range allows efficient repositioning but aggressive prefetching can cause wasted bandwidth when users seek/quit early.

If you want, I can next:

- Turn this into a **compact formula + diagram summary**, or
- Make **5–10 conceptual/short-answer questions** like your prof might ask on an exam and then walk you through the answers.

7.3: best-effort and QoS (packet loss, delay, jitter), Removing jitter, Recovering packet loss (FEC, interleaving)

7.3 Voice-over-IP
Real-time conversational voice over the Internet is often referred to as Internet telephony, since, from the user's perspective, it is similar to the traditional circuit-switched telephone service. It is also commonly called Voice-over-IP (VoIP). In this section we describe the principles and protocols underlying VoIP. Conversational video is similar in many respects to VoIP, except that it includes the video of the participants as well as their voices. To keep the discussion focused and concrete, we focus here only on voice in this section rather than combined voice and video.
7.3.1 Limitations of the Best-Effort IP Service

The Internet's network-layer protocol, IP, provides best-effort service. That is to say the service makes its best effort to move each datagram from source to destination as quickly as possible but makes no promises whatsoever about getting the packet to the destination within some delay bound or about a limit on the percentage of packets lost. The lack of such guarantees poses significant challenges to the design of real-time conversational applications, which are acutely sensitive to packet delay, jitter, and loss.

In this section, we'll cover several ways in which the performance of VoIP over a best-effort network can be enhanced. Our focus will be on application-layer techniques, that is, approaches that do not require any changes in the network core or even in the transport layer at the end hosts. To keep the discussion concrete, we'll discuss the limitations of best-effort IP service in the context of a specific VoIP example. The sender generates bytes at a rate of 8,000 bytes per second; every 20 msecs the sender gathers these bytes into a chunk. A chunk and a special header (discussed below) are encapsulated in a UDP segment, via a call to the socket interface. Thus, the number of bytes in a chunk is (20 msecs)· (8,000 bytes/sec) = 160 bytes, and a UDP segment is sent every 20 msecs.

If each packet makes it to the receiver with a constant end-to-end delay, then packets arrive at the receiver periodically every 20 msecs. In these ideal conditions,

612 CHAPTER 7 • MULTIMEDIA NETWORKING

the receiver can simply play back each chunk as soon as it arrives. But unfortunately, some packets can be lost and most packets will not have the same end-to-end delay, even in a lightly congested Internet. For this reason, the receiver must take more care in determining (1) when to play back a chunk, and (2) what to do with a missing chunk.

Packet Loss

Consider one of the UDP segments generated by our VoIP application. The UDP segment is encapsulated in an IP datagram. As the datagram wanders through the network, it passes through router buffers (that is, queues) while waiting for transmission on outbound links. It is possible that one or more of the buffers in the path from sender to receiver is full, in which case the arriving IP datagram may be discarded, never to arrive at the receiving application.

Loss could be eliminated by sending the packets over TCP (which provides for reliable data transfer) rather than over UDP. However, retransmission mechanisms are often considered unacceptable for conversational real-time audio applications such as VoIP, because they increase end-to-end delay [Bolot 1996].

Furthermore, due to TCP congestion control, packet loss may result in a reduction of the TCP sender's transmission rate to a rate that is lower than the receiver's drain rate, possibly leading to buffer starvation. This can have a severe impact on voice intelligibility at the receiver. For these reasons, most existing VoIP applications run over UDP by default. [Baset 2006] reports that UDP is used by Skype unless a user is behind a NAT or firewall that blocks UDP segments (in which case TCP is used).

But losing packets is not necessarily as disastrous as one might think.

Indeed, packet loss rates between 1 and 20 percent can be tolerated, depending on how voice is encoded and transmitted, and on how the loss is concealed at the receiver. For example, forward error correction (FEC) can help conceal packet loss. We'll see below that with FEC, redundant information is transmitted along with the original information so that some of the lost original data can be recovered from the redundant information. Nevertheless, if one or more of the links between sender and receiver is severely congested, and packet loss exceeds 10 to

20 percent (for example, on a wireless link), then there is really nothing that can be done to achieve acceptable audio quality. Clearly, best-effort service has its limitations.

End-to-End Delay

End-to-end delay is the accumulation of transmission, processing, and queuing delays in routers; propagation delays in links; and end-system processing delays. For real-time conversational applications, such as VoIP, end-to-end delays smaller than 150 msecs are not perceived by a human listener; delays between 150 and 400 msecs can be acceptable but are not ideal; and delays exceeding 400 msecs can seriously hinder the interactivity in voice conversations. The receiving side of a VoIP application will typically disregard any packets that are delayed more than a certain threshold, for example, more than 400 msecs. Thus, packets that are delayed by more than the threshold are effectively lost.

Packet Jitter

A crucial component of end-to-end delay is the varying queuing delays that a packet experiences in the network's routers. Because of these varying delays, the time from when a packet is generated at the source until it is received at the receiver can fluctuate from packet to packet, as shown in Figure 7.1. This phenomenon is called jitter.

As an example, consider two consecutive packets in our VoIP application. The sender sends the second packet 20 msecs after sending the first packet. But at the receiver, the spacing between these packets can become greater than 20 msecs. To see this, suppose the first packet arrives at a nearly empty queue at a router, but just before the second packet arrives at the queue a large number of packets from other sources arrive at the same queue. Because the first packet experiences a small queuing delay and the second packet suffers a large queuing delay at this router, the first and second packets become spaced by more than 20 msecs. The spacing between consecutive packets can also become less than 20 msecs. To see this, again consider two consecutive packets. Suppose the first packet joins the end of a queue with a large number of packets, and the second packet arrives at the queue before this first packet is transmitted and before any packets from other sources arrive at the queue. In this case, our two packets find themselves one right after the other in the queue. If the time it takes to transmit a packet on the router's outbound link is less than 20 msecs, then the spacing between first and second packets becomes less than 20 msecs.

The situation is analogous to driving cars on roads. Suppose you and your friend are each driving in your own cars from San Diego to Phoenix. Suppose you and your friend have similar driving styles, and that you both drive at 100 km/hour, traffic permitting. If your friend starts out one hour before you, depending on intervening traffic, you may arrive at Phoenix more or less than one hour after your friend.

If the receiver ignores the presence of jitter and plays out chunks as soon as they arrive, then the resulting audio quality can easily become unintelligible at the receiver. Fortunately, jitter can often be removed by using sequence numbers, timestamps, and a playout delay, as discussed below.

7.3.2 Removing Jitter at the Receiver for Audio

For our VoIP application, where packets are being generated periodically, the receiver should attempt to provide periodic playout of voice chunks in the presence of random network jitter. This is typically done by combining the following two mechanisms:

- Prepending each chunk with a timestamp. The sender stamps each chunk with the time at which the chunk was generated.

- Delaying playout of chunks at the receiver. As we saw in our earlier discussion of Figure 7.1, the playout delay of the received audio chunks must be long enough so that most of the packets are received before their scheduled playout times. This playout delay can either be fixed throughout the duration of the audio session or vary adaptively during the audio session lifetime.

We now discuss how these three mechanisms, when combined, can alleviate or even eliminate the effects of jitter. We examine two playback strategies: fixed playout delay and adaptive playout delay.

Fixed Playout Delay

With the fixed-delay strategy, the receiver attempts to play out each chunk exactly q msecs after the chunk is generated. So if a chunk is timestamped at the sender at time t, the receiver plays out the chunk at time t + q, assuming the chunk has arrived by that time. Packets that arrive after their scheduled playout times are discarded and considered lost.

What is a good choice for q? VoIP can support delays up to about 400 msecs, although a more satisfying conversational experience is achieved with smaller values of q. On the other hand, if q is made much smaller than 400 msecs, then many packets may miss their scheduled playback times due to the network-induced packet jitter. Roughly speaking, if large variations in end-to-end delay are typical, it is preferable to use a large q; on the other hand, if delay is small and variations in delay are also small, it is preferable to use a small q, perhaps less than 150 msecs.

The trade-off between the playback delay and packet loss is illustrated in Figure 7.7. The figure shows the times at which packets are generated and played out for a single talk spurt. Two distinct initial playout delays are considered. As shown by the leftmost staircase, the sender generates packets at regular intervals—say, every 20 msecs. The first packet in this talk spurt is received at time r.

As shown in the figure, the arrivals of subsequent packets are not evenly spaced due to the network jitter.

For the first playout schedule, the fixed initial playout delay is set to p – r. With this schedule, the fourth packet does not arrive by its scheduled playout time, and the receiver considers it lost. For the second playout schedule, the fixed initial playout delay is set to p – r. For this schedule, all packets arrive before their scheduled playout times, and there is therefore no loss Adaptive Playout Delay

The previous example demonstrates an important delay-loss trade-off that arises when designing a playout strategy with fixed playout delays. By making the initial playout delay large, most packets will make their deadlines and there will therefore be negligible loss; however, for conversational services such as VoIP, long delays can become bothersome if not intolerable. Ideally, we would like the playout delay to be minimized subject to the constraint that the loss be below a few percent.

The natural way to deal with this trade-off is to estimate the network delay and the variance of the network delay, and to adjust the playout delay accordingly at the beginning of each talk spurt. This adaptive adjustment of playout delays at the beginning of the talk spurts will cause the sender's silent periods to be compressed and elongated; however, compression and elongation of silence by a small amount is not noticeable in speech.

Following [Ramjee 1994], we now describe a generic algorithm that the receiver can use to adaptively adjust its playout delays. To this end, let

t
i = the timestamp of the ith packet = the time the packet was generated by the sender

ri = the time packet i is received by receiver
pi = the time packet i is played at receiver
The end-to-end network delay of the ith packet is ri – t
i
. Due to network jitter,
this delay will vary from packet to packet. Let di denote an estimate of the average
network delay upon reception of the ith packet. This estimate is constructed from
the timestamps as follows:
di = (1 – u) di–1 + u (ri – t
i
)
where u is a fixed constant (for example, u = 0.01). Thus di is a smoothed average
of the observed network delays r1 – t
1, . . . , ri – t
i
. The estimate places more weight
on the recently observed network delays than on the observed network delays of the
distant past. This form of estimate should not be completely unfamiliar; a similar
idea is used to estimate round-trip times in TCP, as discussed in Chapter 3. Let vi
denote an estimate of the average deviation of the delay from the estimated average
delay. This estimate is also constructed from the timestamps:
vi = (1 – u) vi–1 + u | ri – t
i – di |
The estimates di and vi are calculated for every packet received, although they are
used only to determine the playout point for the first packet in any talk spurt.
Once having calculated these estimates, the receiver employs the following
algorithm for the playout of packets. If packet i is the first packet of a talk spurt, its
playout time, pi
, is computed as:
pi = t
i + di + Kvi
where K is a positive constant (for example, K = 4). The purpose of the Kvi term is to
set the playout time far enough into the future so that only a small fraction of the arriving
packets in the talk spurt will be lost due to late arrivals. The playout point for any
subsequent packet in a talk spurt is computed as an offset from the point in time when
the first packet in the talk spurt was played out. In particular, let
qi = pi – t
i
be the length of time from when the first packet in the talk spurt is generated until it
is played out. If packet j also belongs to this talk spurt, it is played out at time
pj = t
j + qi
The algorithm just described makes perfect sense assuming that the receiver can
tell whether a packet is the first packet in the talk spurt. This can be done by examining the
signal energy in each received packet.
7.3.3 Recovering from Packet Loss
We have discussed in some detail how a VoIP application can deal with packet jitter.
We now briefly describe several schemes that attempt to preserve acceptable audio
7.3 • VOICE-OVER-IP 617
quality in the presence of packet loss. Such schemes are called loss recovery schemes.
Here we define packet loss in a broad sense: A packet is lost either if it never arrives at

the receiver or if it arrives after its scheduled playout time. Our VoIP example will again serve as a context for describing loss recovery schemes.

As mentioned at the beginning of this section, retransmitting lost packets may not be feasible in a real-time conversational application such as VoIP. Indeed, retransmitting a packet that has missed its playout deadline serves absolutely no purpose. And retransmitting a packet that overflowed a router queue cannot normally be accomplished quickly enough. Because of these considerations, VoIP applications often use some type of loss anticipation scheme. Two types of loss anticipation schemes are forward error correction (FEC) and interleaving.

Forward Error Correction (FEC)

The basic idea of FEC is to add redundant information to the original packet stream. For the cost of marginally increasing the transmission rate, the redundant information can be used to reconstruct approximations or exact versions of some of the lost packets. Following [Bolot 1996] and [Perkins 1998], we now outline two simple FEC mechanisms. The first mechanism sends a redundant encoded chunk after every n chunks. The redundant chunk is obtained by exclusive OR-ing the n original chunks [Shacham 1990]. In this manner if any one packet of the group of n + 1 packets is lost, the receiver can fully reconstruct the lost packet. But if two or more packets in a group are lost, the receiver cannot reconstruct the lost packets. By keeping n + 1, the group size, small, a large fraction of the lost packets can be recovered when loss is not excessive. However, the smaller the group size, the greater the relative increase of the transmission rate. In particular, the transmission rate increases by a factor of 1/n, so that, if n = 3, then the transmission rate increases by 33 percent. Furthermore, this simple scheme increases the playout delay, as the receiver must wait to receive the entire group of packets before it can begin playout. For more practical details about how FEC works for multimedia transport see [RFC 5109].

The second FEC mechanism is to send a lower-resolution audio stream as the redundant information. For example, the sender might create a nominal audio stream and a corresponding low-resolution, low-bit rate audio stream. (The nominal stream could be a PCM encoding at 64 kbps, and the lower-quality stream could be a GSM encoding at 13 kbps.) The low-bit rate stream is referred to as the redundant stream. As shown in Figure 7.8, the sender constructs the nth packet by taking the nth chunk from the nominal stream and appending to it the (n – 1)st chunk from the redundant stream. In this manner, whenever there is nonconsecutive packet loss, the receiver can conceal the loss by playing out the low-bit rate encoded chunk that arrives with the subsequent packet. Of course, low-bit rate chunks give lower quality than the nominal chunks. However, a stream of mostly high-quality chunks, occasional low-quality chunks, and no missing chunks gives good overall audio quality. Note that in this scheme, the receiver only has to receive two packets before playback, so that the increased playout delay is small. Furthermore, if the low-bit rate encoding is much less than the nominal encoding, then the marginal increase in the transmission rate will be small.

