

---

## Chapter 5

### End-to-End Protocols

---

---

## Transport Level

---

- Underlying best-effort network
  - drop messages
  - re-orders messages
  - delivers duplicate copies of a given message
  - limits messages to some finite size
  - delivers messages after an arbitrarily long delay

---

## Transport Level

---

- **Transport level protocols:** support communication between the end application programs (the **end-to-end** protocol)
- Some properties are expected to provide for transport protocols:
  - **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

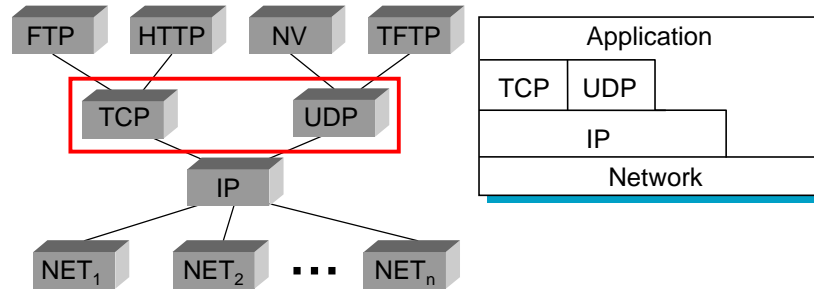
---

## Simple Demultiplexer (UDP)

---

## Internet Architecture

- The Internet architecture is also called the **TCP/IP architecture**
- The transport protocols are
  - UDP protocol
  - TCP protocol

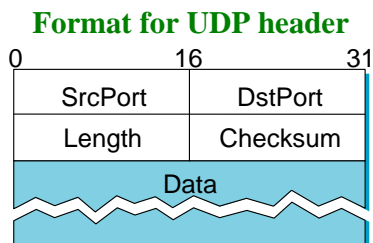


## Simple Demultiplexer

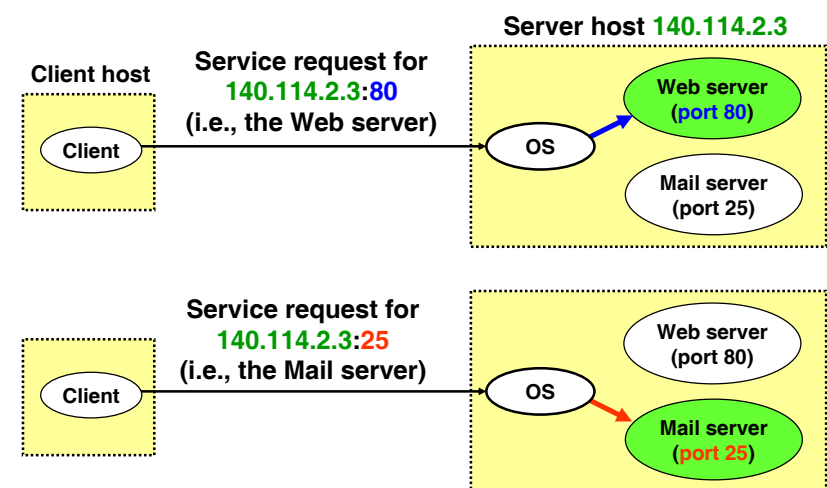
- The **simplest** transport protocol extends the host-to-host delivery service of the underlying network into a **process-to-process** communication service
  - Many processes running on any given host
  - A level of **demultiplexing** is required for multiple processes on each host to share the network
  - The simplest transport protocol adds **no other functionality** to the **best-effort service** provided by the underlying network
- The Internet's **User Datagram Protocol (UDP)** is an example of such a transport protocol
- The only issue is the form of the address used to **identify the target process**

## Simple Demultiplexer (UDP)

- The approach used by UDP is using an **abstract locator**
  - Called a **port** or **mailbox**
  - For a source process to send a message **to a port**, or for a destination process to receive the message **from a port**
- The **UDP port field** is **16 bits** long  $\Rightarrow$  up to **64 K** possible ports on a single host



