UNIT –IV

UDP –TCP – Adaptive Flow Control-Adaptive Retransmission – Congestion Control – Congestion avoidance – QoS.

Transport Layer

Introduction

The following are some of the **common properties that a transport protocol** can be expected to provide:

- guarantees message delivery
- delivers messages in the same order they are sent
- delivers at most one copy of each message
- supports arbitrarily large messages
- supports synchronization between the sender and the receiver
- allows the receiver to apply flow control to the sender
- supports multiple application processes on each host

Some of the more **typical limitations of the network** are that it may

- drop messages
- reorder messages
- deliver duplicate copies of a given message
- limit messages to some finite size
- deliver messages after an arbitrarily long delay

The **transport protocols** provide the following **services**

- a simple asynchronous demultiplexing service (UDP)
- a reliable byte-stream service (TCP)
- a request/reply service. (RPC)

End-to-End Protocols

4.1 Simple Demultiplexer (UDP)

- The simplest possible transport protocol is one that extends the host-to-host delivery service of the underlying network into a process-to-process communication service.
- Demultiplexing is allowing multiple application processes on each host to share the network.
- The Internet's User Datagram Protocol (UDP) is an example of such a transport protocol.

How does UDP identify the target process?

- The common approach, and the one used by UDP, is for processes to indirectly identify each other using an abstract locator, often called a **port or mailbox.**

- The basic idea is for a source process to send a message to a port and for the destination process to receive the message from a port.
- The header for an end-to-end protocol that implements this demultiplexing function typically contains an identifier (port) for both the sender (source) and the receiver (destination) of the message.

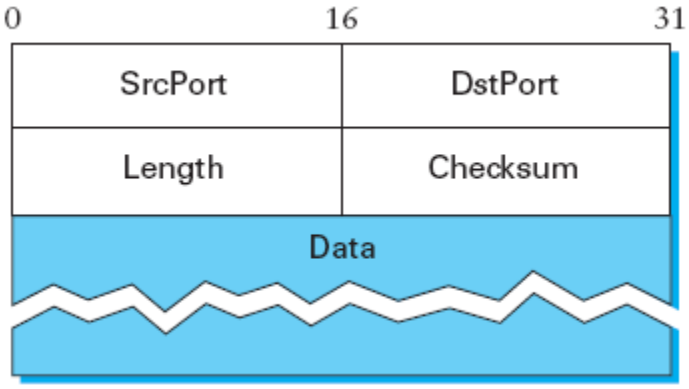

Figure 4.1.1 Format for UDP Header

How does a process learns the port for the process to which it wants to send a message?

- A client process initiates a message exchange with a server process.
- Once a client has contacted a server, the server knows the client's port (it was contained in the message header) and can reply to it.

How does the client learn the server's port in the first place?

- A common approach is for the server to accept messages at a **well-known port.**
- That is, each server receives its messages at some fixed port that is widely published.
- For example, the Domain Name Server (DNS) receives messages at well-known port 53 on each host, the mail service listens for messages at port 25, and the Unix talk program accepts essages at well-known port 517, and so on.
- An alternative strategy is to use only a single well-known port—the one at which the **"Port Mapper"** service accepts messages.
    - A client would send a message to the Port Mapper's well-known port asking for the port it should use to talk to the "whatever" service, and the Port Mapper returns the appropriate port.
    - This strategy makes it easy to change the port associated with different services over time, and for each host to use a different port for the same service.

A port is implemented by a **message queue**, as illustrated in Figure 4.1.2.

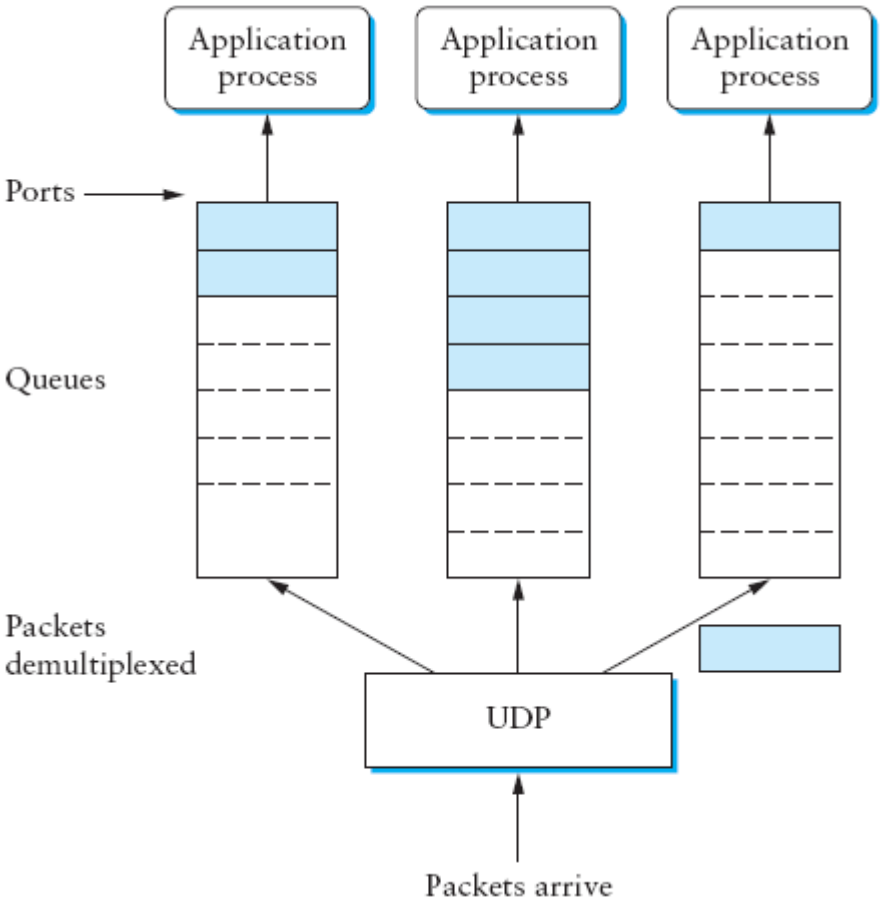

Figure 4.1.2 UDP message queue

- When a message arrives, the protocol (e.g., UDP) appends the message to the end of the queue.
- If the queue is full, the message is discarded.
- There is no flow-control mechanism that tells the sender to slow down.
- When an application process wants to receive a message, one is removed from the front of the queue.
- If the queue is empty, the process blocks until a message becomes available.
- UDP does not implement flow control or reliable/ordered delivery,
- UDP ensures the correctness of the message by the use of a checksum.
    - UDP computes its checksum over the UDP header, the contents of the message body, and something called the **pseudoheader**.

- The pseudoheader consists of three fields from the IP header
    1. protocol number
    2. source IP address
    3. destination IP address + the UDP length field.