In order to cope with consecutive loss, we can use a simple variation. Instead of appending just the (n – 1)st low-bit rate chunk to the nth nominal chunk, the sender can append the (n – 1)st and (n – 2)nd low-bit rate chunk, or append the (n – 1)st and (n – 3)rd low-bit rate chunk, and so on. By appending more low-bit rate chunks to each nominal chunk, the audio quality at the receiver becomes acceptable for a wider variety of harsh best-effort environments. On the other hand, the additional chunks increase the transmission bandwidth and the playout delay.

Interleaving

As an alternative to redundant transmission, a VoIP application can send interleaved audio. As shown in Figure 7.9, the sender resequences units of audio data before transmission, so that originally adjacent units are separated by a certain distance in the transmitted stream. Interleaving can mitigate the effect of packet losses. If, for example, units are 5 msecs in length and chunks are 20 msecs (that is, four units per chunk), then the first chunk could contain units 1, 5, 9, and 13; the second chunk could contain units 2, 6, 10, and 14; and so on. Figure 7.9 shows that the loss of a single packet from an interleaved stream results in multiple small gaps in the reconstructed stream, as opposed to the single large gap that would occur in a noninterleaved stream. Interleaving can significantly improve the perceived quality of an audio stream [Perkins 1998]. It also has low overhead. The obvious disadvantage of interleaving is that it increases latency. This limits its use for conversational applications such as VoIP, although it can perform well for streaming stored audio. A major advantage of interleaving is that it does not increase the bandwidth requirements of a stream.

Alright, let's turn 7.3 into something you can just read once, understand, and walk into the exam with.

I'll cover:

1. **Best-effort IP & QoS issues**: packet loss, delay, jitter
2. **Removing jitter at the receiver**: fixed and adaptive playout delay
3. **Recovering from packet loss**: FEC and interleaving

All in the context of **VoIP** (real-time voice over IP).

---

1. VoIP and the Problem: Best-Effort IP

VoIP (Voice-over-IP) = real-time conversational audio over the Internet.
From the user's perspective: similar to a phone call, but running over the IP network.

The core issue:

> IP is **best-effort**. It **tries** to deliver packets quickly, but it **does not guarantee**:
>
> - Maximum delay
> - Minimum bandwidth
> - Maximum packet loss rate

For **real-time voice**, though, we deeply care about:

- **Packet loss**
- **End-to-end delay**
- **Jitter** (variation in delay)

So all of 7.3 is basically:

> “The network gives us a crappy, best-effort pipe. How can **the application** still make VoIP usable?”

And importantly:
They want **application-layer techniques**. No changes to IP, no changes to routers.

---

2. Concrete VoIP Example They Use

To make things concrete, the book picks a specific setup:

- Sender generates **8,000 bytes per second** (this is typical audio)
- Every **20 ms**, the sender collects the audio generated in that interval → one **chunk**
- 20 ms × 8,000 bytes/s = **160 bytes per chunk**
- Each chunk + header → fits in **one UDP segment**
- So:
    - One UDP packet every **20 ms**
    - Each packet carries 160 bytes of encoded audio

If the network were perfect:

- Each packet arrives exactly 20 ms after the previous one
- Receiver could simply **play each chunk as soon as it arrives**

In reality:

- Some packets are **lost**
- Delays vary (jitter)
- Some packets arrive too late to be useful

The receiver must decide:

1. **When** to play each chunk (this is about delay & jitter)
2. **What to do** if a chunk is missing (this is about packet loss)

---

3. Packet Loss

What is packet loss here?

VoIP uses **UDP** most of the time:

- UDP packets are encapsulated in IP datagrams
- On the path from sender to receiver, each IP packet may:
    - Wait in router queues
    - Get **dropped** if a buffer is full

- If dropped, that audio chunk **never reaches the receiver**

Also, note:
In VoIP, a packet is effectively “lost” if:

- It never arrives, **or**
- It arrives **after its playout deadline** (too late to be used)

So effective packet loss = true loss + packets that came too late.

Why not just use TCP?

TCP would **retransmit** lost segments and guarantee reliability. That sounds nice, but:

1. **Retransmission delay**
    - If a packet is late (e.g., arrives 500 ms after it was sent), it is **useless** for real-time audio.
    - Waiting for retransmissions makes **interactive conversation laggy or broken**.
2. **TCP congestion control**
    - When TCP detects loss, it **reduces its sending rate**.
    - That rate might become **less than** the receiver’s audio consumption rate.
    - Result: the receiver’s audio buffer **starves** → gaps in playback.

Because of this:

> Most VoIP implementations use **UDP**, not TCP.
> Skype, for example, uses UDP by default unless UDP is blocked by NAT/firewall.

How bad is packet loss for voice?

Surprisingly:
Voice can tolerate **some** packet loss:

- Around **1–20% loss** can be acceptable depending on:
    - Codec (how the voice is encoded)
    - Loss concealment methods (e.g. FEC, interpolation)

But if loss > 10–20% (e.g., a really bad wireless link), no clever trick will fully fix the audio.

---

4. End-to-End Delay

End-to-end delay includes:

- Transmission delay
- Propagation delay

- Queueing delay in routers
- Processing at end systems

For **conversational voice**, human perception gives us thresholds:

- **< 150 ms**: essentially not noticeable
- **150–400 ms**: still usable, but you feel a bit of lag
- **> 400 ms**: conversation feels very awkward or nearly unusable

Because of this:

- The receiver will **discard** any packet whose **delay exceeds some threshold** (e.g., 400 ms). That packet is effectively treated as lost.

So high delay not only hurts interactivity, it also **increases effective loss**.

---

5. Packet Jitter (Delay Variation)

Jitter = variation in the packet arrival times.

Even with light congestion, queueing in routers fluctuates:

- One packet may encounter an almost empty queue → small delay
- The next packet may hit a long queue → much larger delay

In our VoIP example:

- Sender sends packet every **20 ms**
- At receiver, the spacing between packets might be:
  - Sometimes > 20 ms
  - Sometimes < 20 ms

So the packets **do not arrive periodically** even though they are **sent periodically**.

If the receiver:

- Just plays chunks **immediately when they arrive**, then:
  - Audio will speed up / slow down / distort badly
  - It might become **unintelligible**

So we need a mechanism at the receiver to **smooth out jitter**.

---

6. Removing Jitter at the Receiver

Key idea:

> Sacrifice a bit of delay to get **smooth, periodic playback**.

Two tools are used:

1. **Timestamps**
    - Sender tags each chunk with the **generation time** t_i
2. **Playout delay at receiver**
    - Receiver **waits a bit** before playing packets
    - This delay must be **long enough** so most packets arrive in time
    - That delay can be:
        - **Fixed** (same throughout call)
        - **Adaptive** (adjusted

6.1 Fixed Playout Delay

Strategy:

- Let q = fixed playout delay.
- If packet has timestamp t, the receiver plays it at time:

$$t + q$$

- Packets arriving after their scheduled time $t + q$ are **discarded** as late (lost).

How to choose q?

- Larger q:
    - More packets arrive before their playout time → **fewer late losses**
    - But overall delay is higher → conversation feels laggier
- Smaller q:
    - Lower delay (good)
    - More packets will arrive too late → more effective loss (bad)

So there is a **trade-off**:

- Networks with:
    - Big delay variation → need larger q
    - Small variation → can use smaller q

Typical constraint:

- Want q less than about 400 ms, ideally around or under 150 ms if possible.

The book's figure 7.7:

- Shows two different fixed playout delays:
    - Smaller delay → one packet arrives too late → considered lost

  - Slightly larger delay → all packets arrive in time → no loss
- But that larger delay means more latency.

6.2 Adaptive Playout Delay

Fixed q is rigid. But network conditions change over time.

Better idea:

> **Estimate** the current network delay & variability, and adjust playout delay per **talk spurt**.

A **talk spurt** = a continuous period of speech (no silence gap).
Silent gaps between words/sentences separate talk spurts.

Why is this useful?

- Humans don't notice small stretching/compressing of **silence**, but they do care about speech timing.
- So we can **adjust playout timing at the start of each talk spurt** without sounding weird.

Variables

For packet i:

- t_i: timestamp (when generated at sender)
- r_i: receive time at receiver
- End-to-end delay for packet i: $r_i - t_i$

We maintain two estimates:

1. **Average delay** d_i

$$d_i = (1-u)d_{i-1} + u(r_i - t_i)$$

   - This is an **exponential weighted moving average**
   - u is a small constant, e.g. 0.01
   - Heavier weight on recent delays

2. **Average deviation** v_i from the mean:

$$v_i = (1-u)v_{i-1} + u\big|(r_i - t_i) - d_i\big|$$

   - Measures how much actual delay fluctuates around the estimated average
   - Essentially, an estimate of jitter magnitude

We compute these **for every packet**, but they are **used primarily at the start of a talk spurt**.

Choosing playout time for first packet of a talk spurt

If packet i is the **first packet of a talk spurt**, we set:

$$p_i = t_i + d_i + Kv_i$$

Where:

- p_i: playout time for packet i
- d_i: estimated average network delay
- v_i: estimated average deviation (jitter)
- K: safety factor (e.g., 4)

Intuition:

- $t_i + d_i$: expected arrival time
- Kv_i: extra cushion to absorb jitter
- This tries to ensure that **most** packets in this talk spurt arrive before their scheduled playout times.

Then define:

$$q_i = p_i - t_i$$

This is the playout delay for the **first packet** of that talk spurt.

For any **other packet** j in the **same talk spurt**, playout time is:

$$p_j = t_j + q_i$$

So:

- All packets in that talk spurt share the **same effective playout delay**
- But each talk spurt may have a different q_i, adjusting to changing network conditions

Detecting “first packet in talk spurt”:

- Receiver can inspect **signal energy**:
  - Silence vs speech
  - Start of speech → new talk spurt

---

7. Recovering from Packet Loss (FEC & Interleaving)

Now suppose some packets are just **lost** (or arrive too late).

We do **not** want to use retransmission (too slow).
Instead, we use **loss anticipation schemes**:

1. **Forward Error Correction (FEC)** – send **extra redundancy**
2. **Interleaving** – rearrange packet contents to **spread loss out**

Remember: “packet loss” now means:

- Either never arrived

- Or arrived after its playout deadline

7.1 Forward Error Correction (FEC)

Goal:

> Add a little extra data so that **some lost information can be recovered** without retransmission.

Two simple mechanisms:

(A) XOR-based FEC over groups of packets

Mechanism:

- Group **n** consecutive original chunks into a block.
- Compute an extra **redundant chunk** by XOR-ing all n chunks:

$$R = C_1 \oplus C_2 \oplus \cdots \oplus C_n$$

- Send all **n + 1** packets: the n original chunks + the redundant one.

Property:

- If **any one** of the n + 1 packets is lost:
  - The receiver can reconstruct the missing one by XOR-ing the others.

Example (n = 3):

- Packets: C1, C2, C3, R (where R = C1⊕C2⊕C3)
- If C2 is lost:

$$C_2 = C_1 \oplus C_3 \oplus R$$

- So fully recoverable.

Trade-offs:

- **Pros**:
  - Can recover many lost packets if loss is **isolated** (non-bursty)
  - Simple and exact recovery when it works
- **Cons**:
  - If **two or more** packets in the group are lost → cannot fully reconstruct.
  - Transmission rate **increases**:
    - We send 1 extra packet for every n original packets
    - Overhead = **1/n**

    - Example: n = 3 → one redundant per 3 originals → +33% bandwidth
  - Increases **playout delay**:
    - Receiver must wait until **all n+1 packets** in the group are received before safely correcting and playing.

So XOR-FEC works well when:

- Loss rate is not too high
- Losses are mostly **single-packet** rather than bursty

(B) Redundant low-bitrate stream (multi-description style)

Idea:

- Maintain **two** encodings of the audio:
  1. Nominal (high-quality) stream, e.g., PCM at 64 kbps
  2. Redundant (low-quality, low-bitrate) stream, e.g., GSM at 13 kbps

Construction (simplified version from book):

- For packet n, send:
  - **nth high-quality chunk** (from full-rate stream)
  - **(n−1)th low-quality chunk** (from low-bitrate stream) appended to it

So packet n contains:

- HQ(n) + LQ(n−1)

If packet n−1 is lost:

- You still have LQ(n−1) inside packet n → can play a **lower-quality version** instead of silence.

Benefits:

- You still mostly play **high-quality audio**.
- When a packet is lost:
  - Instead of a gap, you can play a **low-quality backup version**.
- Only need to receive **two packets** (n−1 and n) to recover, so playout delay increase is **small**.
- If low-bitrate encoding is small, bandwidth overhead is modest.

You can extend this idea:

- Append LQ(n−1) and LQ(n−2), etc., to handle **consecutive losses**.
- But that increases:
  - Bandwidth overhead
  - Possibly playout delay

So there's a **quality vs bandwidth vs robustness** tradeoff.

7.2 Interleaving

Interleaving is a different idea:

> Rearrange audio units across packets so that **losing one packet** results in **many tiny gaps**, not one big gap.

How it works:

- Let the raw audio be divided into **small units** (e.g. 5 ms each).
- A chunk (packet) carries several units, say **4 units** → **20 ms**.
- Instead of putting **units 1,2,3,4** in packet 1, **5,6,7,8** in packet 2, etc., we:

Example:

- Packet 1: units 1, 5, 9, 13
- Packet 2: units 2, 6, 10, 14
- Packet 3: units 3, 7, 11, 15
- Packet 4: units 4, 8, 12, 16

Now suppose **packet 2 is lost**:

- Units lost: 2, 6, 10, 14
- These correspond to **many small 5 ms gaps** spaced apart in the reconstructed stream.

Perceived effect:

- Instead of a **single 20 ms hole**, you get multiple **tiny 5 ms "holes"** scattered in time.
- Human hearing can often **tolerate or mentally fill in** these small gaps much better.

Pros:

- Can significantly improve perceived quality even **without extra bandwidth**.
- No redundant information is added → bandwidth is unchanged.

Cons:

- Introduces **extra delay**:
  - You must buffer some data to reorder the units back into proper time order at the receiver.
- For **conversational voice (VoIP)**, additional delay may be problematic.
- Works better for **streaming stored audio** (not interactive): you can afford more buffering.