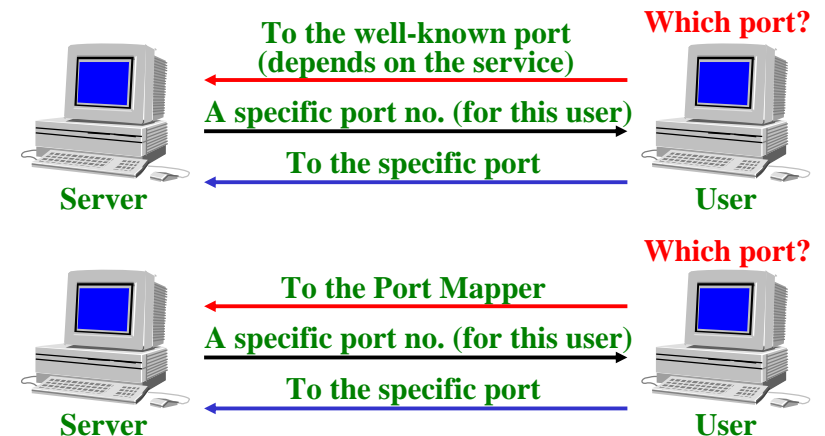
## Port



## Simple Demultiplexer (UDP)

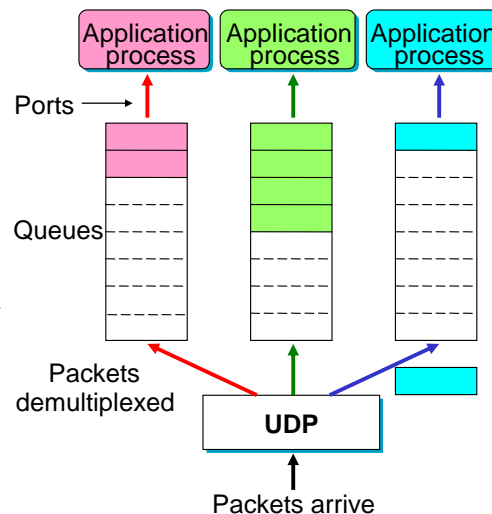
- How does the client learn the server's port in the first place?
- A common approach is for the server to accept messages at a **well-known port**, i.e. some fixed port widely published
  - Domain Name Server (DNS): port 53**
  - The mail server: port 25**
  - The Unix talk program: port 517**
- A well-known port is the **starting point** for communication:
  - The client and server use the well-known port to **agree on some other port** for subsequent communications
- An alternative strategy is using only a well-known port for the **Port Mapper** service to accept messages
  - A client send a message to ask for the port it should use

## Simple Demultiplexer (UDP)



## Simple Demultiplexer (UDP)

- A port is implemented by a **message queue**
- For an arrived message, the protocol appends it to the end of the queue
- When a 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



## Reliable Byte Stream (TCP)

## Reliable Byte Stream (TCP)

- A **reliable, connection-oriented, byte-stream service**:
  - Do not need to worry about **missing** or **reordered** data
- **TCP**: the Internet's **Transmission Control Protocol**
  - Guarantees the **reliable, in-order delivery** of a stream of bytes
  - A **full-duplex** protocol: each TCP connection supports a pair of byte streams
  - A **flow-control** mechanism: allows the receiver to limit the amount of data that the sender can transmit at a given time
  - A **demultiplexing** mechanism
  - A **congestion-control** mechanism

## End-to-End Issue (Variant RTT)

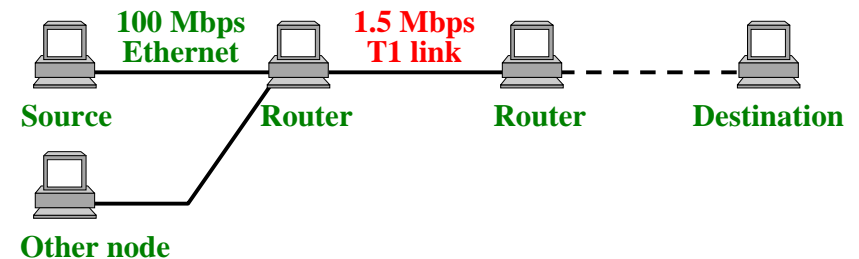
- The **sliding window algorithm** in TCP runs over the Internet
  - Which is quite different to point-to-point link
- TCP needs an explicit **connection establishment phase**
  - The two sides agree to exchange data with each other
  - The two parties establish some **shared state** to enable the sliding window algorithm to begin
- TCP also has an explicit **connection teardown phase**
  - For each host to know it is OK to **free this state**
- Different connections may have **widely different RTTs**
  - The TCP protocol must be able to support all conditions with different round-trip times
  - The **timeout mechanism** that triggers retransmissions must be **adaptive**