- The pseudoheader is to verify that this message has been delivered between the correct two endpoints.
-

4.2 Reliable Byte Stream (TCP)

- The Internet's Transmission Control Protocol (TCP) is probably the most widely used transport protocol
- TCP guarantees the reliable, in-order delivery of a stream of bytes.
- It is a full-duplex protocol, meaning that each TCP connection supports a pair of byte streams, one flowing in each direction.
- It also includes a flow-control mechanism for each of these byte streams that allows the receiver to limit how much data the sender can transmit at a given time.
- Finally, like UDP, TCP supports a demultiplexing mechanism that allows multiple application programs on any given host to simultaneously carry on a conversation with their peers.

End-to-End Issues

- At the heart of TCP is the sliding window algorithm.
- Even though this is the same basic algorithm because TCP runs over the Internet rather than a point-to-point link, there are many important differences.
- The sliding window algorithm we already learnt runs over a single physical link that always connects the same two computers.
- TCP supports logical connections between processes that are running on any two computers in the Internet.
- This means that TCP needs an explicit connection establishment phase during which the two sides of the connection agree to exchange data with each other. T
- A single physical link that always connects the same two computers has a fixed RTT, TCP connections are likely to have widely different round-trip times.
- Variations in the RTT are even possible during a single TCP connection that lasts only a few minutes.
- What this means to the sliding window algorithm is that the timeout mechanism that triggers retransmissions must be adaptive
- Packets may be reordered as they cross the Internet, but this is not possible on a point-to-point link where the first packet put into one end of the link must be the first to appear at the other end.
    - Packets that are slightly out of order do not cause a problem since the sliding window algorithm can reorder packets correctly using the sequence number.
    - The real issue is how far out-of-order packets can get, or said another way, how late a packet can arrive at the destination.

- TCP also implements a highly tuned flow control and congestion-control mechanisms.
- Flow control involves preventing senders from overrunning the capacity of receivers.
- Congestion control involves preventing too much data from being injected into the network, thereby causing switches or links to become overloaded.
- Thus, flow control is an end-to-end issue, while congestion control is concerned with how hosts and networks interact.

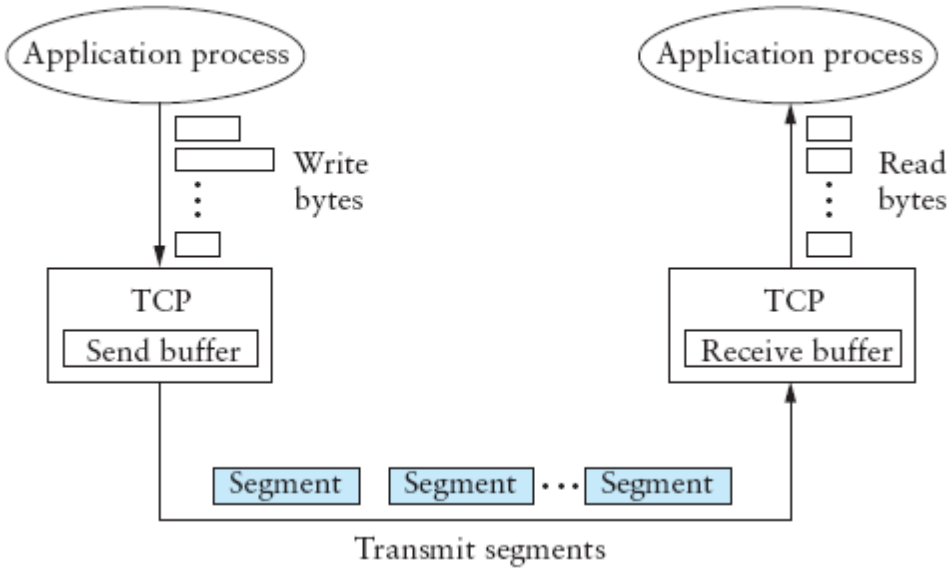

Figure 4.2.1 How TCP manages a byte stream.

How TCP manages a byte stream?

- TCP is a byte-oriented protocol, which means that the sender writes bytes into a TCP connection and the receiver reads bytes out of the TCP connection.
- TCP on the source host buffers enough bytes from the sending process to fill a reasonably sized packet and then sends this packet to its peer on the destination host.
- TCP on the destination host then empties the contents of the packet into a receive buffer, and the receiving process reads from this buffer at its leisure.
- This situation is illustrated in Figure 4.2.1, which, for simplicity, shows data flowing in only one direction.
- A single TCP connection supports byte streams flowing in both directions.
- The packets exchanged between TCP peers are called **segments** since each one carries a segment of the byte stream.

TCP Segment Format

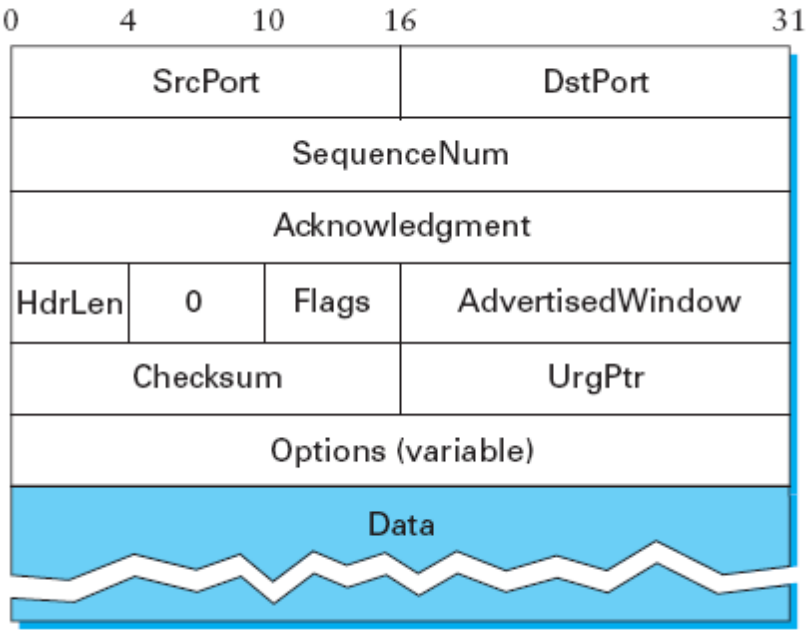

Figure 4.2.3 TCP header format.

- Each TCP segment contains the header schematically depicted in Figure 4.2.3.
- The SrcPort and DstPort fields identify the source and destination ports, respectively,just as in UDP.
- The Acknowledgment, SequenceNum, and AdvertisedWindow fields are all involved in TCP's sliding window algorithm.
- Each byte of data has a sequence number; the SequenceNum field contains the sequence number for the first byte of data carried in that segment.
- The Acknowledgment and AdvertisedWindow fields carry information about the flow of data going in the other Direction .
- The 6-bit Flags field is used to relay control information between TCP peers.
- The possible flags include SYN, FIN, RESET, PUSH, URG, and ACK.
- The SYN and FIN flags are used when establishing and terminating a TCP connection, respectively.
- The ACK flag is set any time the Acknowledgment field is valid, implying that the receiver should pay attention to it.
- The URG flag signifies that this segment contains urgent data. When this flag is set, the UrgPtr field indicates where the nonurgent data contained in this segment begins.

- The PUSH flag signifies that the sender invoked the push operation, which indicates to the receiving side of TCP that it should notify the receiving process of this fact.
- The RESET flag signifies that the receiver has become confused—for example, because it received a segment it did not expect to receive—and so wants to abort the connection.