So:

- Interleaving = good for stored streaming (e.g., music), maybe less ideal for tight real-time VoIP because of extra latency.

Final Mental Map (for the exam)

- **Best-effort IP**:
  - No guarantees on delay, delay variation (jitter), or loss.
  - VoIP is sensitive to all three.
- **Packet loss**:
  - Dropped or too-late packets.
  - TCP not ideal (retransmissions + congestion control → big delays, buffer starvation).
  - 1–20% loss can be tolerable with good concealment.
- **Delay constraints**:
  - <150 ms: ideal
  - 150–400 ms: acceptable
  - 400 ms: bad
- **Jitter**:
  - Variable delay due to queueing.
  - Fixed playout delay: tradeoff between delay and late loss.
  - Adaptive playout delay: estimate delay + jitter (d_i, v_i), compute playout time with $p_i = t_i + d_i + Kv_i$ per talk spurt.
- **Loss recovery**:
  - **FEC**:
    - XOR over groups of n chunks (extra 1/n bandwidth; can recover 1 lost packet per group; increased delay).
    - Redundant low-bitrate stream appended to each packet (higher robustness with modest overhead).
  - **Interleaving**:
    - Rearrange small audio units across packets so loss → many mini-gaps instead of one big gap.
    - No extra bandwidth, but increases latency.

If you want, next we can do:

- A **question set** where I give you short VoIP/jitter/loss setups and you tell me what the receiver should do and which scheme helps, or
- A **compact one-page LaTeX summary** of 7.3 with formulas and definitions only.

7.4: RTP (basics, fields)

7.4 Protocols for Real-Time Conversational
Applications
Real-time conversational applications, including VoIP and video conferencing, are
compelling and very popular. It is therefore not surprising that standards bodies,
such as the IETF and ITU, have been busy for many years (and continue to be
busy!) at hammering out standards for this class of applications. With the appropriate standards
in place for real-time conversational applications, independent companies are creating new
products that interoperate with each other. In this section we
examine RTP and SIP for real-time conversational applications. Both standards are
enjoying widespread implementation in industry products.
-
-
-
- -
-
-
7.4 • PROTOCOLS FOR REAL-TIME CONVERSATIONAL APPLICATIONS 623
7.4.1 RTP
In the previous section, we learned that the sender side of a VoIP application appends
header fields to the audio chunks before passing them to the transport layer. These
header fields include sequence numbers and timestamps. Since most multimedia
networking applications can make use of sequence numbers and timestamps, it is convenient to
have a standardized packet structure that includes fields for audio/video data,
sequence number, and timestamp, as well as other potentially useful fields. RTP,
defined in RFC 3550, is such a standard. RTP can be used for transporting common
formats such as PCM, ACC, and MP3 for sound and MPEG and H.263 for video. It can
also be used for transporting proprietary sound and video formats. Today, RTP enjoys
widespread implementation in many products and research prototypes. It is also
complementary to other important real-time interactive protocols, such as SIP.
In this section, we provide an introduction to RTP. We also encourage you to
visit Henning Schulzrinne's RTP site [Schulzrinne-RTP 2012], which provides a
wealth of information on the subject. Also, you may want to visit the RAT site
[RAT 2012], which documents VoIP application that uses RTP.
RTP Basics
RTP typically runs on top of UDP. The sending side encapsulates a media chunk
within an RTP packet, then encapsulates the packet in a UDP segment, and then
hands the segment to IP. The receiving side extracts the RTP packet from the UDP
segment, then extracts the media chunk from the RTP packet, and then passes the
chunk to the media player for decoding and rendering.
As an example, consider the use of RTP to transport voice. Suppose the voice
source is PCM-encoded (that is, sampled, quantized, and digitized) at 64 kbps. Further suppose
that the application collects the encoded data in 20-msec chunks, that is,
160 bytes in a chunk. The sending side precedes each chunk of the audio data with
an RTP header that includes the type of audio encoding, a sequence number, and a
timestamp. The RTP header is normally 12 bytes. The audio chunk along with the
RTP header form the RTP packet. The RTP packet is then sent into the UDP socket

interface. At the receiver side, the application receives the RTP packet from its socket interface. The application extracts the audio chunk from the RTP packet and uses the header fields of the RTP packet to properly decode and play back the audio chunk.

If an application incorporates RTP—instead of a proprietary scheme to provide payload type, sequence numbers, or timestamps—then the application will more easily interoperate with other networked multimedia applications. For example, if two different companies develop VoIP software and they both incorporate RTP into their product, there may be some hope that a user using one of the VoIP products will be able to communicate with a user using the other VoIP product. In Section 7.4.2, we'll see that RTP is often used in conjunction with SIP, an important standard for Internet telephony.

It should be emphasized that RTP does not provide any mechanism to ensure timely delivery of data or provide other quality-of-service (QoS) guarantees; it does not even guarantee delivery of packets or prevent out-of-order delivery of packets. Indeed, RTP encapsulation is seen only at the end systems. Routers do not distinguish between IP datagrams that carry RTP packets and IP datagrams that don't.

624 CHAPTER 7 • MULTIMEDIA NETWORKING

RTP allows each source (for example, a camera or a microphone) to be assigned its own independent RTP stream of packets. For example, for a video conference between two participants, four RTP streams could be opened—two streams for transmitting the audio (one in each direction) and two streams for transmitting the video (again, one in each direction). However, many popular encoding techniques—including MPEG 1 and MPEG 2—bundle the audio and video into a single stream during the encoding process. When the audio and video are bundled by the encoder, then only one RTP stream is generated in each direction.

RTP packets are not limited to unicast applications. They can also be sent over one-to-many and many-to-many multicast trees. For a many-to-many multicast session, all of the session's senders and sources typically use the same multicast group for sending their RTP streams. RTP multicast streams belonging together, such as audio and video streams emanating from multiple senders in a video conference application, belong to an RTP session.

RTP Packet Header Fields

As shown in Figure 7.11, the four main RTP packet header fields are the payload type, sequence number, timestamp, and source identifier fields.

The payload type field in the RTP packet is 7 bits long. For an audio stream, the payload type field is used to indicate the type of audio encoding (for example, PCM, adaptive delta modulation, linear predictive encoding) that is being used. If a sender decides to change the encoding in the middle of a session, the sender can inform the receiver of the change through this payload type field. The sender may want to change the encoding in order to increase the audio quality or to decrease the RTP stream bit rate. Table 7.2 lists some of the audio payload types currently supported by RTP.

For a video stream, the payload type is used to indicate the type of video encoding (for example, motion JPEG, MPEG 1, MPEG 2, H.261). Again, the sender can change video encoding on the fly during a session. Table 7.3 lists some of the video payload types currently supported by RTP. The other important fields are the following:

• Sequence number field. The sequence number field is 16 bits long. The sequence number increments by one for each RTP packet sent, and may be used by the

Payload
type
Sequence
number

Synchronization
source identifier
Miscellaneous
fields Timestamp
Figure 7.11 ␂ RTP header fields
7.4 • PROTOCOLS FOR REAL-TIME CONVERSATIONAL APPLICATIONS 625
receiver to detect packet loss and to restore packet sequence. For example, if the
receiver side of the application receives a stream of RTP packets with a gap
between sequence numbers 86 and 89, then the receiver knows that packets 87
and 88 are missing. The receiver can then attempt to conceal the lost data.
• Timestamp field. The timestamp field is 32 bits long. It reflects the sampling
instant of the first byte in the RTP data packet. As we saw in the preceding
section, the receiver can use timestamps to remove packet jitter introduced in
the network and to provide synchronous playout at the receiver. The time␂stamp is derived from
a sampling clock at the sender. As an example, for
audio the timestamp clock increments by one for each sampling period (for
example, each 125 ␃sec for an 8 kHz sampling clock); if the audio applica␂tion generates chunks
consisting of 160 encoded samples, then the timestamp
increases by 160 for each RTP packet when the source is active. The
626 CHAPTER 7 • MULTIMEDIA NETWORKING
Payload-Type Number Audio Format Sampling Rate Rate
0 PCM ␃-law 8 kHz 64 kbps
1 1016 8 kHz 4.8 kbps
3 GSM 8 kHz 13 kbps
7 LPC 8 kHz 2.4 kbps
9 G.722 16 kHz 48–64 kbps
14 MPEG Audio 90 kHz —
15 G.728 8 kHz 16 kbps
Table 7.2 ␂ Audio payload types supported by RTP
Payload-Type Number Video Format
26 Motion JPEG
31 H.261
32 MPEG 1 video
33 MPEG 2 video
Table 7.3 ␂ Some video payload types supported by RTP
timestamp clock continues to increase at a constant rate even if the source is
inactive.
• Synchronization source identifier (SSRC). The SSRC field is 32 bits long. It iden␂tifies the source
of the RTP stream. Typically, each stream in an RTP session has
a distinct SSRC. The SSRC is not the IP address of the sender, but instead is a
number that the source assigns randomly when the new stream is started. The
probability that two streams get assigned the same SSRC is very small. Should
this happen, the two sources pick a new SSRC value

Nice, this is a compact section – we can chew through all the important RTP basics + header fields in one clean pass.

I'll break 7.4.1 into:

1. What RTP is and why it exists
2. How RTP runs over UDP and is used in VoIP
3. Multiple RTP streams and sessions
4. The **main header fields**: payload type, sequence number, timestamp, SSRC

---

1. What is RTP and why do we need it?

From 7.3 you already saw that **real-time media** (voice/video) needs:

- **Sequence numbers** → to detect loss/order packets
- **Timestamps** → to fix jitter and schedule playout
- Info about **which codec/format** is used

You *could* invent your own proprietary packet format for each application, but then:

- Different vendors' systems wouldn't interoperate
- Every app would reinvent the same wheel (seq#, timestamps, type info)

So the IETF defined **RTP (Real-time Transport Protocol, RFC 3550)**:

> A **standardized packet format** for carrying real-time audio/video over the network.

Key points about RTP:

- It is **not** a transport protocol in the "TCP vs UDP" sense.
- It is a **layer above UDP**:
  - App data (audio/video chunk) → put into an RTP packet
  - RTP packet → put into a UDP segment
  - UDP → IP → network

RTP provides:

- Common header structure
- Fields for:
  - **Payload type** (which codec/format)
  - **Sequence number**
  - **Timestamp**
  - **Source identifier** (SSRC)

RTP is used for many media formats:

- Audio: PCM, MP3, AAC, GSM, etc.
- Video: Motion JPEG, MPEG-1, MPEG-2, H.261, etc.
- Also works with proprietary encodings.

Because RTP is standardized, if:

- Company A and Company B both implement RTP,
- Their apps **have a chance of talking to each other**, especially when combined with SIP (signaling) for call setup.

---

2. How RTP runs over UDP (example with VoIP)

Typical usage in VoIP:

1. **Audio encoding**
   - Example from book:
     - PCM at 64 kbps
     - Collect **20 ms** worth of samples into a chunk
     - 64 kbps = 64,000 bits/sec = 8,000 bytes/sec
     - 20 ms = 0.02 s → 0.02 × 8000 = **160 bytes per chunk**
2. **RTP encapsulation**
   - For each 160-byte chunk:
     - Prepend a **12-byte RTP header**
     - This header contains payload type, sequence number, timestamp, SSRC, etc.
   - The result is an **RTP packet**.
3. **UDP & IP**
   - RTP packet is placed in the payload of a **UDP segment**.
   - UDP segment is encapsulated in an IP datagram.
   - Sent over the network.
4. **At the receiver**
   - Application gets UDP data from its socket.
   - Extracts the RTP header and the audio payload.
   - Uses:
     - **Sequence number** → detect missing or out-of-order packets.
     - **Timestamp** → schedule playout to remove jitter.
     - **Payload type** → know which codec to use to decode.
   - Decodes and plays the audio.

So you can think of it as:

$$\textbf{audio/video chunk} \longrightarrow \text{[RTP header + media]} \longrightarrow \text{UDP} \longrightarrow \text{IP}$$

---

3. What RTP does *not* do (important for exam)

RTP **does not**:

- Guarantee timely delivery
- Guarantee in-order delivery
- Guarantee *any* quality-of-service (QoS)
- Perform retransmissions

Routers:

- Do **not** treat IP packets with RTP differently from any other IP packets.
- Most of the time, routers don't even know these are RTP packets.

All QoS tricks (buffering, FEC, interleaving, adaptive playout) are done **at the endpoints**, not inside RTP itself.

RTP is just a **packetization format** that gives your application useful fields.

---

4. Multiple RTP streams and sessions

RTP is designed to handle multiple **sources** and **media types**.

Examples:

- A 2-person video call has **4 RTP streams**:
    - Audio A→B
    - Audio B→A
    - Video A→B
    - Video B→A

Each source (e.g., A's microphone, A's camera) can have its **own RTP stream**.

In some encodings (e.g., many MPEG variants):

- Audio and video are **multiplexed together** into a single stream by the codec.
- In that case, you might just have **1 RTP stream per direction**.

RTP can also be used over **multicast**:

- One-to-many: a live lecture broadcast
- Many-to-many: several participants sending to the same multicast group in a conference

A group of related RTP streams (e.g., all streams in a conference) is called an **RTP session**.

---

5. RTP Packet Header – Main Fields

The RTP header is shown (schematically) as:

> **[Payload type | Sequence number | Timestamp | SSRC | misc fields]**

We'll focus on the four key ones: **payload type, sequence number, timestamp, SSRC**.

5.1 Payload Type (PT) – 7 bits

- Tells the receiver **what format the payload uses**.

For **audio**:

- PT might indicate:
  - PCM µ-law
  - GSM
  - LPC
  - G.722
  - MP3 (MPEG audio)
  - etc.

For **video**:

- PT might indicate:
  - Motion JPEG
  - H.261
  - MPEG-1 video
  - MPEG-2 video
  - etc.

Why is this important?

- The receiver needs to know **which decoder** to use.
- During a session, the sender can **change codecs on the fly**:
  - e.g., Switch from a high-bitrate codec to a lower-bitrate codec if bandwidth drops.
  - This change is signaled by changing the **payload type value** in the RTP header.

There are:

- Some **standardized static mappings** (e.g., PT 0 = PCM µ-law at 8 kHz).
- Often **dynamic mappings** negotiated at session setup (via SIP/SDP).