## End-to-End Issue (Flow-control)

- The packets may be **reordered** as they cross the Internet
    - Packets that are **slightly out of order** can be correctly reordered by using the **sequence number**
    - If a packet is delayed until IP's time to live (**TTL**) **field expires**, the packet will be **discarded**
  - The amount of **resources** dedicated to any one TCP connection is **highly variable**
    - Each side must **"learn"** what resources (e.g. buffer space) **the other side** is able to apply to the connection
- ⇒ The **flow-control** mechanism

## End-to-End Issue (Network Congestion)

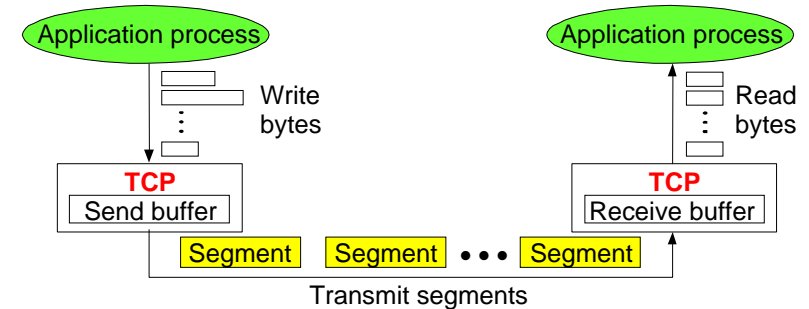
- The sending side of a TCP connection has no idea what links will be traversed to reach the destination
  - 100 Mbps fast Ethernet ↔ 1.5 Mbps T1 link ↔ ...
  - This leads to the problem of **network congestion**



## Segment Format

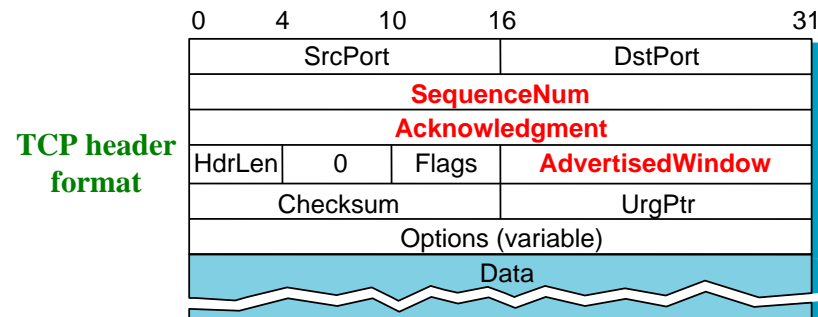
## Segment

- TCP connection supports **byte streams flowing** in both direction
  - The source host buffers enough bytes from the sending process to fill a reasonably sized packet
  - The packet is called **segment**



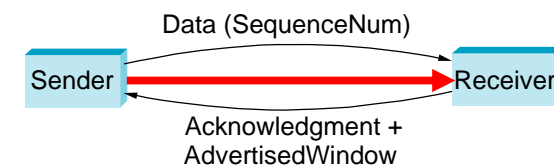
## Segment Format

- SrcPort** and **DstPort**:
  - The source and destination ports
- Acknowledgment, SequenceNum, and AdvertisedWindow**:
  - All involved in TCP's sliding window algorithm



## Segment Format

- SequenceNum**:
  - Contains the sequence number for the **first byte of data** carried in the segment
  - Each byte of data has a sequence number
- Acknowledgment** and **AdvertisedWindow**:
  - Carry information about the flow of data going in **the other direction**