4.3 TCP Connection Establishment and Termination

- A TCP connection begins with a client (caller) doing an active open to a server (callee).
- Assuming that the server had earlier done a passive open, the two sides engage in an exchange of messages to establish the connection.
- Only after this connection establishment phase is over do the two sides begin sending data.
- As soon as a participant is done sending data, it closes one direction of the connection, which causes TCP to initiate a round of connection termination messages.

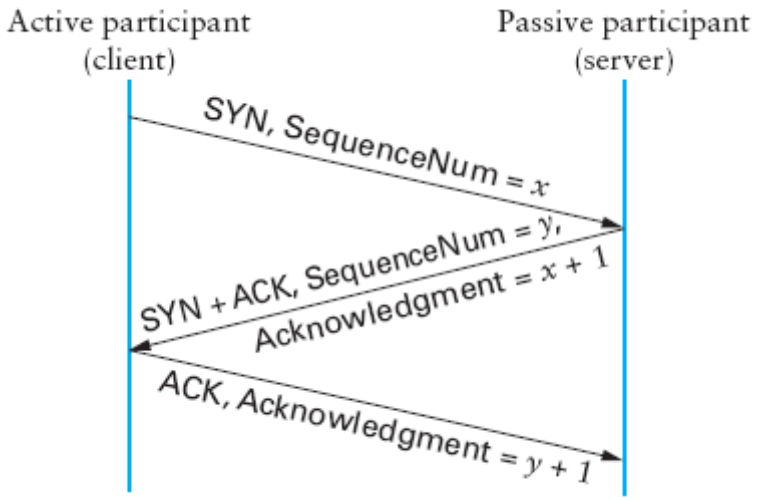

Figure 4.3.1 Timeline for three-way handshake algorithm.

Three-Way Handshake

- The algorithm used by TCP to establish and terminate a connection is called a three- way handshake.
- The three-way handshake involves the exchange of three messages between the client and the server, as illustrated by the timeline given in Figure 4.3.1.
- First, the client (the active participant) sends a segment to the server (the passive participant) stating the initial sequence number it plans to use (Flags = SYN, SequenceNum = x).
- The server then responds with a single segment that both acknowledges the client's sequence number (Flags = ACK, Ack = x + 1) and states its own beginning sequence number (Flags = SYN, SequenceNum = y). That is, both the SYN and ACK bits are set in the Flags field of this second message.

- Finally, the client responds with a third segment that acknowledges the server's sequence number (Flags = ACK, Ack = y + 1).
- The reason that each side acknowledges a sequence number that is one larger than the one sent is that the Acknowledgment field actually identifies the "next sequence number expected."

TCP State Transition Diagram

- TCP state transition diagram is given in Figure 4.3.2.
- This diagram shows only the states involved in opening a connection (everything above ESTABLISHED) and in closing a connection (everything below ESTABLISHED). Everything that goes on while a connection is open—that is, the operation of the sliding window algorithm—is hidden in the ESTABLISHED state.

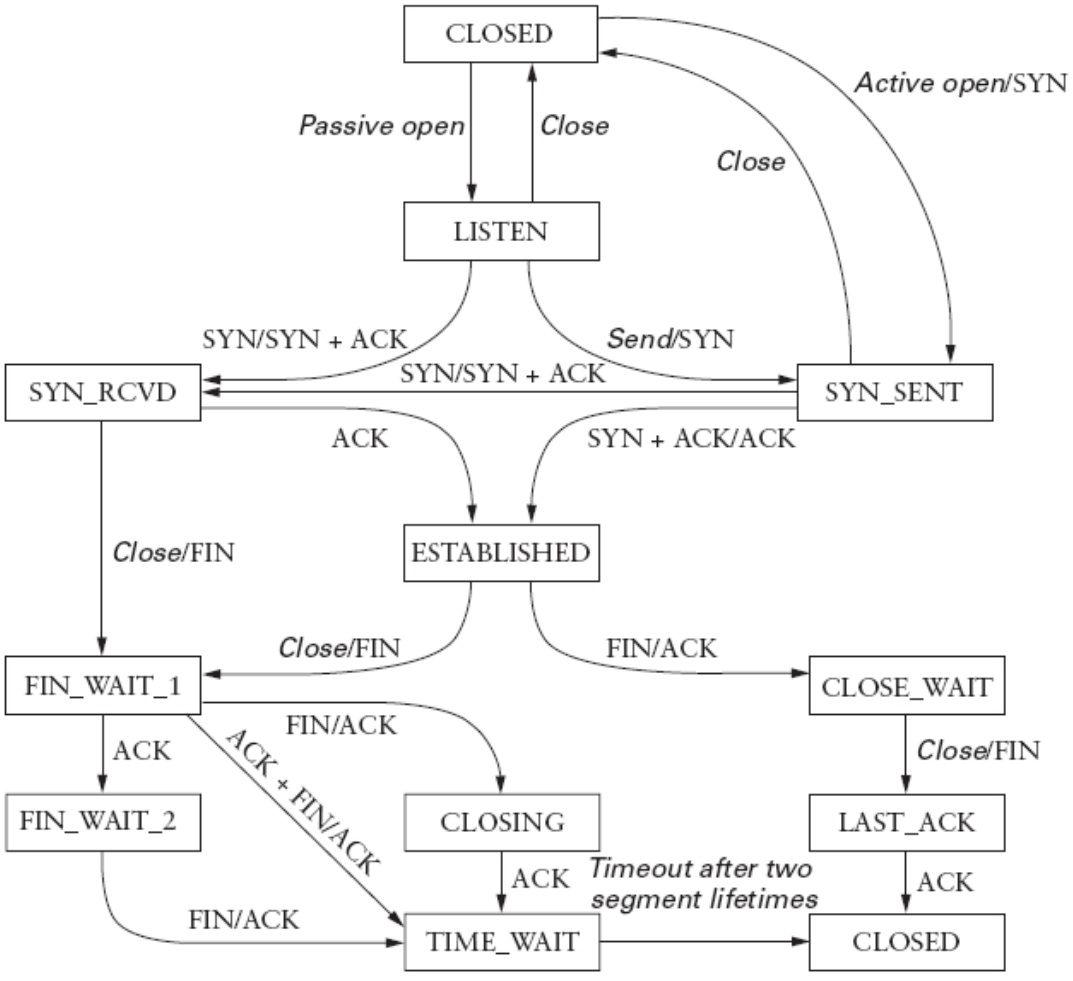

Figure 4.3.2. TCP state transition diagram.

- Each circle denotes a state that one end of a TCP connection can find itself in.
- All connections start in the CLOSED state.
- Each arc is labeled with a tag of the form event/action.