But for CS232, the key idea is:

> PT = "what kind of encoding is this payload using right now?"

5.2 Sequence Number – 16 bits

- Increments by **1 for each RTP packet sent** in a stream.
- Initial value is chosen randomly for security/robustness reasons.

At the receiver, sequence numbers are used to:

1. **Detect losses**
    - If you see … 85, 86, 89, 90 …
      → you know 87 and 88 are missing.
2. **Restore order**
    - Packets may arrive out-of-order due to routing or queueing.
    - Receiver can reorder by sequence number before decoding/playing.

Sequence numbers are **per RTP stream** (distinguished by SSRC).

5.3 Timestamp – 32 bits

- Reflects the **sampling instant of the first byte** in the RTP payload.
- It is derived from a **sampling clock at the sender**.

For audio:

- Suppose sampling rate = 8 kHz (8000 samples/sec)
- The timestamp increments by **1 for each sampling period**.
    - One sample every 125 μs → timestamp +1 every 125 μs.
- If each RTP packet contains 160 samples (20 ms of audio):
    - Timestamp increases by **160** per packet during active speech.

Important points:

- Even if the source is **silent**, the timestamp clock usually **keeps running** steadily.
  (So that timing remains consistent.)
- The receiver uses timestamps to:
    - Compensate for **jitter** (variable delay)
    - Schedule smooth **playout** (as in 7.3's jitter removal discussion)
    - Synchronize **audio and video** streams (if both are present)

You saw earlier in 7.3:

- Playout time = timestamp + chosen playout delay (fixed or adaptive).
- Timestamps are the basis for those computations.

5.4 SSRC – Synchronization Source Identifier (32 bits)

- SSRC = **Synchronization Source** identifier.
- It uniquely identifies **one RTP source** (e.g., one microphone or one camera) within an RTP session.

Properties:

- 32-bit field; chosen **randomly** by the source when it starts a new stream.
- The chance of collision is low; if a collision is detected, sources pick new SSRC values.
- It is *not* an IP address; it's an **RTP-level identifier**.

Why do we need SSRC?

- In a conference call with multiple senders:
  - You may receive multiple RTP streams on the same IP/port combination.
  - You need to know **which stream belongs to which participant/source**.
- SSRC distinguishes different streams so the application can:
  - Play each participant's audio separately
  - Sync a particular user's audio and video
  - Group related streams (e.g., audio + video from the same user) together

---

Quick exam-style summary

If you need a short "gist" to write beside §7.4.1 in your notes:

> **RTP**: A standard packet format for real-time audio/video over UDP. Carries payload type (codec), sequence number (loss/order), timestamp (jitter removal & sync), and SSRC (stream identity). Runs above UDP, does *not* provide reliability or QoS, and is processed only at end systems—not by routers.

If you'd like, I can now:

- Turn this into a **small table of the four fields + what they're used for**, or
- Give you a few **mini conceptual questions** (like "what happens if the payload type changes mid-call?") to test yourself on RTP.

7.5: dimensioning, multiple classes of service (motivation, scheduling, policing), Per-connection QoS

7.5 Network Support for Multimedia
In Sections 7.2 through 7.4, we learned how application-level mechanisms such as
client buffering, prefetching, adapting media quality to available bandwidth, adaptive playout,
and loss mitigation techniques can be used by multimedia applications
632 CHAPTER 7 • MULTIMEDIA NETWORKING
9

5
6
4
7
2
3
1
8
SIP registrar
upenn.edu
SIP proxy
umass.edu
SIP client
217.123.56.89
SIP client
197.87.54.21
SIP registrar
eurcom.fr
Figure 7.13 Session initiation, involving SIP proxies and registrars

to improve a multimedia application's performance. We also learned how content distribution networks and P2P overlay networks can be used to provide a system-level approach for delivering multimedia content. These techniques and approaches are all designed to be used in today's best-effort Internet. Indeed, they are in use today precisely because the Internet provides only a single, best-effort class of service. But as designers of computer networks, we can't help but ask whether the network (rather than the applications or application-level infrastructure alone) might provide mechanisms to support multimedia content delivery. As we'll see shortly, the answer is, of course, "yes"! But we'll also see that a number of these new network-level mechanisms have yet to be widely deployed. This may be due to their complexity and to the fact that application-level techniques together with best-effort service and properly dimensioned network resources (for example, bandwidth) can indeed provide a "good-enough" (even if not-always-perfect) end-to-end multimedia delivery service.

Table 7.4 summarizes three broad approaches towards providing network-level support for multimedia applications.

• Making the best of best-effort service. The application-level mechanisms and infrastructure that we studied in Sections 7.2 through 7.4 can be successfully used in a well-dimensioned network where packet loss and excessive end-to-end

7.5 • NETWORK SUPPORT FOR MULTIMEDIA 633

Approach Granularity Guarantee Mechanisms Complexity Deployment to date
Making the all traffic none, or application- minimal everywhere
best of best- treated soft layer support,
effort service. equally CDNs, overlays,
network-level
resource
provisioning
Differentiated different none, packet marking, medium some
service classes of or soft policing,
traffic scheduling
treated
differently

Per-connection each soft or hard, packet marking, light little
Quality-of- source- once flow policing, scheduling;
Service (QoS) destination is admitted call admission and
Guarantees flows treated signaling
differently

Table 7.4 Three network-level approaches to supporting multimedia applications

delay rarely occur. When demand increases are forecasted, the ISPs deploy additional bandwidth and switching capacity to continue to ensure satisfactory delay and packet-loss performance [Huang 2005]. We'll discuss such network dimensioning further in Section 7.5.1.

- Differentiated service. Since the early days of the Internet, it's been envisioned that different types of traffic (for example, as indicated in the Type-of-Service field in the IP4v packet header) could be provided with different classes of service, rather than a single one-size-fits-all best-effort service. With differentiated service, one type of traffic might be given strict priority over another class of traffic when both types of traffic are queued at a router. For example, packets belonging to a real-time conversational application might be given priority over other packets due to their stringent delay constraints. Introducing differentiated service into the network will require new mechanisms for packet marking (indicating a packet's class of service), packet scheduling, and more. We'll cover differentiated service, and new network mechanisms needed to implement this service, in Section 7.5.2.
- Per-connection Quality-of-Service (QoS) Guarantees. With per-connection QoS guarantees, each instance of an application explicitly reserves end-to-end bandwidth and thus has a guaranteed end-to-end performance. A hard guarantee means the application will receive its requested quality of service (QoS) with certainty. A soft guarantee means the application will receive its requested quality of service with high probability. For example, if a user wants to make a VoIP call from Host A to Host B, the user's VoIP application reserves bandwidth explicitly in each link along a route between the two hosts. But permitting applications to make reservations and requiring the network to honor the reservations requires some big changes. First, we need a protocol that, on behalf of the applications, reserves link bandwidth on the paths from the senders to their receivers. Second, we'll need new scheduling policies in the router queues so that per-connection bandwidth reservations can be honored. Finally, in order to make a reservation, the applications must give the network a description of the traffic that they intend to send into the network and the network will need to police each application's traffic to make sure that it abides by that description. These mechanisms, when combined, require new and complex software in hosts and routers. Because per-connection QoS guaranteed service has not seen significant deployment, we'll cover these mechanisms only briefly in Section 7.5.3.

7.5.1 Dimensioning Best-Effort Networks

Fundamentally, the difficulty in supporting multimedia applications arises from their stringent performance requirements––low end-to-end packet delay, delay

634 CHAPTER 7 • MULTIMEDIA NETWORKING

jitter, and loss—and the fact that packet delay, delay jitter, and loss occur whenever the network becomes congested. A first approach to improving the quality of multimedia applications—an approach that can often be used to solve just about any problem where resources are constrained—is simply to "throw money at the problem" and thus simply avoid resource contention. In the case of networked multimedia, this means providing enough link capacity throughout the network so that network congestion, and its consequent packet delay and loss, never (or only very rarely) occurs. With enough link capacity, packets could zip through today's Internet without queuing delay or loss. From many perspectives this is an ideal situation—multimedia applications would perform perfectly, users would be happy, and this could all be achieved with no changes to Internet's besteffort architecture.

The question, of course, is how much capacity is "enough" to achieve this nirvana, and whether the costs of providing "enough" bandwidth are practical from a business standpoint to the ISPs. The question of how much capacity to provide at network links in a given topology to achieve a given level of performance is often known as bandwidth provisioning. The even more complicated problem of how to design a network topology (where to place routers, how to interconnect routers with links, and what capacity to assign to links) to achieve a given level of end-to-end performance is a network design problem often referred to as network dimensioning. Both bandwidth provisioning and network dimensioning are complex topics, well beyond the scope of this textbook. We note here, however, that the following issues must be addressed in order to predict application-level performance between two network end points, and thus provision enough capacity to meet an application's performance requirements.

- Models of traffic demand between network end points. Models may need to be specified at both the call level (for example, users "arriving" to the network and starting up end-to-end applications) and at the packet level (for example, packets being generated by ongoing applications). Note that workload may change over time.
- Well-defined performance requirements. For example, a performance requirement for supporting delay-sensitive traffic, such as a conversational multimedia application, might be that the probability that the end-to-end delay of the packet is greater than a maximum tolerable delay be less than some small value [Fraleigh 2003].
- Models to predict end-to-end performance for a given workload model, and techniques to find a minimal cost bandwidth allocation that will result in all user requirements being met. Here, researchers are busy developing performance models that can quantify performance for a given workload, and optimization techniques to find minimal-cost bandwidth allocations meeting performance requirements.

Given that today's best-effort Internet could (from a technology standpoint) support multimedia traffic at an appropriate performance level if it were dimensioned to do so, the natural question is why today's Internet doesn't do so. The answers are primarily economic and organizational. From an economic standpoint, would users be willing to pay their ISPs enough for the ISPs to install sufficient bandwidth to support multimedia applications over a best-effort Internet? The organizational issues are perhaps even more daunting. Note that an end-to-end path between two multimedia end points will pass through the networks of multiple ISPs. From an organizational standpoint, would these ISPs be willing to cooperate (perhaps with revenue sharing) to ensure that the end-to-end path is properly dimensioned to support multimedia applications? For a perspective on these economic and organizational issues, see [Davies 2005]. For a perspective on provisioning tier-1 backbone networks to support delay-sensitive traffic, see [Fraleigh 2003].

7.5.2 Providing Multiple Classes of Service

Perhaps the simplest enhancement to the one-size-fits-all best-effort service in today's Internet is to divide traffic into classes, and provide different levels of service to these different classes of traffic. For example, an ISP might well want to provide a higher class of service to delay-sensitive Voice-over-IP or teleconferencing traffic (and charge more for this service!) than to elastic traffic such as email or HTTP. Alternatively, an ISP may simply want to provide a higher quality of service to customers willing to pay more for this improved service. A number of residential wired-access ISPs and cellular wireless-access ISPs have adopted such tiered levels of service—with platinum-service subscribers receiving better performance than gold- or silver-service subscribers.

We're all familiar with different classes of service from our everyday lives—first-class airline passengers get better service than business-class passengers, who in turn get better service than those of us who fly economy class; VIPs are provided immediate entry to events while everyone else waits in line; elders are revered in some countries and provided seats of honor and the finest food at a table. It's important to note that such differential service is provided among aggregates of traffic, that is, among classes of traffic, not among individual connections. For example, all first-class passengers are handled the same (with no first-class passenger receiving any better treatment than any other first-class passenger), just as all VoIP packets would receive the same treatment within the network, independent of the particular end-to-end connection to which they belong. As we will see, by dealing with a small number of traffic aggregates, rather than a large number of individual connections, the new network mechanisms required to provide better-than-best service can be kept relatively simple.

The early Internet designers clearly had this notion of multiple classes of service in mind. Recall the type-of-service (ToS) field in the IPv4 header in Figure 4.13. IEN123 [ISI 1979] describes the ToS field also present in an ancestor of the IPv4 datagram as follows: "The Type of Service [field] provides an indication of the abstract parameters of the quality of service desired. These parameters are to be used to guide the selection of the actual service parameters when transmitting a datagram through a particular network. Several networks offer service precedence, which somehow treats high precedence traffic as more important that other traffic." More than four decades ago, the vision of providing different levels of service to different classes of traffic was clear! However, it's taken us an equally long period of time to realize this vision.

Motivating Scenarios

Let's begin our discussion of network mechanisms for providing multiple classes of service with a few motivating scenarios.

Figure 7.14 shows a simple network scenario in which two application packet flows originate on Hosts H1 and H2 on one LAN and are destined for Hosts H3 and H4 on another LAN. The routers on the two LANs are connected by a 1.5 Mbps link. Let's assume the LAN speeds are significantly higher than 1.5 Mbps, and focus on the output queue of router R1; it is here that packet delay and packet loss will occur if the aggregate sending rate of H1 and H2 exceeds 1.5 Mbps. Let's further suppose that a 1 Mbps audio application (for example, a CD-quality audio call) shares the 1.5 Mbps link between R1 and R2 with an HTTP Web-browsing application that is downloading a Web page from H2 to H4.

R1
1.5 Mbps link R2
H2

H1
H4
H3
Figure 7.14 Competing audio and HTTP applications
7.5 • NETWORK SUPPORT FOR MULTIMEDIA 637

In the best-effort Internet, the audio and HTTP packets are mixed in the output queue at R1 and (typically) transmitted in a first-in-first-out (FIFO) order. In this scenario, a burst of packets from the Web server could potentially fill up the queue, causing IP audio packets to be excessively delayed or lost due to buffer overflow at R1. How should we solve this potential problem? Given that the HTTP Web-browsing application does not have time constraints, our intuition might be to give strict priority to audio packets at R1. Under a strict priority scheduling discipline, an audio packet in the R1 output buffer would always be transmitted before any HTTP packet in the R1 output buffer. The link from R1 to R2 would look like a dedicated link of 1.5 Mbps to the audio traffic, with HTTP traffic using the R1-to-R2 link only when no audio traffic is queued. In order for R1 to distinguish between the audio and HTTP packets in its queue, each packet must be marked as belonging to one of these two classes of traffic. This was the original goal of the type-of-service (ToS) field in IPv4. As obvious as this might seem, this then is our first insight into mechanisms needed to provide multiple classes of traffic:

Insight 1: Packet marking allows a router to distinguish among packets belonging to different classes of traffic.