---

## Segment Format

---

- **HdrLen field:**
  - The length of the header in **32-bit words**
- The **6-bit Flags field:**
  - Used to relay **control information** between TCP peers
- **UrgPtr field:**
  - Indicates where the **nonurgent data** contained in this segment begins
  - **Urgent data** is contained in the front of a segment
- **Checksum field:**
  - Error detection

---

## Segment Format

---

- The possible flags include **SYN, FIN, RESET, PUSH, URG** and **ACK** (**6 bits**  $\Rightarrow$  **6 flags**)
  - **SYN:** is used when **establishing** a TCP connection
  - **FIN:** is used when **terminating** a TCP connection
  - **RESET:** is used when the receiver has become confused, and so wants to **abort the connection**
  - **PUSH:** is used when the sending process **invokes the push operation** to efficiently flush the buffer of unsent bytes
  - **URG:** is used when this segment contains **urgent data**
  - **ACK:** is set when the **Acknowledgment** field is valid

---

## Connection Establishment and Termination

---

## Connection Establishment and Termination

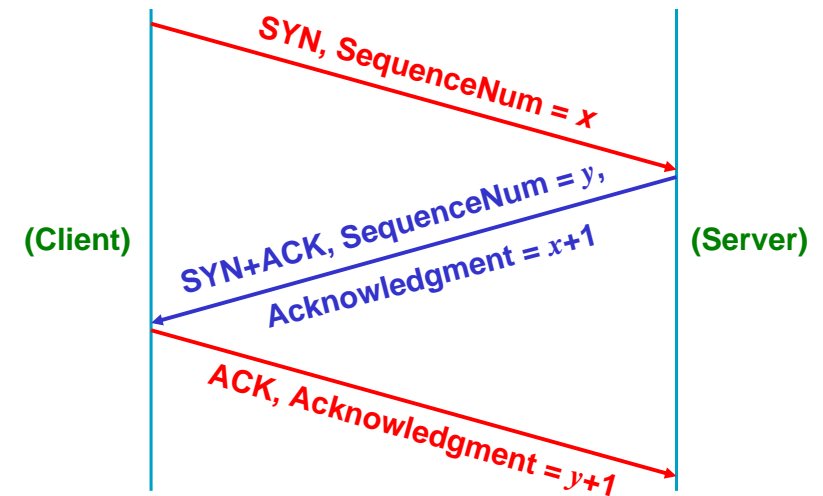
---

- A TCP connection begins with a **client** (caller) doing an active open to a **server** (callee)
- The two sides engage in an **exchange of messages** to establish the connection
- Only after this connection establishment phase is over, the two sides can begin sending data
- The algorithm used by TCP to establish and terminate a connection is called a **three-way handshake**
  - Involves **the exchange of three messages** between the client and the server

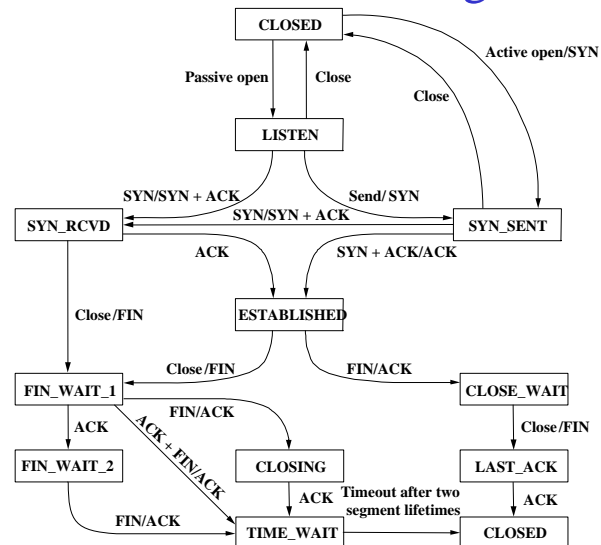
## Connection Establishment and Termination

- The client sends a segment to the server stating the **initial sequence number**
  - Flags = **SYN**, SequenceNum =  **$x$**
- The server responds with a single segment
  - To **acknowledge** the client's sequence number
    - Flags = **ACK**, Ack =  **$x+1$**  (next sequence number expected is  $x+1$ )
  - To state its own **beginning sequence number**
    - Flags = **SYN**, SequenceNum =  **$y$**
- The client responds with a segment that **acknowledges** the server's sequence number
  - Flags = **ACK**, Ack =  **$y+1$**