- Thus, if a connection is in the LISTEN state and a SYN segment arrives (i.e., a segment with the SYN flag set), the connection makes a transition to the SYN RCVD state and takes the action of replying with an ACK + SYN segment.
- Now let's trace the typical transitions taken through the diagram in Figure 4.3.2.
- When opening a connection, the server first invokes a passive open operation on TCP, which causes TCP to move to the LISTEN state.
- At some later time, the client does an active open, which causes its end of the connection to send a SYN segment to the server and to move to the SYN SENT state.
- When the SYN segment arrives at the server, it moves to the SYN RCVD state and responds with a SYN+ACK segment.
- The arrival of this segment causes the client to move to the ESTABLISHED state and to send an ACK back to the server.
- When this ACK arrives, the server finally moves to the ESTABLISHED state. In other words, we have just traced the three-way handshake.
- Now to the process of terminating a connection, the important thing to keep in mind is that the application process on both sides of the connection must independently close its half of the connection.
- If only one side closes the connection, then this means it has no more data to send, but it is still available to receive data from the other side.
- Thus, on any one side there are three combinations of transitions that get a connection from the ESTABLISHED state to the CLOSED state:
    - ■ This side closes first:
      ESTABLISHED → FIN WAIT 1 → FIN WAIT 2 → TIME WAIT → CLOSED.
    - ■ The other side closes first:
      ESTABLISHED → CLOSE WAIT → LAST ACK → CLOSED.
    - ■ Both sides close at the same time:
      ESTABLISHED → FIN WAIT 1 → CLOSING → TIME WAIT → CLOSED.

4.4 Adaptive Flow Control

- Flow control is preventing the sender from overrunning the capacity of the receiver.
- We go back to our sliding window algorithm.
- Both buffers are of some finite size, denoted MaxSendBuffer and MaxRcvBuffer,
- The size of the window sets the amount of data that can be sent without waiting for acknowledgment from the receiver.
- Thus, the **receiver throttles the sender by advertising a window** that is no larger than the
- amount of data that it can buffer.
- Observe that **TCP on the receive side** must keep
  LastByteRcvd − LastByteRead ≤ MaxRcvBuffer
- to avoid overflowing its buffer.
- It therefore advertises a window size of
  AdvertisedWindow = MaxRcvBuffer −LastByteRcvd − LastByteRead)

- which represents the amount of free space remaining in its buffer.

- TCP on the sender side must then adhere to the advertised window it gets from the receiver.
- This means that at any given time, it must ensure that

  LastByteSent − LastByteAcked ≤ AdvertisedWindow
- The sender computes an effective window that limits how much data it can send:

  EffectiveWindow = AdvertisedWindow − (LastByteSent − LastByteAcked)
- The sender side must also make sure that the local application process does not overflow the send buffer, that is,

  LastByteWritten − LastByteAcked ≤ MaxSendBuffer

4.5 Adaptive Retransmission

- TCP guarantees the reliable delivery of data, hence it retransmits each segment if an ACK is not received in a certain period of time.
- TCP sets this timeout as a function of the RTT it expects between the two ends of the connection.
- Since TCP can have a wide variety of RTTs, it uses an adaptive retransmission mechanism.

Original Algorithm

- To compute a timeout value between a pair of hosts.
- This is the algorithm that was originally described in the TCP specification
- The idea is to keep a running average of the RTT and then to compute the timeout as a function of this RTT.
- Specifically, every time TCP sends a data segment, it records the time.
- When an ACK for that segment arrives, TCP reads the time again and then takes the difference between these two times as a SampleRTT.
- TCP then computes an EstimatedRTT as a weighted average between the previous estimate and this new sample. That is,

  EstimatedRTT = α × EstimatedRTT + (1 − α) × SampleRTT
- The parameter α is selected to smooth the EstimatedRTT.
- α lies between 0.8 and 0.9.

  TimeOut = 2 × EstimatedRTT

Karn/Partridge Algorithm

- A flaw was discovered in the above simple algorithm.
- The problem was that an ACK does not really acknowledge a transmission; it actually acknowledges the receipt of data.
- It is necessary to know which transmission to associate the ACK with so as to compute an accurate SampleRTT.
- As illustrated in Figure 4.5, if you assume that the ACK is for the original transmission but it was really for the second, then the SampleRTT is too large (a),

- while if you assume that the ACK is for the second transmission but it was actually for the first, then the SampleRTT is too small (b)

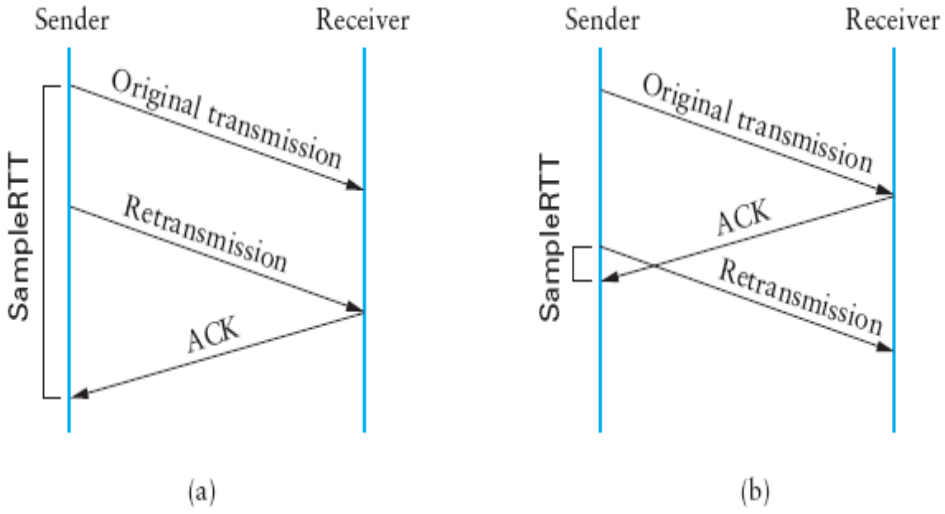

Figure 4.5 Associating the ACK with (a) original transmission versus (b) retransmission.

- The solution is simple.
- Whenever TCP retransmits a segment, it stops taking samples of the RTT; it only measures SampleRTT for segments that have been sent only once.
- This solution is known as the Karn/Partridge algorithm, after its inventors.
- Their proposed fix also includes a second small change to TCP's timeout mechanism.
- Each time TCP retransmits, it sets the next timeout to be twice the last timeout, rather than basing it on the last EstimatedRTT.
- That is, Karn and Partridge proposed that TCP use exponential backoff, similar to what the Ethernet does.

Jacobson/Karels Algorithm

- The Karn/Partridge algorithm was introduced at a time when the Internet was suffering from high levels of network congestion.
- Their approach was designed to fix some of the causes of that congestion, and although it was an improvement, the congestion was not eliminated.
- A couple of years later, two other researchers—Jacobson and Karels—proposed a more drastic change to TCP to battle congestion.
- It should be clear how the timeout mechanism is related to congestion—if you time out too soon, you may unnecessarily retransmit a segment, which only adds to the load on the network.
- The main problem with the original computation is that it does not take the variance of the sample RTTs into account.
- Intuitively, if the variation among samples is small, then the EstimatedRTT can be better trusted.
- In the new approach, the sender measures a new SampleRTT as before. It then folds this new sample into the timeout calculation as follows:

Difference = SampleRTT − EstimatedRTT

EstimatedRTT = EstimatedRTT + (δ × Difference)
Deviation = Deviation + δ(|Difference| − Deviation)

- where δ is a fraction between 0 and 1.
- TCP then computes the timeout value as a function of both EstimatedRTT and
- Deviation as follows:

TimeOut = μ × EstimatedRTT + φ × Deviation

- μ is typically set to 1 and φ is set to 4.
- Thus, when the variance is small, TimeOut is close to EstimatedRTT; a large variance causes the Deviation term to dominate the calculation.