Note that although our example considers a competing multimedia and elastic flow, the same insight applies to the case that platinum, gold, and silver classes of service are implemented—a packet-marking mechanism is still needed to indicate that class of service to which a packet belongs.

Now suppose that the router is configured to give priority to packets marked as belonging to the 1 Mbps audio application. Since the outgoing link speed is 1.5 Mbps, even though the HTTP packets receive lower priority, they can still, on average, receive 0.5 Mbps of transmission service. But what happens if the audio application starts sending packets at a rate of 1.5 Mbps or higher (either maliciously or due to an error in the application)? In this case, the HTTP packets will starve, that is, they will not receive any service on the R1-to-R2 link. Similar problems would occur if multiple applications (for example, multiple audio calls), all with the same class of service as the audio application, were sharing the link’s bandwidth; they too could collectively starve the FTP session. Ideally, one wants a degree of isolation among classes of traffic so that one class of traffic can be protected from the other. This protection could be implemented at different places in the network—at each and every router, at first entry to the network, or at inter-domain network boundaries. This then is our second insight:

Insight 2: It is desirable to provide a degree of traffic isolation among classes so that one class is not adversely affected by another class of traffic that misbehaves.

638 CHAPTER 7 • MULTIMEDIA NETWORKING

We’ll examine several specific mechanisms for providing such isolation among traffic classes. We note here that two broad approaches can be taken. First, it is possible to perform traffic policing, as shown in Figure 7.15. If a traffic class or flow must meet certain criteria (for example, that the audio flow not exceed a peak rate of 1 Mbps), then a policing mechanism can be put into place to ensure that these criteria are indeed observed. If the policed application misbehaves, the policing mechanism will take some action (for example, drop or delay packets that are in violation of the criteria) so that the traffic actually entering the network conforms to the criteria. The leaky bucket mechanism that we'll examine shortly is perhaps the most widely used policing mechanism. In Figure 7.15, the packet classification and marking mechanism (Insight 1) and the policing mechanism (Insight 2) are both implemented together at the network's edge, either in the end system or at an edge router.

A complementary approach for providing isolation among traffic classes is for the link-level packet-scheduling mechanism to explicitly allocate a fixed amount of link bandwidth to each class. For example, the audio class could be allocated 1 Mbps at R1, and the HTTP class could be allocated 0.5 Mbps. In this case, the audio and HTTP flows see a logical link with capacity 1.0 and 0.5 Mbps, respectively, as shown in Figure 7.16. With strict enforcement of the link-level allocation of bandwidth, a class can use only the amount of bandwidth that has been allocated; in particular, it cannot utilize bandwidth that is not currently being used by others. For example, if the audio flow goes silent (for example, if the speaker pauses and generates no audio packets), the HTTP flow would still not be able to transmit more than 0.5 Mbps over the R1-to-R2 link, even though the audio flow's 1 Mbps bandwidth allocation is not being used at that moment. Since bandwidth is a "use-it-or-lose-it" resource, there is no reason to prevent HTTP traffic from using bandwidth not used by the audio traffic. We'd like to use bandwidth as efficiently as possible, never wasting it when it could be otherwise used. This gives rise to our third insight:

Insight 3: While providing isolation among classes or flows, it is desirable to use resources (for example, link bandwidth and buffers) as efficiently as possible.

Scheduling Mechanisms

Recall from our discussion in Section 1.3 and Section 4.3 that packets belonging to various network flows are multiplexed and queued for transmission at the output buffers associated with a link. The manner in which queued packets are selected for transmission on the link is known as the link-scheduling discipline. Let us now consider several of the most important link-scheduling disciplines in more detail.

640 CHAPTER 7 • MULTIMEDIA NETWORKING

R1
1.5 Mbps link
1.0 Mbps
logical link
0.5 Mbps
logical link
R2
H2
H1
H4
H3

Figure 7.16 ␂ Logical isolation of audio and HTTP traffic classes

First-In-First-Out (FIFO)

Figure 7.17 shows the queuing model abstractions for the FIFO link-scheduling discipline. Packets arriving at the link output queue wait for transmission if the link is currently busy transmitting another packet. If there is not sufficient buffering space to hold the arriving packet, the queue's packet-discarding policy then determines whether the packet will be dropped (lost) or whether other packets will be removed from the queue to make space for the arriving packet. In our discussion below, we will ignore packet discard. When a packet is completely transmitted over the outgoing link (that is, receives service) it is removed from the queue.

The FIFO (also known as first-come-first-served, or FCFS) scheduling discipline selects packets for link transmission in the same order in which they arrived at the output link queue. We're all familiar with FIFO queuing from bus stops (particularly in England, where queuing seems to have been perfected) or other service centers, where arriving customers join the back of the single waiting line, remain in order, and are then served when they reach the front of the line.

Figure 7.18 shows the FIFO queue in operation. Packet arrivals are indicated by numbered arrows above the upper timeline, with the number indicating the order

R1
1.5 Mbps link
R1 output
interface queue
R2
H2
H1
H4
H3
Figure 7.17 ␂ FIFO queuing abstraction
7.5 • NETWORK SUPPORT FOR MULTIMEDIA 641

in which the packet arrived. Individual packet departures are shown below the lower timeline. The time that a packet spends in service (being transmitted) is indicated by the shaded rectangle between the two timelines. Because of the FIFO discipline, packets leave in the same order in which they arrived. Note that after the departure of packet 4, the link remains idle (since packets 1 through 4 have been transmitted and removed from the queue) until the arrival of packet 5.

Priority Queuing

Under priority queuing, packets arriving at the output link are classified into priority classes at the output queue, as shown in Figure 7.19. As discussed in the previous section, a packet's priority class may depend on an explicit marking that it carries in its packet header (for example, the value of the ToS bits in an IPv4 packet), its source or destination IP address, its destination port number, or other criteria. Each priority class typically has its own queue. When choosing a packet to transmit, the priority queuing discipline will transmit a packet from the highest priority class that has a nonempty queue (that is, has packets waiting for transmission). The choice among packets in the same priority class is typically done in a FIFO manner.

Figure 7.20 illustrates the operation of a priority queue with two priority classes. Packets 1, 3, and 4 belong to the high-priority class, and packets 2 and 5 belong to the low-priority class. Packet 1 arrives and, finding the link idle, begins transmission. During the transmission of packet 1, packets 2 and 3 arrive and are queued in the low- and high-priority queues, respectively. After the transmission of packet 1, packet 3 (a high-priority packet) is selected for transmission over packet 2 (which, even though it arrived earlier, is a low-priority packet). At the end of the transmission of packet 3, packet 2 then begins transmission. Packet 4 (a high-priority packet) arrives during the transmission of packet 2 (a low-priority packet). Under a nonpreemptive priority queuing discipline, the transmission o a packet is not interrupted once it has begun. In this case, packet 4 queues for transmission and begins being transmitted after the transmission of packet 2 is completed.

Round Robin and Weighted Fair Queuing (WFQ)

Under the round robin queuing discipline, packets are sorted into classes as with priority queuing. However, rather than there being a strict priority of service among classes, a round robin scheduler alternates service among the classes. In the simplest form of round robin scheduling, a class 1 packet is transmitted, followed by a class 2 packet, followed by a class 1 packet, followed by a class 2 packet, and so on. A so-called work-conserving queuing discipline will never allow the link to remain idle whenever there are packets (of any class) queued for

Arrivals
Departures
Packet
in service
1
1 3 2 4 5
2 3
1 3 2 4 5
4 5
Time
Time
t = 0 t = 2 t = 4 t = 6 t = 8 t = 10 t = 12 t = 14
Figure 7.20 Operation of the priority queue
Arrivals Departures
Low-priority queue
(waiting area)
Classify
High-priority queue
(waiting area)
Link
(server)
Figure 7.19 Priority queuing model
7.5 • NETWORK SUPPORT FOR MULTIMEDIA 643

transmission. A work-conserving round robin discipline that looks for a packet of a given class but finds none will immediately check the next class in the round robin sequence.

Figure 7.21 illustrates the operation of a two-class round robin queue. In this example, packets 1, 2, and 4 belong to class 1, and packets 3 and 5 belong to the second class. Packet 1 begins transmission immediately upon arrival at the output queue. Packets 2 and 3 arrive during the transmission of packet 1 and thus queue for transmission. After the transmission of packet 1, the link scheduler looks for a class 2 packet and thus transmits packet 3. After the transmission of packet 3, the scheduler looks for a class 1 packet and thus transmits packet 2. After the transmission of packet 2, packet 4 is the only queued packet; it is thus transmitted immediately after packet 2.

A generalized abstraction of round robin queuing that has found considerable use in QoS architectures is the so-called weighted fair queuing (WFQ) discipline [Demers 1990; Parekh 1993]. WFQ is illustrated in Figure 7.22. Arriving packets are classified and queued in the appropriate per-class waiting area. As in round robin scheduling, a WFQ scheduler will serve classes in a circular manner—first serving class 1, then serving class 2, then serving class 3, and then (assuming there are three classes) repeating the service pattern. WFQ is also a work-conserving queuing discipline and thus will immediately move on to the next class in the service

sequence when it finds an empty class queue.
WFQ differs from round robin in that each class may receive a differential amount of service in any interval of time. Specifically, each class, i, is assigned a weight, wi
. Under WFQ, during any interval of time during which there are class i packets to send, class i will then be guaranteed to receive a fraction of service equal to wi
/(∑wj
), where the sum in the denominator is taken over all classes that also have packets queued for transmission. In the worst case, even if all classes have queued packets, class i will still be guaranteed to receive a fraction wi
/(∑wj
) of the ..

bandwidth. Thus, for a link with transmission rate R, class i will always achieve a throughput of at least R · wi
/(∑wj
). Our description of WFQ has been an idealized one, as we have not considered the fact that packets are discrete units of data and a packet's transmission will not be interrupted to begin transmission of another packet; [Demers 1990] and [Parekh 1993] discuss this packetization issue. As we will see in the following sections, WFQ plays a central role in QoS architectures. It is also available in today's router products [Cisco QoS 2012].
Policing: The Leaky Bucket
One of our earlier insights was that policing, the regulation of the rate at which a class or flow (we will assume the unit of policing is a flow in our discussion below) is allowed to inject packets into the network, is an important QoS mechanism. But what aspects of a flow's packet rate should be policed? We can identify three important policing criteria, each differing from the other according to the time scale over which the packet flow is policed:
• Average rate. The network may wish to limit the long-term average rate (packets per time interval) at which a flow's packets can be sent into the network. A crucial issue here is the interval of time over which the average rate will be policed. A flow whose average rate is limited to 100 packets per second is more constrained than a source that is limited to 6,000 packets per minute, even though both have the same average rate over a long enough interval of time. For example, the latter constraint would allow a flow to send 1,000 packets in a given second-long interval of time, while the former constraint would disallow this sending behavior.
Classify
Arrivals Departures
w1
w2
w3 Link
Figure 7.22 Weighted fair queuing (WFQ)
7.5 • NETWORK SUPPORT FOR MULTIMEDIA 645
• Peak rate. While the average-rate constraint limits the amount of traffic that can be sent into the network over a relatively long period of time, a peak-rate constraint limits the maximum number of packets that can be sent over a shorter period of time. Using our example above, the network may police a flow at an average rate of 6,000 packets per minute, while limiting the flow's peak rate to

1,500 packets per second.

- Burst size. The network may also wish to limit the maximum number of packets (the “burst” of packets) that can be sent into the network over an extremely short interval of time. In the limit, as the interval length approaches zero, the burst size limits the number of packets that can be instantaneously sent into the network. Even though it is physically impossible to instantaneously send multiple packets into the network (after all, every link has a physical transmission rate that cannot be exceeded!), the abstraction of a maximum burst size is a useful one.

The leaky bucket mechanism is an abstraction that can be used to characterize these policing limits. As shown in Figure 7.23, a leaky bucket consists of a bucket that can hold up to b tokens. Tokens are added to this bucket as follows. New tokens, which may potentially be added to the bucket, are always being generated at a rate of r tokens per second. (We assume here for simplicity that the unit of time is a second.) If the bucket is filled with less than b tokens when a token is generated, the newly generated token is added to the bucket; otherwise the newly generated token is ignored, and the token bucket remains full with b tokens.

Let us now consider how the leaky bucket can be used to police a packet flow. Suppose that before a packet is transmitted into the network, it must first remove a

646 CHAPTER 7 • MULTIMEDIA NETWORKING

To network

Packets

Remove

token

Token

wait area

Bucket holds

up to

b tokens

r tokens/sec

Figure 7.23 The leaky bucket policer

token from the token bucket. If the token bucket is empty, the packet must wait for a token. (An alternative is for the packet to be dropped, although we will not consider that option here.) Let us now consider how this behavior polices a traffic flow. Because there can be at most b tokens in the bucket, the maximum burst size for a leaky-bucket-policed flow is b packets. Furthermore, because the token generation rate is r, the maximum number of packets that can enter the network of any interval of time of length t is rt + b. Thus, the token-generation rate, r, serves to limit the long-term average rate at which packets can enter the network. It is also possible to use leaky buckets (specifically, two leaky buckets in series) to police a flow’s peak rate in addition to the long-term average rate; see the homework problems at the end of this chapter.

Leaky Bucket + Weighted Fair Queuing = Provable Maximum Delay in a Queue

Let’s close our discussion of scheduling and policing by showing how the two can be combined to provide a bound on the delay through a router’s queue. Let’s consider a router’s output link that multiplexes n flows, each policed by a leaky bucket with parameters bi and ri, i = 1, . . . , n, using WFQ scheduling. We use the term flow here loosely to refer to the set of packets that are not distinguished from each other by the scheduler. In practice, a flow might be comprised of traffic from a single end-to-end connection or a collection of many such connections, see Figure 7.24.