4.6 TCP Congestion Control

- The idea of TCP congestion control is for each source to determine how much capacity is available in the network, so that it knows how many packets it can safely have in transit.
- Once a given source has this many packets in transit, it uses the arrival of an ACK as a signal that one of its packets has left the network, and that it is therefore safe to insert a new packet into the network without adding to the level of congestion.
- By using ACKs to pace the transmission of packets, TCP is said to be self-clocking.

Congestion Control Mechanisms

1. Additive Increase/Multiplicative Decrease

- TCP maintains a new state variable for each connection, called CongestionWindow.
- Congestion Window is used by the source to limit how much data it is allowed to have in transit at a given time.
- The congestion window is congestion control's counterpart to flow control's advertised window.
- TCP is modified such that the maximum number of bytes of unacknowledged data allowed is now the minimum of the congestion window and the advertised window.
- Effective window is revised as follows:

MaxWindow = MIN(CongestionWindow, AdvertisedWindow)
EffectiveWindow = MaxWindow − (LastByteSent − LastByteAcked).

- That is, MaxWindow replaces AdvertisedWindow in the calculation of EffectiveWindow.
- Thus, a TCP source is allowed to send no faster than the slowest component—the network or the destination host—can accommodate.
- The AdvertisedWindow, is sent by the receiving side of the connection.
- The TCP source sets the CongestionWindow based on the level of congestion it perceives to exist in the network.
- This involves decreasing the congestion window when the level of congestion goes up and increasing the congestion window when the level of congestion goes down.

- This mechanism is commonly called additive increase/multiplicative decrease (AIMD).

How does the source determine that the network is congested and that it should decrease the congestion window?

- TCP interprets timeouts as a sign of congestion and reduces the rate at which it is transmitting.
- Specifically, each time a timeout occurs, the source sets CongestionWindow to half of its previous value.
- This halving of the CongestionWindow for each timeout corresponds to the "multiplicative decrease" part of AIMD.
- Although CongestionWindow is defined in terms of bytes, it is easiest to understand multiplicative decrease if we think in terms of whole packets.
- For example, suppose the CongestionWindow is currently set to 16 packets. If a loss is detected, CongestionWindow is set to 8.
- Additional losses cause CongestionWindow to be reduced to 4, then 2, and finally to 1 packet.
- CongestionWindow is not allowed to fall below the size of a single packet, or in TCP terminology, the maximum segment size (MSS).
- We also need to be able to increase the congestion window to take advantage of newly available capacity in the network.
- This is the "additive increase" part of AIMD, and it works as follows.
- Every time the source successfully sends a CongestionWindow's worth of packets—that is, each packet sent out during the last RTT has been ACKed—it adds the equivalent of one packet to CongestionWindow.
- This linear increase is illustrated in Figure 4.6.1.
- Specifically, the congestion window is incremented as follows each time an ACK arrives:

Increment = MSS × (MSS/CongestionWindow)
CongestionWindow + = Increment

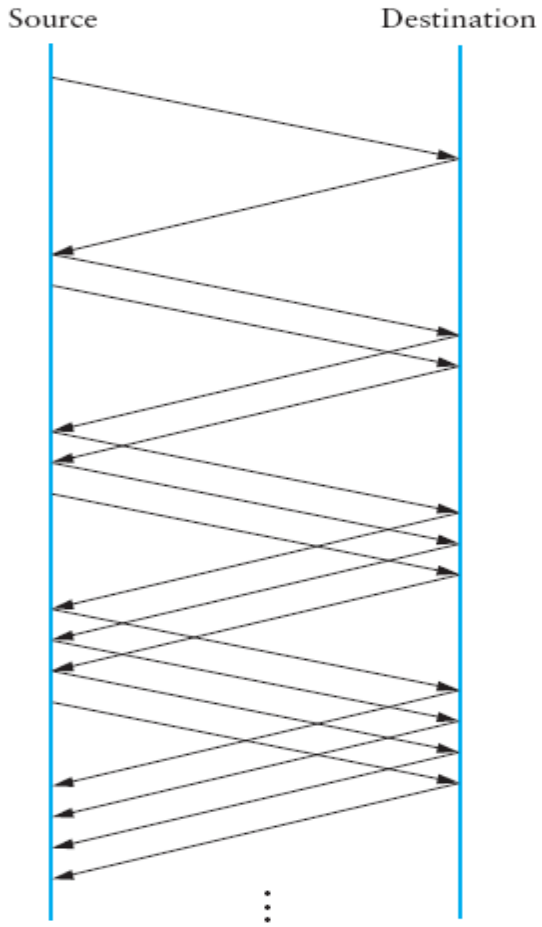

Figure 4.6.1 Packets in transit during additive increase, with one packet being added each RTT.

2. Slow Start

- The additive increase mechanism is used when the source is operating close to the available capacity of the network.
- It takes too long to ramp up a new TCP connection .

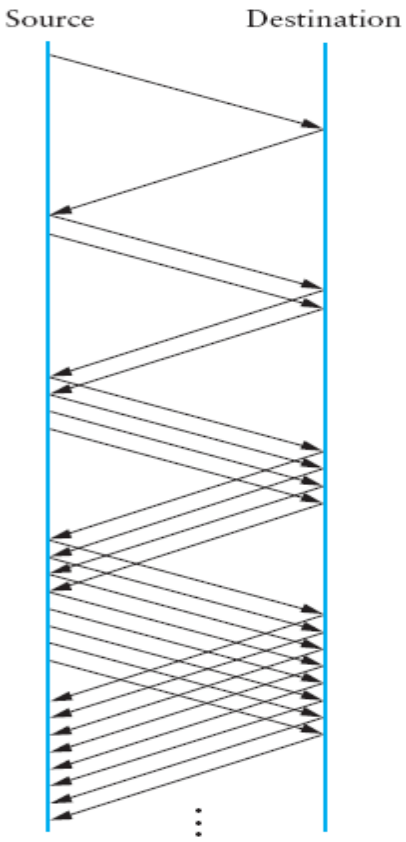

Figure 4.6.2 Packets in transit during slow start.

- Slow start is used to increase the congestion window rapidly from a cold start.
- Slow start effectively increases the congestion window exponentially, rather than linearly.
- Specifically, the source starts out by setting CongestionWindow to one packet.
- When the ACK for this packet arrives, TCP adds 1 to CongestionWindow and then
- sends two packets.
- Upon receiving the corresponding two ACKs, TCP increments CongestionWindow by 2—one for each ACK—and next sends four packets.
- The end result is that TCP effectively doubles the number of packets it has in transit every RTT.
- Figure 4.6.2 shows the growth in the number of packets in transit during slow start.
- Compare this to the linear growth of additive increase illustrated in Figure 4.6.1.

3. Fast Retransmit and Fast Recovery

- Fast retransmit is a heuristic that sometimes **triggers the retransmission of a dropped packet sooner than the regular timeout mechanism.**

- The fast retransmit mechanism does not replace regular timeouts; it just enhances that facility.
- The idea of fast retransmit is straightforward.
- Every time a data packet arrives at the receiving side, the receiver responds with an acknowledgment, even if this sequence number has already been acknowledged.
- Thus, when a packet arrives out of order— that is, TCP cannot yet acknowledge the data the packet contains because earlier data has not yet arrived—TCP resends the same acknowledgment it sent the last time.
- This second transmission of the same acknowledgment is called a **duplicate ACK**.
- When the sending side sees a duplicate ACK, it knows that the other side must have received a packet out of order, which suggests that an earlier packet might have been lost.
- Since it is also possible that the earlier packet has only been delayed rather than lost, the sender waits until it sees some number of duplicate ACKs and then retransmits the missing packet.
- In practice, TCP waits until it has seen **three duplicate ACKs** before retransmitting the packet.

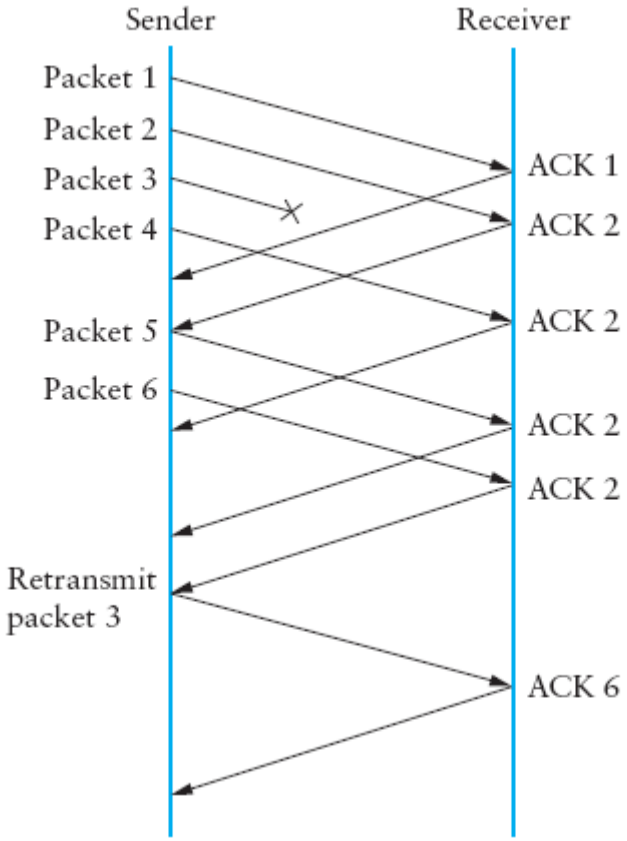

Figure 4.6.3 Fast retransmit based on duplicate ACKs.

- Figure 4.6.3 illustrates how duplicate ACKs lead to a fast retransmit.

- In this example, the destination receives packets 1 and 2, but packet 3 is lost in the network.
- Thus, the destination will send a duplicate ACK for packet 2 when packet 4 arrives, again when packet 5 arrives, and so on.
- When the sender sees the third duplicate ACK for packet 2—the one sent because the receiver had gotten packet 6—it retransmits packet 3.
- Note that when the retransmitted copy of packet 3 arrives at the destination, the receiver then sends a cumulative ACK for everything up to and including packet 6 back to the source.
- When the fast retransmit mechanism signals congestion, it is possible to use the ACKs that are still in the pipe to clock the sending of packets.
- This mechanism, which is called **fast recovery**, effectively removes the slow start phase that happens between when fast retransmit detects a lost packet and additive increase begins.
- For example, fast recovery avoids the slow and instead simply cuts the congestion window in half and resumes additive increase.
- In other words, slow start is only used at the beginning of a connection .
- At all other times, the congestion window is following a pure additive increase/multiplicative decrease pattern.

4.7 Congestion-Avoidance Mechanisms

- Congestion Avoidance is to predict when congestion is about to happen and then to reduce the rate at which hosts send data just before packets start being discarded.
- This section describes three different congestion-avoidance mechanisms.
- They are DECbit , RED and Source-based Congestion Avoidance.
- The first two take a similar approach: They put a small amount of additional functionality into the router to assist the end node in the anticipation of congestion.
- The third mechanism is very different from the first two: It attempts to avoid congestion purely from the end nodes.

1. DECbit

- The first mechanism was developed for use on the Digital Network Architecture (DNA), a connectionless network with a connection-oriented transport protocol.
- Each router monitors the load it is experiencing and explicitly notifies the end nodes when congestion is about to occur.
- This notification is implemented by setting a binary congestion bit in the packets that flow through the router.
- The destination host then copies this congestion bit into the ACK it sends back to the source.
- Finally, the source adjusts its sending rate so as to avoid congestion.
- **What happens in the router?**
    - A single congestion bit is added to the packet header.

- A router sets this bit in a packet if its average queue length is greater than or equal to 1 at the time the packet arrives.
- This average queue length is measured over a time interval that spans the last busy+idle cycle, plus the current busy cycle. (The router is busy when it is transmitting and idle when it is not.)

- **What does the source do to avoid congestion?**
- The source records how many of its packets resulted in some router setting the congestion bit.
- The source maintains a congestion window, just as in TCP, and watches to see what
- fraction of the last window's worth of packets resulted in the bit being set.
- If less than 50% of the packets had the bit set, then the source increases its congestion window by one packet.
- If 50%or more of the last window's worth of packets had the congestion bit
- set, then the source decreases its congestion window to 0.875 times the previous value.
- The "increase by 1, decrease by 0.875" rule was selected because additive increase/multiplicative decrease makes the mechanism stable.

2. Random Early Detection (RED)

- A second mechanism, called random early detection (RED), is similar to the DECbit scheme in that each router is programmed to monitor its own queue length, and when
- it detects that congestion is imminent, to notify the source to adjust its congestion window.
- RED differs from the DECbit scheme in two major ways.
    1. The first is that rather than explicitly sending a congestion notification message to the source, RED is most commonly implemented such that it implicitly notifies the source of congestion by dropping one of its packets. The source is, therefore, effectively notified by the subsequent timeout or duplicate ACK.
    2. The second difference between RED and DECbit is in the details of how RED decides when to drop a packet and what packet it decides to drop.

- When to drop a packet and what packet it to drop?
    - Drop each arriving packet with some **drop probability** whenever the queue length exceeds some **drop level.**
    - This idea is called **early random drop**.
    - The RED algorithm defines the details of how to monitor the queue length and when to drop a packet.
- RED computes an average queue length .
- That is, AvgLen is computed as

  AvgLen = (1 −Weight) × AvgLen +Weight × SampleLen
- where 0 <Weight < 1 and SampleLen is the length of the queue when a sample measurement is made.
- Second, RED has two queue length thresholds that trigger certain activity:

- ✓ MinThreshold and MaxThreshold.