Recall from our discussion of WFQ that each flow, i, is guaranteed to receive a

share of the link bandwidth equal to at least R · wi
/(∑wj
), where R is the transmission rate of the link in packets/sec. What then is the maximum delay that a packet will
experience while waiting for service in the WFQ (that is, after passing through the
leaky bucket)? Let us focus on flow 1. Suppose that flow 1's token bucket is initially
full. A burst of b1 packets then arrives to the leaky bucket policer for flow 1. These
packets remove all of the tokens (without wait) from the leaky bucket and then join
the WFQ waiting area for flow 1. Since these b1 packets are served at a rate of at least
R · wi
/(∑wj
) packet/sec, the last of these packets will then have a maximum delay,
dmax, until its transmission is completed, where
The rationale behind this formula is that if there are b1 packets in the queue and
packets are being serviced (removed) from the queue at a rate of at least R · w1/
(∑wj
) packets per second, then the amount of time until the last bit of the last packet
is transmitted cannot be more than b1/(R · w1/(∑wj
)). A homework problem asks you
to prove that as long as r1 < R · w1/(∑wj
), then dmax is indeed the maximum delay
that any packet in flow 1 will ever experience in the WFQ queue.

7.5.3 Diffserv

Having seen the motivation, insights, and specific mechanisms for providing multiple classes of
service, let's wrap up our study of approaches toward proving multiple classes of service with an
example—the Internet Diffserv architecture [RFC
2475; RFC Kilkki 1999]. Diffserv provides service differentiation—that is, the ability to handle
different classes of traffic in different ways within the Internet in a scalable manner. The need for
scalability arises from the fact that millions of
simultaneous source-destination traffic flows may be present at a backbone router.
We'll see shortly that this need is met by placing only simple functionality within
the network core, with more complex control operations being implemented at the
network's edge.
Let's begin with the simple network shown in Figure 7.25. We'll describe one
possible use of Diffserv here; other variations are possible, as described in RFC
2475. The Diffserv architecture consists of two sets of functional elements:

- Edge functions: packet classification and traffic conditioning. At the incoming edge of the
network (that is, at either a Diffserv-capable host that generates
traffic or at the first Diffserv-capable router that the traffic passes through), arriving packets are
marked. More specifically, the differentiated service (DS) field in
the IPv4 or IPv6 packet header is set to some value [RFC 3260]. The definition
of the DS field is intended to supersede the earlier definitions of the IPv4 type-of-service field and
the IPv6 traffic class fields that we discussed in Chapter 4.
For example, in Figure 7.25, packets being sent from H1 to H3 might be marked
dmax = b1
R w1> gwj
648 CHAPTER 7 • MULTIMEDIA NETWORKING
at R1, while packets being sent from H2 to H4 might be marked at R2. The mark
that a packet receives identifies the class of traffic to which it belongs. Different
classes of traffic will then receive different service within the core network.
- Core function: forwarding. When a DS-marked packet arrives at a Diffserv-capable router, the

packet is forwarded onto its next hop according to the so-called per-hop behavior (PHB) associated with that packet's class. The per-hop behavior influences how a router's buffers and link bandwidth are shared among the competing classes of traffic. A crucial tenet of the Diffserv architecture is that a router's perhop behavior will be based only on packet markings, that is, the class of traffic to which a packet belongs. Thus, if packets being sent from H1 to H3 in Figure 7.25 receive the same marking as packets being sent from H2 to H4, then the network routers treat these packets as an aggregate, without distinguishing whether the packets originated at H1 or H2. For example, R3 would not distinguish between packets from H1 and H2 when forwarding these packets on to R4. Thus, the Diffserv architecture obviates the need to keep router state for individual source-destination pairs—a critical consideration in making Diffserv scalable.

An analogy might prove useful here. At many large-scale social events (for example, a large public reception, a large dance club or discothèque, a concert, or a football game), people entering the event receive a pass of one type or another: VIP passes for Very Important People; over-21 passes for people who are 21 years old or older (for example, if alcoholic drinks are to be served); backstage passes at concerts; press passes for reporters; even an ordinary pass for the Ordinary Person. These passes are typically distributed upon entry to the event, that is, at the edge of the event. It is here at the edge where computationally intensive operations, such as paying for entry, checking for the appropriate type of invitation, and matching an invitation against a piece of identification, are performed. Furthermore, there may be a limit on the number of people of a given type that are allowed into an event. If there is such a limit, people may have to wait before entering the event. Once inside the event, one's pass allows one to receive differentiated service at many locations around the event—a VIP is provided with free drinks, a better table, free food, entry to exclusive rooms, and fawning service. Conversely, an ordinary person is excluded from certain areas, pays for drinks, and receives only basic service. In both cases, the service received within the event depends solely on the type of one's pass. Moreover, all people within a class are treated alike.

7.5 • NETWORK SUPPORT FOR MULTIMEDIA 649
R4
Leaf router
Key:
Core router
R2
R1 R6
R7
R3 R5
H1
H2
H4
H3
R2 R3

Figure 7.25 A simple Diffserv network example

Figure 7.26 provides a logical view of the classification and marking functions within the edge router. Packets arriving to the edge router are first classified. The classifier selects packets based on the values of one or more packet header fields (for example, source address, destination address, source port, destination port, and protocol ID) and steers the packet to the appropriate marking function. As noted above, a packet's marking is carried in the DS field in the packet header.

In some cases, an end user may have agreed to limit its packet-sending rate to conform to a declared traffic profile. The traffic profile might contain a limit on the peak rate, as well as the burstiness of the packet flow, as we saw previously with the leaky bucket mechanism. As long as the user sends packets into the network in a way that conforms to the negotiated traffic profile, the packets receive their priority marking and are forwarded along their route to the destination. On the other hand, if the traffic profile is violated, out-of-profile packets might be marked differently, might be shaped (for example, delayed so that a maximum rate constraint would be observed), or might be dropped at the network edge. The role of the metering function, shown in Figure 7.26, is to compare the incoming packet flow with the negotiated traffic profile and to determine whether a packet is within the negotiated traffic profile. The actual decision about whether to immediately remark, forward, delay, or drop a packet is a policy issue determined by the network administrator and is not specified in the Diffserv architecture.

So far, we have focused on the marking and policing functions in the Diffserv architecture. The second key component of the Diffserv architecture involves the per-hop behavior (PHB) performed by Diffserv-capable routers. PHB is rather cryptically, but carefully, defined as “a description of the externally observable forwarding behavior of a Diffserv node applied to a particular Diffserv behavior aggregate” [RFC 2475]. Digging a little deeper into this definition, we can see several important considerations embedded within:

- A PHB can result in different classes of traffic receiving different performance (that is, different externally observable forwarding behaviors While a PHB defines differences in performance (behavior) among classes, it does not mandate any particular mechanism for achieving these behaviors. As long as the externally observable performance criteria are met, any implementation mechanism and any buffer/bandwidth allocation policy can be used. For example, a PHB would not require that a particular packet-queuing discipline (for example, a priority queue versus a WFQ queue versus a FCFS queue) be used to achieve a particular behavior. The PHB is the end, to which resource allocation and implementation mechanisms are the means.
- Differences in performance must be observable and hence measurable.

Two PHBs have been defined: an expedited forwarding (EF) PHB [RFC 3246] and an assured forwarding (AF) PHB [RFC 2597]. The expedited forwarding PHB specifies that the departure rate of a class of traffic from a router must equal or exceed a configured rate. The assured forwarding PHB divides traffic into four classes, where each AF class is guaranteed to be provided with some minimum amount of bandwidth and buffering.

Let’s close our discussion of Diffserv with a few observations regarding its service model. First, we have implicitly assumed that Diffserv is deployed within a single administrative domain, but typically an end-to-end service must be fashioned from multiple ISPs sitting between communicating end systems. In order to provide end-to-end Diffserv service, all the ISPs between the end systems must not only provide this service, but most also cooperate and make settlements in order to offer end customers true end-to-end service. Without this kind of cooperation, ISPs directly selling Diffserv service to customers will find themselves repeatedly saying: “Yes, we know you paid extra, but we don’t have a service agreement with the ISP that dropped and delayed your traffic. I’m sorry that there were so many gaps in your

Packets Forward
Classifier Marker
Drop
Shaper/
Dropper

Meter
Figure 7.26 A simple Diffserv network example
7.5 • NETWORK SUPPORT FOR MULTIMEDIA 651
VoIP call!” Second, if Diffserv were actually in place and the network ran at only
moderate load, most of the time there would be no perceived difference between a
best-effort service and a Diffserv service. Indeed, end-to-end delay is usually dominated by
access rates and router hops rather than by queuing delays in the routers.
Imagine the unhappy Diffserv customer who has paid more for premium service but
finds that the best-effort service being provided to others almost always has the
same performance as premium service!

7.5.4 Per-Connection Quality-of-Service (QoS) Guarantees: Resource Reservation and Call Admission

In the previous section, we have seen that packet marking and policing, traffic isolation, and
link-level scheduling can provide one class of service with better performance than another.
Under certain scheduling disciplines, such as priority scheduling,
the lower classes of traffic are essentially “invisible” to the highest-priority class of
traffic. With proper network dimensioning, the highest class of service can indeed
achieve extremely low packet loss and delay—essentially circuit-like performance.
But can the network guarantee that an ongoing flow in a high-priority traffic class
will continue to receive such service throughout the flow’s duration using only the
mechanisms that we have described so far? It cannot. In this section, we’ll see why
yet additional network mechanisms and protocols are required when a hard service
guarantee is provided to individual connections.
Let’s return to our scenario from Section 7.5.2 and consider two 1 Mbps
audio applications transmitting their packets over the 1.5 Mbps link, as shown in
Figure 7.27. The combined data rate of the two flows (2 Mbps) exceeds the link
652 CHAPTER 7 • MULTIMEDIA NETWORKING
R1
1.5 Mbps link
1 Mbps
audio
1 Mbps
audio
R2
H2
H1
H4
H3
Figure 7.27 Two competing audio applications overloading
the R1-to-R2 link
capacity. Even with classification and marking, isolation of flows, and sharing of
unused bandwidth (of which there is none), this is clearly a losing proposition.
There is simply not enough bandwidth to accommodate the needs of both applications at the
same time. If the two applications equally share the bandwidth,
each application would lose 25 percent of its transmitted packets. This is such an
unacceptably low QoS that both audio applications are completely unusable;
there’s no need even to transmit any audio packets in the first place.
Given that the two applications in Figure 7.27 cannot both be satisfied simultaneously, what
should the network do? Allowing both to proceed with an unusable
QoS wastes network resources on application flows that ultimately provide no utility to the end
user. The answer is hopefully clear—one of the application flows

should be blocked (that is, denied access to the network), while the other should be allowed to proceed on, using the full 1 Mbps needed by the application. The telephone network is an example of a network that performs such call blocking—if the required resources (an end-to-end circuit in the case of the telephone network) cannot be allocated to the call, the call is blocked (prevented from entering the network) and a busy signal is returned to the user. In our example, there is no gain in allowing a flow into the network if it will not receive a sufficient QoS to be considered usable. Indeed, there is a cost to admitting a flow that does not receive its needed QoS, as network resources are being used to support a flow that provides no utility to the end user.

By explicitly admitting or blocking flows based on their resource requirements, and the source requirements of already-admitted flows, the network can guarantee that admitted flows will be able to receive their requested QoS. Implicit in the need to provide a guaranteed QoS to a flow is the need for the flow to declare its QoS requirements. This process of having a flow declare its QoS requirement, and then having the network either accept the flow (at the required QoS) or block the flow is referred to as the call admission process. This then is our fourth insight (in addition to the three earlier insights from Section 7.5.2) into the mechanisms needed to provide QoS.

Insight 4: If sufficient resources will not always be available, and QoS is to be guaranteed, a call admission process is needed in which flows declare their QoS requirements and are then either admitted to the network (at the required QoS) or blocked from the network (if the required QoS cannot be provided by the network).

Our motivating example in Figure 7.27 highlights the need for several new network mechanisms and protocols if a call (an end-to-end flow) is to be guaranteed a given quality of service once it begins:

- Resource reservation. The only way to guarantee that a call will have the resources (link bandwidth, buffers) needed to meet its desired QoS is to explicitly allocate those resources to the call—a process known in networking parlance as resource reservation. Once resources are reserved, the call has on-demand access to these resources throughout its duration, regardless of the demands of all other calls. If a call reserves and receives a guarantee of x Mbps of link bandwidth, and never transmits at a rate greater than x, the call will see loss- and delay-free performance.
- Call admission. If resources are to be reserved, then the network must have a mechanism for calls to request and reserve resources. Since resources are not infinite, a call making a call admission request will be denied admission, that is, be blocked, if the requested resources are not available. Such a call admission is performed by the telephone network—we request resources when we dial a number. If the circuits (TDMA slots) needed to complete the call are available, the circuits are allocated and the call is completed. If the circuits are not available, then the call is blocked, and we receive a busy signal. A blocked call can try again to gain admission to the network, but it is not allowed to send traffic into the network until it has successfully completed the call admission process. Of course, a router that allocates link bandwidth should not allocate more than is available at that link. Typically, a call may reserve only a fraction of the link's bandwidth, and so a router may allocate link bandwidth to more than one call. However, the sum of the allocated bandwidth to all calls should be less than the link capacity if hard quality of service guarantees are to be provided.
- Call setup signaling. The call admission process described above requires that a call be able to reserve sufficient resources at each and every network router on its source-to-destination path to ensure that its end-to-end QoS

requirement is met. Each router must determine the local resources required by
the session, consider the amounts of its resources that are already committed to
other ongoing sessions, and determine whether it has sufficient resources to
satisfy the per-hop QoS requirement of the session at this router without vio␂lating local QoS
guarantees made to an already-admitted session. A signaling
protocol is needed to coordinate these various activities—the per-hop alloca␂tion of local
resources, as well as the overall end-to-end decision of whether or
not the call has been able to reserve sufficient resources at each and every
router on the end-to-end path. This is the job of the call setup protocol, as
shown in Figure 7.28. The RSVP protocol [Zhang 1993, RFC 2210] was
proposed for this purpose within an Internet architecture for providing quality␂of-service
guarantees. In ATM networks, the Q2931b protocol [Black 1995]
carries this information among the ATM network's switches and end point.
Despite a tremendous amount of research and development, and even prod␂ucts that provide for
per-connection quality of service guarantees, there has been
almost no extended deployment of such services. There are many possible rea␂sons. First and
foremost, it may well be the case that the simple application-level
mechanisms that we studied in Sections 7.2 through 7.4, combined with proper
654 CHAPTER 7 • MULTIMEDIA NETWORKING
network dimensioning (Section 7.5.1) provide "good enough" best-effort network
service for multimedia applications. In addition, the added complexity and cost of
deploying and managing a network that provides per-connection quality of serv␂ice guarantees
may be judged by ISPs to be simply too high given predicted cus␂tomer revenues for that service.
7.6 Summary
Multimedia networking is one of the most exciting developments in the Internet
today. People throughout the world are spending less time in front of their radios
and televisions, and are instead turning to the Internet to receive audio and video
transmissions, both live and prerecorded. This trend will certainly continue as high␂speed
wireless Internet access becomes more and more prevalent. Moreover, with
sites like YouTube, users have become producers as well as consumers of multime␂dia Internet
content. In addition to video distribution, the Internet is also being used
to transport phone calls. In fact, over the next 10 years, the Internet, along with wire␂less Internet
access, may make the traditional circuit-switched telephone system a
thing of the past. VoIP not only provides phone service inexpensively, but also pro␂vides
numerous value-added services, such as video conferencing, online directory
services, voice messaging, and integration into social networks such as Facebook
and Google+

Alright, this is a big one but it all hangs together nicely. Let's turn §7.5 into something you can read once and feel like "okay, I get what's going on here."

I'll structure it as:

1. Big picture: three approaches (Table 7.4)
2. 7.5.1 Dimensioning best-effort networks
3. 7.5.2 Multiple classes of service
    - Motivation

  - Scheduling (FIFO, Priority, Round Robin, WFQ)
  - Policing (leaky bucket)
4. 7.5.4 Per-connection QoS (resource reservation + call admission)

(I'll skip deep Diffserv details unless needed; focus on the concepts your prof listed.)

---

1. Big picture – what §7.5 is about

So far, we've seen **application-level tricks** for multimedia:

- Client buffering & prefetching
- Adaptive quality (change bitrate)
- Adaptive playout, FEC, interleaving, etc.

Those all assume the network is just **best effort** and we make do.

Section 7.5 asks: *Can the **network itself** help multimedia, beyond best effort?*
Three broad approaches (Table 7.4):

1. **"Make the best of best effort"**
   - Treat all traffic the same.
   - Just provision enough capacity (dimensioning/provisioning) so congestion is rare.
   - Low complexity, fully deployed today (this is basically the real Internet now).
2. **Multiple classes / Differentiated Service**
   - Different traffic classes get different treatment (e.g., VoIP vs web).
   - Needs:
     - **Marking** (identify class in packet header)
     - **Scheduling** (priorities, WFQ, etc.)
     - **Policing** (prevent abuse: leaky bucket)
   - "Soft" or no guarantees, but "better than best effort" for some classes.
3. **Per-connection QoS guarantees**
   - Each flow explicitly reserves resources end-to-end.
   - Can offer **hard or soft guarantees** on bandwidth, delay, loss.
   - Needs:
     - Resource reservation
     - Call admission
     - Signaling protocol (e.g., RSVP)
   - Conceptually powerful, but hardly deployed in the real Internet.

The rest of the section digs into each.

---

2. 7.5.1 Dimensioning best-effort networks

Core idea

Congestion → delay, jitter, loss.
One way to avoid that is **brute force**: provision so much capacity that congestion almost never happens.

This is **bandwidth provisioning / network dimensioning**:

- **Bandwidth provisioning**:
  Given a fixed network topology, choose link capacities so performance meets some requirement.
- **Network dimensioning**:
  Design both the **topology** (what is connected to what) **and** link capacities so end-to-end performance targets are met.

If you do this well, then even with simple **best-effort IP**, multimedia works fine—no need for fancy QoS mechanisms.

What you need to do dimensioning correctly

To design such a network, you need:

1. **Traffic demand models**
   - How many users start sessions? (call-level arrival)
   - How many packets per session, at what rates? (packet-level model)
   - How does traffic change over time (busy hour vs off-peak)?
2. **Performance requirements**
   - For example:
     "For delay-sensitive traffic,
     $P(\text{end-to-end delay} > D_{\max}) < \epsilon$"
   - This is a probabilistic requirement (e.g., only 1 in 10,000 packets may exceed 150 ms, etc.).
3. **Performance models + optimization**
   - Models that predict end-to-end delay, loss, etc. given:
     - Topology
     - Link capacities
     - Workload model
   - Optimization techniques to choose **minimal-cost** capacities that still meet requirements.

Why don't we just do this and be done?

Two main reasons:

1. **Economics**
   - ISPs would have to install a lot of capacity.

    - Question: will customers pay enough extra to justify this?
2. **Organization**
    - End-to-end path spans **multiple ISPs**.
    - They'd need to coordinate provisioning, maybe share revenue, agree on SLAs.
    - Hard to get global cooperation.

So in practice, we do *some* provisioning (especially in backbone networks), but not enough to give hard guarantees for everyone. That's why the other approaches (classes of service, QoS) were proposed.

---

3. 7.5.2 Multiple classes of service

Here the idea is: instead of **all traffic treated equally**, have a **small number of classes**:

- e.g., VoIP class, video class, best-effort class, maybe "premium user" vs "regular user".

Then give the "important" class:

- Lower delay, fewer drops, maybe more bandwidth share.

Motivating scenario (very exam-ish)

Network:

- Two hosts H1, H2 on LAN 1 → router R1
- R1 to R2 link = 1.5 Mbps
- R2 → H3, H4 on LAN 2

Traffic:

- H1 → H3: audio stream, 1 Mbps (delay-sensitive)
- H2 → H4: HTTP / web download (elastic, no rigid delay constraint)

On the 1.5 Mbps R1–R2 link, they share bandwidth.

If we just do FIFO best effort:

- Audio packets and HTTP packets go into the same queue.
- A burst of HTTP packets can:
    - Fill buffer → audio packets delayed or dropped.
    - User hears gaps / glitches in the audio.

We'd like to fix this. The chapter extracts three **insights**:

Insight 1 – Packet marking

> **Packet marking lets routers distinguish traffic classes.**

For the router to treat audio differently from HTTP, it needs to know which packet is which.

We can:

- Use a field in the IP header (e.g., ToS / DSCP).
- Or classify based on port number, IP addresses, etc.
- Then mark the packet as "class X".

This is the basic requirement: *classification* + *marking*.

Insight 2 – Isolation between classes

Even with marking and priority, problems can occur:

- Suppose we give **strict priority** to audio.
- Audio sends at 1 Mbps normally, but what if:
  - It (bug or attack) sends at 1.5 Mbps or more, or
  - Multiple high-priority flows appear?

Then HTTP gets **starved** – they get almost no service.

We want:

> **Isolation**: one class shouldn't be completely ruined by another class that misbehaves.

Two broad ways to get isolation:

1. **Policing**
    - Enforce rules at the edge:
      - "Audio flow can't exceed 1 Mbps on average, can't burst too hard."
    - If it misbehaves → drop or delay packets so they conform.
2. **Scheduling with reserved shares**
    - At the link, explicitly allocate some share of bandwidth to each class (e.g., 1 Mbps for audio, 0.5 Mbps for HTTP) using something like **WFQ**.

We'll dig into both shortly.

Insight 3 – Efficiency (don't waste unused bandwidth)

Suppose we allocate:

- 1 Mbps to audio
- 0.5 Mbps to HTTP

What if the audio user is silent for a while?

- If we rigidly enforce 1.0 and 0.5 Mbps with no sharing:

    - HTTP could still use only 0.5 Mbps → wasted capacity.
- But bandwidth is “use it or lose it”; unused share should be available to others.

So:

> **While providing isolation, we still want to use bandwidth efficiently.**

That’s exactly what smarter schedulers like WFQ do: guarantee minimum shares but allow grabbing more when others are idle.

3.1 Scheduling mechanisms (how router decides who goes next)

This is about what happens at a single **output queue** to decide which queued packet is sent next.

1. FIFO (First-In-First-Out)

- Single queue for all packets.
- Order = arrival order.
- Easy, simple, but **no QoS differentiation**.

Conceptually:

- Packets arrive → join end of queue.
- When link is free → next packet from head-of-line is transmitted.

This is the default in plain best-effort routers.

---

2. Priority Queuing

- Packets classified into **priority classes** (e.g., high = VoIP, low = HTTP).
- Each class has its own queue.
- Transmit rule:
    - Always choose the **highest priority non-empty queue**.
    - Within a queue: FIFO.

Example:

- Packets 1, 3, 4 → high priority
- Packets 2, 5 → low priority
- Even if packet 2 arrived before 3, 3 will be sent first because high > low.

Usually non-preemptive:

- Once a packet starts transmitting, it’s not interrupted, even if a higher priority arrives mid-transmission.

Pros:

- Great for **delay-sensitive** traffic.

Cons:

- Lower-priority flows may **starve** if high-priority traffic volume is too big.
- This is why we still need **policing** and/or more nuanced scheduling.

---

3. Round Robin (RR)

- Packets are still grouped into classes.
- But scheduling cycles through classes in order.

Basic RR:

- If classes 1 and 2:
  - Send one packet from class 1
  - Then one from class 2
  - Then back to class 1, etc.
- If a class's queue is empty, we skip it (work-conserving).

Pros:

- Fairer than priority: no permanent starvation.

Cons:

- Doesn't account for differences in required bandwidth.
- Class that needs more share still only gets "one turn per round".

---

4. Weighted Fair Queuing (WFQ)

WFQ is a generalized RR that allows **different weights per class**.

- Each class i gets a **weight** w_i.
- Idea: in any time period, each non-empty class i gets at least a fraction:

$$\frac{w_i}{\sum_j w_j}$$

of the link service (bandwidth).

If link capacity = R packets/sec, then class i gets at least:

$$\text{throughput}_i \geq R \cdot \frac{w_i}{\sum_j w_j}$$

- It's **work-conserving**: if some class queues are empty, others can use extra bandwidth.

- Ensures:
  - **Isolation** (each class has a guaranteed minimum share).
  - **Efficiency** (unused share can be borrowed by others).

Real implementations are approximate because packets are discrete (can't interrupt a packet mid-send), but conceptually this is the idea.

WFQ is a key building block for QoS systems.

3.2 Policing – controlling how much each flow can inject

Even with scheduling, you want flows to **behave** as promised (no sending 10 Mbps when you reserved 1 Mbps).

Policing checks:

1. **Average rate**
   - Long-term limit on packets / second.
   - Example: ≤ 100 packets/sec on average.
2. **Peak rate**
   - Short-term maximum.
   - Example: ≤ 1500 packets/sec bursts, even if 6000/minute average is allowed.
3. **Burst size**
   - Max bursts allowed in a very short interval.
   - Think: how many packets can be sent "back-to-back" at line rate before being throttled.

The leaky bucket / token bucket abstraction

The book's "leaky bucket" here is actually the **token bucket** model:

- Bucket can hold **b tokens** max.
- New tokens arrive at rate **r tokens/sec**.
  - If bucket has < b tokens, a new token is added.
  - If bucket already has b tokens, extra tokens are discarded.

Policing rule:

- To send **one packet**, flow must **take one token** from the bucket.
- If no tokens available → the packet must wait (or be dropped, depending on policy).

What does this enforce?

- At most **b packets** can be sent in a burst (since that's how many tokens max).
- Over any interval of length t, max packets ≤ **r·t + b**.

  - Long-term average rate ≈ r.
  - Short-term bursts allowed, but limited by b.

So:

- **r** ≈ average rate
- **b** = burst size bound

With some extensions (two buckets, etc.), you can control **peak rate** as well.

Combining leaky bucket + WFQ → delay bound

Imagine:

- A link with capacity R packets/sec.
- It uses WFQ to schedule flows.
- Each flow i is policed with a token bucket (b_i, r_i).
- Flow i has a WFQ weight w_i.

WFQ guarantees that flow i gets at least:

$$R \cdot \frac{w_i}{\sum_j w_j}$$

If we focus on flow 1:

- Suppose its bucket is full (b_1 tokens).
- A burst of **b_1 packets** arrives → all b_1 enter the WFQ queue for flow 1.

These b_1 packets are served at ≥ $R \cdot \frac{w_1}{\sum_j w_j}$ packets/sec.

So the **max queueing delay** for the last burst packet is:

$$d_{\max} = \frac{b_1}{R \cdot \frac{w_1}{\sum_j w_j}}$$

Interpretation:

> maximum delay ≈ (max burst size) / (guaranteed per-flow rate).

If the long-term sending rate r_1 is **less** than the guaranteed rate, then no worse delay appears later. So this gives a **provable delay bound** for that flow.

This is how:

> **Policing (limit bursts) + Scheduling (reserve min rate) ⇒ you can mathematically bound delay.**

---

4. 7.5.4 Per-connection QoS (resource reservation + call admission)

So far, multiple classes give “better than best effort” **for classes**, but do not guarantee that a **specific flow** will always meet strict QoS.

Now we go one level deeper: **per-connection QoS**.

The overloaded link example

Revisit the earlier topology, but now:

- Two audio flows, each needing 1 Mbps.
- Link R1–R2 is **only 1.5 Mbps**.

Combined demand = 2 Mbps > 1.5 Mbps.

No matter what clever scheduling you do:

- If both are allowed in, each at best gets ~0.75 Mbps → heavy loss/delay.
- End result: **both calls are unusable**.

So what should the network do?

> **Block one call, admit the other.**

Let only one 1 Mbps call proceed, giving it enough bandwidth to be good quality. The blocked one gets the equivalent of a “busy signal.”

This leads to the 4th insight:

> **Insight 4: If resources are limited and you want QoS guarantees, you need a call admission process.**
> Flows must declare requirements, and the network must either **admit** (guaranteeing those requirements) or **reject** them.

Three extra mechanisms for per-connection QoS

To guarantee QoS to individual flows, we need:

1. **Resource reservation**
    - Each flow reserves specific resources on each router/link along its path:
        - Bandwidth on links
        - Buffer space
    - Once reserved, those resources are “set aside” for that flow—it gets them whenever it needs (up to its reserved rate).
2. **Call admission**
    - Before a flow starts sending, it *asks* the network:
      “Can I have, say, 1 Mbps and delay ≤ D?”