- When a packet arrives at the gateway, RED compares the current AvgLen with these two thresholds, according to the following rules:

if AvgLen ≤ MinThreshold
−→ queue the packet
if MinThreshold < AvgLen < MaxThreshold
−→ calculate probability P
−→ drop the arriving packet with probability P
if MaxThreshold ≤ AvgLen
−→ drop the arriving packet

3. Source-Based Congestion Avoidance

- The general idea of these techniques is to watch for some sign from the network that some router's queue is building up and that congestion will happen soon if nothing is done about it.
- For example, the source might notice that as **packet queues build up in the network's routers, there is a measurable increase in the RTT for each successive packet it sends.**

Algorithm 1:

- For **every two round-trip delays** the algorithm checks to see
  - if the **current RTT** is **greater** than the **average of the minimum and maximum RTTs** seen so far.
- If it is, then the algorithm **decreases the congestion window by one-eighth**.

Algorithm 2:

- The window is adjusted once **every two round-trip delays** based on the product

(CurrentWindow − OldWindow) ×(CurrentRTT − OldRTT)

- If the **result is positive**, the source **decreases the window size by one-eighth.**
- If the **result is negative or zero**, the source **increases the window by one maximum packet size.**

Algorithm 3:

- For every RTT, the source **increases the window size by one packet** and **compares the throughput achieved to the throughput when the window was one packet smaller.**
- If the **difference is less than one-half the throughput achieved when only one packet was in transit ,** then **decreases the window by one packet.**
- The throughput is calculated by dividing the number of bytes outstanding in
- the network by the RTT.

Algortihm 4: (TCP Vegas)

- This last algorithm looks at changes in the throughput rate, or more specifically, changes in the sending rate.

- It compares the measured throughput rate with an expected throughput rate.
- First, define a given flow's BaseRTT to be the RTT of a packet when the flow is not congested.

  BaseRTT = minimum of all measured round-trip times;
- The expected throughput is given by

  ExpectedRate = CongestionWindow/BaseRTT
- The current sending rate, ActualRate is calculated as

  ActualRate = Number of bytes transmitted / sample RTT
- This calculation is done once per round-trip time.
- Compare ActualRate to ExpectedRate and adjust the window accordingly.

  Let **Diff = ExpectedRate−ActualRate.**
- Note that Diff is positive or 0 by definition, since ActualRate > ExpectedRate implies that we need to change BaseRTT to the latest sampled RTT.
- We also define two thresholds, **$\alpha < \beta$**.
- When **Diff < α**, it **increases the congestion window** linearly during the next RTT
- When **Diff > β**, it **decreases the congestion window** linearly during the next RTT.
- The **congestion window is unchanged** when **$\alpha <$ Diff $< \beta$**.
- We can see that the **farther away the actual throughput gets from the expected throughput,** the **more congestion** there is in the network, which implies that the sending rate should be reduced. The **β threshold triggers this decrease.**
- On the other hand, when the **actual throughput rate gets too close to the expected throughput**, the **connection is in danger** of not utilizing the available bandwidth. The **α threshold triggers this increase**.

4.8 Quality of Service

- For many years, packet-switched networks have offered the promise of supporting multimedia applications, that is, those that combine audio, video, and data.
- There is more to transmitting audio and video over a network than just providing sufficient bandwidth.
- For example , participants in a telephone conversation, expect to be able to converse in such a way that one person can respond to something said by the other and be heard almost immediately.
- Thus, the **timeliness of delivery** can be very important.
- We refer to **applications that are sensitive to the timeliness of data** as **real-time applications.**
- The **distinguishing characteristic of real-time applications** is that they need some sort of **assurance from the network** that data is likely to **arrive on time** .
- Whereas a **non-real-time application(elastic)** can use an end-to-end retransmission strategy to make sure that data arrives correctly, such a strategy **cannot provide timeliness.**
- Timely arrival must be provided by the network itself (the routers), not just at the network edges (the hosts).

- Hence the best-effort model, in which the network tries to deliver your data but makes no promises is not sufficient for real-time applications.
- What we need is a **new service model**, in which **applications that need higher assurances can ask the network for them**.
- The network may then respond by providing an assurance that it will do better .
- This implies that the network will treat some packets differently from others—something that is not done in the best-effort model.
- **A network that can provide these different levels of service is often said to support quality of service (QoS).**

Taxonomy of Real-Time Applications

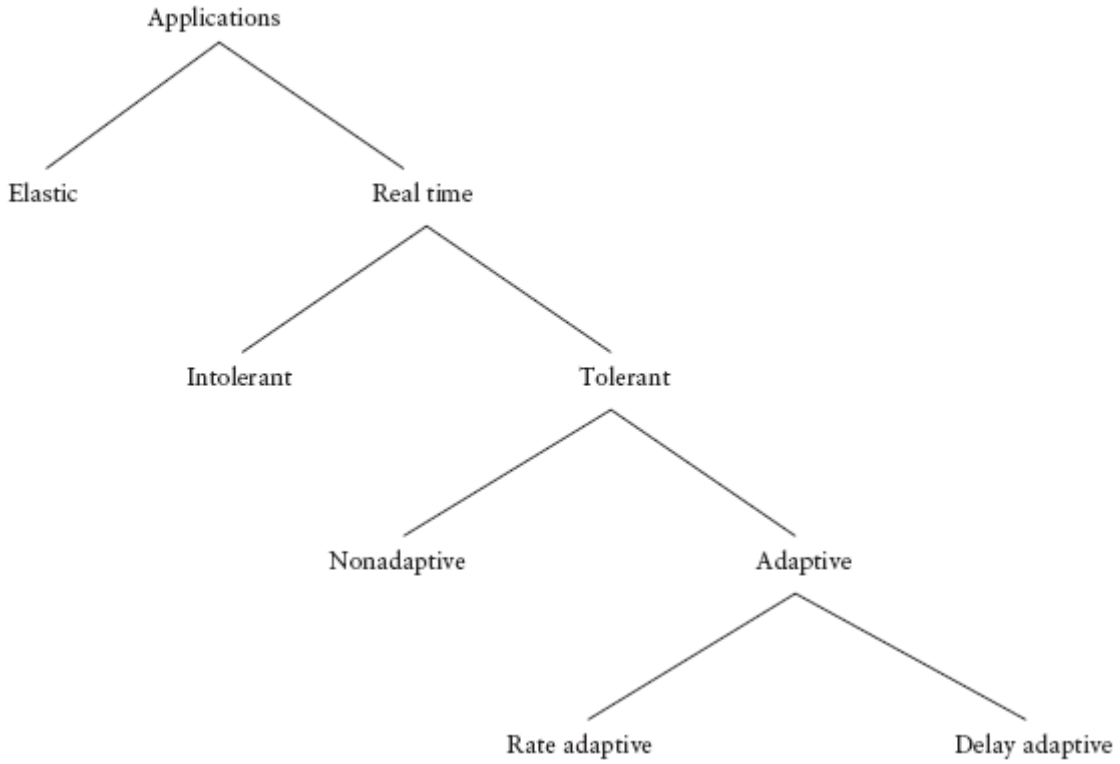

Figure 4.8 Taxonomy of Applications

- The taxonomy of applications is summarized in Figure 4.8
  1. **Tolerant and Intolerant Real –time Applications**
     - The first characteristic by which we can categorize applications is their tolerance of loss of data, where "loss" might occur because a packet arrived too late to be played back as well as arising from the usual causes in the network.
     - For example, one **lost audio sample** can be interpolated from the surrounding samples with relatively little effect on the perceived audio quality**(can tolerate loss)** whereas **a robot control program** is likely to be an example of a **real-time application that cannot tolerate loss**—

losing the packet that contains the command instructing the robot arm to stop is unacceptable.