   - Each router along the path checks:
     - How much capacity is already reserved?
     - Does admitting this one violate guarantees to existing flows?
   - If **yes**, flow is admitted.
   - If **no**, flow is blocked (like a busy phone line).
   - Only admitted flows are allowed to send traffic.
3. **Call setup signaling**
   - There must be a **protocol** to:
     - Carry QoS request from the source to every router on the path.
     - Tell each router what to reserve.
     - Collect accept/reject decisions on the way back.
   - Examples:
     - **RSVP** in IP networks.
     - **Q.2931** signaling in ATM networks.

So per-connection QoS = **IntServ-style** architecture:

- Flows ask for guarantees
- Network uses reservation + admission + signaling
- If admitted, flows get hard QoS (as long as they obey traffic profile).

Why hasn't this taken over the Internet?

Despite lots of research and some products, per-flow QoS hasn't seen widespread deployment, because:

- Application-level tricks + good provisioning + best-effort are often "good enough."
- Per-flow QoS is **complex** and expensive to implement and manage at Internet scale:
  - Every router along the path must maintain state for each flow.
  - Signaling and admission control logic is required.
- Revenue vs complexity trade-off often doesn't justify it for ISPs.

So most of today's Internet still runs with:

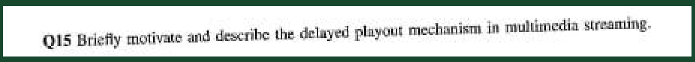
- **Best-effort + some dimensioning + application-level smarts**,
- Some limited **differentiated service** in operator networks,
- Very little true **per-connection guaranteed QoS**.

Quick "cheat-sheet" recap for §7.5

- **7.5.1 Dimensioning**
  - Goal: add enough capacity so congestion is rare → best effort is "good enough."

  - Needs traffic models, performance targets, performance modeling & optimization.
  - Barriers: cost and multi-ISP coordination.
- **Multiple classes of service**
  - Treat aggregates of traffic differently (e.g., VoIP vs web).
  - **Insight 1**: mark packets to identify class.
  - **Insight 2**: policing and/or bandwidth allocation to isolate classes.
  - **Insight 3**: use bandwidth efficiently (WFQ).
  - **Scheduling**:
    - FIFO: simple, no QoS.
    - Priority: good for low delay, risk of starvation.
    - Round Robin: fairness between classes.
    - WFQ: guarantees minimum share per class + efficient usage.
  - **Policing**:
    - Limit average rate, peak rate, burst size.
    - Token/leaky bucket: parameters r (rate), b (burst).
    - With WFQ, you can bound max delay:
      delay ∝ burst size / guaranteed rate.
- **Per-connection QoS**
  - If you want **guaranteed** QoS per flow:
    - Need **resource reservation**, **call admission**, **signaling**.
    - Flows declare QoS needs; network admits or rejects.
  - Very powerful but rarely deployed due to complexity + economics.

If you want next, I can condense all of this into **a 1-page formula + concept sheet** for fast CS232 revision, or make a short set of **exam-style conceptual questions** (e.g., “Explain the difference between scheduling and policing using the audio/HTTP example”).



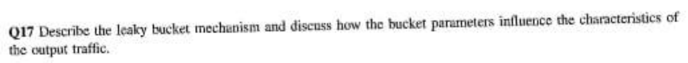
Q15 Briefly motivate and describe the delayed playout mechanism in multimedia streaming.

exam ques, answer it

Delayed playout in multimedia streaming

- **Motivation:** In best-effort IP networks, packets (audio/video chunks) experience variable end-to-end delay (jitter).
  If the client plays a chunk immediately on arrival, some later chunks may not arrive in time, causing freezes, gaps, or skips in playback.

- **Mechanism:**
  1. When streaming starts, the client **does not play immediately**. It first **buffers** some amount Q of media data.
  2. Each chunk has a **timestamp** (generation time). The client chooses a **playout delay** q and schedules chunk i (timestamp t_i) to be played at time $t_i + q$.
  3. As long as most packets arrive before their scheduled playout times, playback is smooth; packets arriving after $t_i + q$ are discarded as "lost."
- **Effect:** By intentionally delaying playout a bit (a few hundred ms or a few seconds for video), the client's buffer can absorb network jitter and short-term bandwidth drops, providing **continuous, periodic playback** instead of stalls and glitches.



Q17 Describe the leaky bucket mechanism and discuss how the bucket parameters influence the characteristics of the output traffic.

question from last year, answer in details, idk anything about leaky bucket

Here's a **clean, full-credit, exam-ready explanation** of the **Leaky Bucket mechanism**, written assuming **you know nothing about it**:

Leaky Bucket Mechanism – Definition, Working, and Effect of Parameters

✅ 1. What is the Leaky Bucket Mechanism? (Definition)

The **leaky bucket mechanism is a traffic policing and shaping algorithm** used in computer networks to:

- **Control the rate** at which packets enter the network
- **Prevent congestion**
- **Enforce Quality of Service (QoS)** guarantees

It ensures that a sender **cannot transmit data faster than an allowed rate**, even if it tries to send in bursts.

✅ 2. Why Do We Need the Leaky Bucket? (Motivation)

In real networks:

- Applications may send **bursty traffic**
- Bursts cause **buffer overflow, delay, and packet loss**

- Multimedia (voice/video) traffic needs **controlled, smooth transmission**

So the leaky bucket is used to:

- **Smooth bursty traffic**
- **Limit bandwidth usage**
- **Protect the network from misbehaving senders**

✅ 3. How the Leaky Bucket Works (Step-by-Step)

Think of a **real water bucket with a small hole at the bottom**:

- Water enters the bucket irregularly (packets arrive)
- Water leaks out at a **constant rate** (controlled transmission)

🔹 Components:

Component	Meaning
Bucket size (b)	Maximum number of packets allowed in burst
Leak rate (r)	Constant output rate (packets/sec)
Packets	Data from sender

🔹 Operation:

1. **Packets arrive** from the sender and are placed in the bucket.
2. The bucket can hold **only b packets maximum**.
3. Packets **leak out at constant rate r**.
4. If the bucket is **full and new packets arrive** → **packets are dropped**.
5. Output traffic becomes **smooth and regulated**.

✅ **Conclusion:**
👉 Input traffic can be bursty,
👉 Output traffic is **strictly controlled at rate r**.

✅ 4. Policing vs Shaping

Type	Behavior
Leaky Bucket Policing	Drops excess packets
Leaky Bucket Shaping	Delays excess packets instead of dropping

---

✅ 5. Output Traffic Characteristics

After leaky bucket:

- ✅ **Constant rate transmission**
- ✅ **No sudden bursts**
- ✅ **Network congestion reduced**
- ✅ **Predictable QoS behavior**

✅ 6. How Bucket Parameters Affect Output Traffic

This part is **very important for exams**.

🔹 (A) Bucket Size (b) — Controls Burst Tolerance

- Maximum number of packets that can be sent at once.

If b is small:

- Only small bursts allowed
- High burst traffic → packet loss

If b is large:

- Larger bursts allowed
- Better support for multimedia and bursty apps

✅ **Effect of b**:

Controls **maximum burst size**

🔹 (B) Leak Rate (r) — Controls Long-Term Average Rate

- Maximum steady output speed of the flow.

If r is small:

- Slow output
- More buffering and delay

If r is large:

- Fast output
- Higher bandwidth usage

✅ **Effect of r**:

Controls **average transmission rate**

🔹 (C) Relationship Between b and r

Maximum packets allowed in time interval **T**:

$$\text{Packets} \leq rT + b$$

This means:

- **r controls long-term rate**
- **b allows short-term burst**

✅ 7. What Does Leaky Bucket Guarantee?

Property	Guaranteed?
Average rate limit	✅ YES
Burst control	✅ YES
Peak rate control	❌ Needs token bucket
Zero packet loss	❌ NO (policing mode drops packets)

✅ 8. Difference Between Leaky Bucket and Token Bucket (Quick Exam Note)

Feature	Leaky Bucket	Token Bucket
Output rate	Constant	Variable
Burst allowed	Limited	Larger
Excess traffic	Dropped	Allowed if tokens saved
Use case	Strict shaping	Flexible shaping

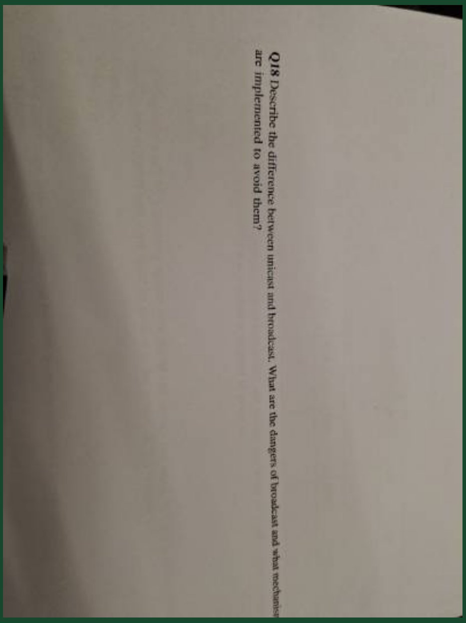
✅ 9. Final Exam-Ready Answer (You Can Write This Exactly)

> The leaky bucket mechanism is a traffic policing and shaping algorithm used to regulate the rate at which packets enter a network. It consists of a bucket of finite capacity **b** and a constant leak rate **r**. Packets arriving from the source are stored in the bucket and transmitted at the fixed rate **r**. If the bucket becomes full, excess packets are either delayed or dropped.

The parameter **r** controls the long-term average transmission rate, while **b** controls the maximum burst size allowed. A larger **b** allows larger bursts without packet loss, while a larger **r** allows higher sustained bandwidth. The output traffic is therefore smoothed and constrained, reducing congestion and enabling quality-of-service guarantees.

If you want, I can also:
✅ Give a **numerical example**
✅ Show a **diagram explanation**
✅ Or convert this into a **4–5 mark vs 8–10 mark exam version**

Just tell me how many marks this question has.



Q18 Describe the difference between unicast and broadcast. What are the dangers of broadcast and what mechanis
are implemented to avoid them?

Here is a **complete, exam-ready, high-scoring answer** to **Q18**:

Q18. Difference Between Unicast and Broadcast. Dangers of Broadcast and Mechanisms to Avoid Them

---

✅ 1. Unicast vs Broadcast (Core Difference)

Feature	Unicast	Broadcast
Communication type	One-to-one	One-to-all
Sender	One sender	One sender
Receiver	One specific receiver	All hosts in the network
Bandwidth usage	Efficient	Very inefficient

Feature	Unicast	Broadcast
Addressing	Uses a **specific destination IP**	Uses **broadcast IP (e.g., 255.255.255.255)**
Network traffic	Limited	Floods the entire network
Example	Web browsing, email, file transfer	ARP requests, DHCP discovery
Scalability	Scales well	Does **not** scale

✅ **Summary:**

- **Unicast = private communication**
- **Broadcast = public announcement to the whole network**

✅ 2. What is Broadcast Used For? (Legitimate Uses)

Broadcast is used only when **all devices must listen**, such as:

- **ARP** – Find MAC address of a device
- **DHCP Discover** – Find a DHCP server
- **Routing updates (older protocols)**
- **Local service discovery**

✅ These are **control-plane uses**, not for large data transfer.

✅ 3. Dangers of Broadcast (Very Important for Exams)

Broadcast is **dangerous if overused or abused** because:

🔴 (1) Broadcast Storm

If too many broadcast packets are generated:

- Every device must process every packet
- CPU overload
- Network slowdown or complete crash

✅ This is called a **Broadcast Storm**

🔴 (2) Bandwidth Wastage

- Broadcast packets are delivered to **all hosts**
- Even hosts that do not need the data must receive it
- Waste of:

    - Bandwidth
    - Processing power
    - Energy (especially in wireless networks)

🔴 (3) Network Congestion

Since broadcasts multiply rapidly:

- Queue overflow
- Packet drops
- Delay increases for normal unicast traffic

🔴 (4) Security Risks

Attackers can use broadcast for:

- **ARP spoofing**
- **Smurf attacks**
- **MAC flooding**
- **Network reconnaissance**

✅ One attacker → whole network impacted

🔴 (5) Poor Scalability

As network size increases:

- Broadcast traffic increases **quadratically**
- Eventually becomes **unmanageable**

✅ 4. Mechanisms Used to Avoid Broadcast Dangers

✅ (A) Using Routers (Broadcast Domain Limiting)

- Routers **do NOT forward broadcast packets**
- This:
    - Confines broadcast to **local LAN only**
    - Prevents global broadcast storms

✅ Each router creates a **new broadcast domain**

✅ (B) Using VLANs (Virtual LANs)

- One physical network is divided into multiple logical networks
- Broadcast is restricted to **only one VLAN**
- Reduces unnecessary traffic

✅ Used heavily in large enterprise networks

✅ (C) Multicast Instead of Broadcast

Broadcast	Multicast
To everyone	Only to subscribed members
Wasteful	Efficient
No membership control	Group-based delivery

✅ Multicast = **controlled broadcast**

Examples:

- Live video streaming
- Stock price updates
- Online lectures

✅ (D) Broadcast Rate Limiting / Storm Control

Switches enforce:

- Maximum broadcast traffic per second
- If limit exceeded → packets dropped

✅ Prevents **broadcast storms**

✅ (E) ARP Caching

- Reduces repeated ARP broadcasts
- Devices remember MAC/IP mappings

✅ Fewer ARP broadcasts → lower traffic

✅ 5. Final Exam-Ready Answer (You Can Write This Directly)

Unicast is a one-to-one communication in which one sender transmits data to one specific receiver, whereas broadcast is a one-to-all communication in which one sender transmits data to all hosts in the network. Unicast is bandwidth-efficient and scalable, whereas broadcast consumes excessive bandwidth and does not scale well.

The dangers of broadcast include broadcast storms, bandwidth wastage, congestion, security vulnerabilities such as ARP spoofing and Smurf attacks, and poor scalability in large networks.

These dangers are avoided using mechanisms such as routers to limit broadcast domains, VLANs to contain broadcast within logical groups, multicast to replace inefficient broadcast delivery, broadcast rate limiting to prevent storms, and ARP caching to reduce unnecessary broadcasts.

If you want, I can also:
✅ Add **diagrams**
✅ Convert into **5-mark or 10-mark format**
✅ Give **real-world attack examples (Smurf, ARP poisoning)**
✅ Or prepare **mock exam questions for this topic**

Just tell me your exam marking scheme.