- Thus, we can categorize real-time applications as tolerant or intolerant depending on whether they can tolerate occasional loss.

2. **Non-adaptive and Adaptive Real –time Applications**

- A second way to characterize real-time applications is by their adaptability.
- For example, an audio application might be able to adapt to the amount of delay that packets experience as they traverse the network.
- If we notice that packets are almost always arriving within 300 ms of being sent, then we can set our playback point accordingly.
- Suppose that we subsequently observe that all packets are arriving within 100 ms of being sent.
- If we moved up our playback point to 100 ms, then the users of the application would probably perceive an improvement.
- The process of shifting the playback point would actually require us to play out samples at an increased rate for some period of time.
- Thus, playback point adjustment is fairly easy in this case, and it has been effectively implemented for several voice applications.

3. **Rate Adaptive and Delay Adaptive Real –time Applications**

- We call applications that can adjust their playback point delay-adaptive applications.
- Another class of adaptive applications are rate adaptive.
- For example, many video coding algorithms can trade off bit rate versus quality.
- Thus, if we find that the network can support a certain bandwidth, we can set our coding parameters accordingly.
- If more bandwidth becomes available later, we can change parameters to increase the quality.

Approaches to QoS Support

Some of the approaches that have been developed to provide a range of qualities of service, are classified into two broad categories:

- **fine-grained approaches**, which provide QoS to individual applications or flows
- **coarse-grained approaches**, which provide QoS to large classes of data or aggregated traffic.

Integrated Services (RSVP)

- The term "Integrated Services" (often called IntServ for short) refers to a body of work that was produced by the IETF around 1995–97.

- The IntServ working group developed specifications of a number of service classes designed to meet the needs of some of the application types described above.
- It also defined how RSVP could be used to make reservations using these service classes.

1. Service Classes

- One of the service classes is designed for intolerant applications.
- These applications require that a packet never arrive late.
- The network should guarantee that the maximum delay that any packet will experience has some specified value.
- The application can then set its playback point so that no packet will ever arrive after its playback time.
- We assume that early arrival of packets can always be handled by buffering.
- This service is referred to as the **guaranteed service**.
- The service that meets the needs of tolerant, adaptive applications is known as **controlled load** .

2. Overview of Mechanisms

a) **Flowspec**

- We need to tell the network something about what we are going to inject into it, since a low-bandwidth application is going to require fewer network resources than a high-bandwidth application.
- The set of information that we provide to the network is referred to as a **flowspec**.
- There are two separable parts to the flowspec:
- the part that describes the **flow's traffic characteristics** (called the **TSpec**)
- and the part that describes the **service requested from the network** (the **RSpec**).
- The RSpec is very service specific.
- The TSpec is a little more complicated. We need to give the network enough information about the bandwidth used by the flow to allow intelligent admission control decisions to be made.

For Example,

- Suppose that we have 10 flows that arrive at a switch on separate input ports and that all leave on the same 10-Mbps link.
- Assume that over some suitably long interval each flow can be expected to send no more than 1 Mbps.
- You might think that this presents no problem.
- However, if these are variable bit rate applications, such as compressed video, then they will occasionally send more than their average rates.
- If enough sources send at above their average rates, then the total rate at which data arrives at the switch will be greater than 10 Mbps.
- This excess data will be queued before it can be sent on the link.
- The longer this condition persists, the longer the queue will get.
- Packets might have to be dropped, and even if it doesn't come to that, data sitting in the queue is being delayed.
- If packets are delayed long enough, the service that was requested will not be provided.

- One way to describe the bandwidth characteristics of sources is called a **token bucket filter**.
- Such a filter is described by **two parameters**: a **token rate r** and a **bucket depth B**. It works as follows.

    - To be able to send a byte, I must have a token.
    - To send a packet of length n, I need n tokens.
    - I start with no tokens and I accumulate them at a rate of r per second.
    - I can accumulate no more than B tokens.
    - What this means is that I can send a burst of as many as B bytes into the network as fast as I want, but over a sufficiently long interval, I can't send more than r bytes per second.
    - It turns out that this information is very helpful to the admission control algorithm when it tries to figure out whether it can accommodate a new request for service.

b) Admission Control

- For example, if 10 users ask for a service in which each will consistently use 2 Mbps of link capacity, and they all share a link with 10-Mbps capacity, the network will have to say no to some of them.
- **The process of deciding when to say no is called admission control.**
- The idea behind admission control is simple.
- When some new flow wants to receive a particular level of service, admission control looks at the TSpec and RSpec of the flow and tries to decide if the desired service can be provided to that amount of traffic, given the currently available resources, without causing any previously admitted flow to receive worse service than it had requested.
- If it can provide the service, the flow is admitted; if not, then it is denied.
- The hard part is figuring out when to say yes and when to say no.
- Admission control is very dependent on the type of requested service and on the queuing discipline employed in the routers.

c) Resource Reservation

- The mechanism by which the users of the network and the components of the network itself exchange information such as requests for service, flowspecs, and admission control decisions is called **resource reservation**.
- It is achieved using a **Resource Reservation Protocol**.

d) Packet Scheduling

- Finally, when flows and their requirements have been described, and admission control decisions have been made, the network switches and routers need to meet the requirements of the flows.
- A key part of meeting these requirements is managing the way packets are queued and scheduled for transmission in the switches and routers.

- This last mechanism is **packet scheduling.**

Differentiated Services

- The Integrated Services architecture allocates resources to individual flows, the Differentiated Services model (often called DiffServ for short) allocates resources to a small number of classes of traffic.
- In fact, some proposed approaches to DiffServ simply divide traffic into two classes.
- Suppose that we have decided to enhance the best-effort service model by adding
- just one new class, which we'll call **"premium."**
- Clearly, we will need some way to figure out which packets are premium and which are regular old best effort.
- This could obviously be done by using a bit in the packet header—if that bit is a 1, the packet is a premium packet; if it's a 0, the packet is best effort.

  - **Who sets the premium bit, and under what circumstances?**

    The router at the edge of anInternet service provider's network might set the bit for packets arriving on an interface that connects to a particular company's network. The Internet service provider mightdo this because that company has paid for a higher level of service than best effort.

  - **What does a router do differently when it sees a packet with the bit set?**

    The Differentiated Services working group of the IETF is standardizing a set of router behaviors to be applied to marked packets. These are called **"per-hop behaviors".**
    (PHBs), a term that indicates that they define the behavior of individual routers rather than end-to-end services.