## Connection Establishment and Termination



## State Transition Diagram



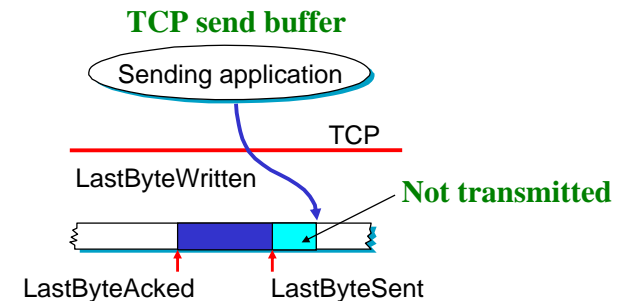
## Sliding Window Algorithm

## Sliding Window Algorithm

- **TCP sliding window algorithm:**
  - It guarantees the **reliable delivery** of data
  - It ensures that data is delivered **in order**
  - It enforces **flow control** between the sender and the receiver
- Rather than having a fixed-size sliding window, the receiver **advertises a window size** to the sender
  - Based on the **amount of memory** allocated to the connection for the purpose of buffering data
  - Using the **AdvertisedWindow** field in the TCP header
- The sender is limited to having **no more than** a value of AdvertisedWindow bytes of **unacknowledged data**

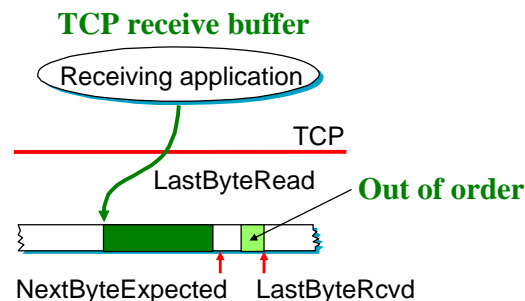
## Sliding Window Algorithm (Sending Side)

- TCP on the sending side maintains a **send buffer** used to store
  - The data that **has been sent** but **not yet acknowledged**
  - The data that **has been written** by the sending application, but **not transmitted**



## Sliding Window Algorithm (Receiving Side)

- TCP on the receiving side maintains a **receive buffer** used to hold
  - The data that **arrives out of order**
  - The data that is in the correct order, but that the application process **has not yet had the chance to read**



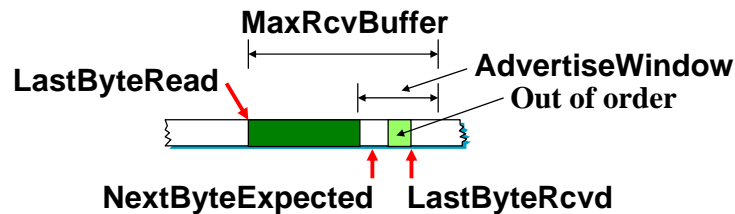
## Sliding Window Algorithm

- In the sending side, three pointers are maintained into the send buffer: **LastByteAked**, **LastByteSent**, and **LastByteWritten**
  - **LastByteAked ≤ LastByteSent**
  - **LastByteSent ≤ LastByteWritten**
- In the receiving side, three pointers are maintained into the receive buffer: **LastByteRead**, **NextByteExpected**, and **LastByteRcvd**
  - **LastByteRead < NextByteExpected**
  - **NextByteExpected ≤ LastByteRcvd + 1**
    - “=” holds when there is no out of order byte



## Flow Control (Receive Buffer)

- The buffer sizes are finite: **MaxSendBuffer**, **MaxRcvBuffer**
- To avoid overflowing the **receive buffer**
  - $\text{LastByteRcvd} - \text{LastByteRead} \leq \text{MaxRcvBuffer}$
- The receiver advertises a window size representing the amount of **free space** remaining in its buffer
  - $\text{AdvertiseWindow} = \text{MaxRcvBuffer} - ((\text{NextByteExpected} - 1) - \text{LastByteRead})$

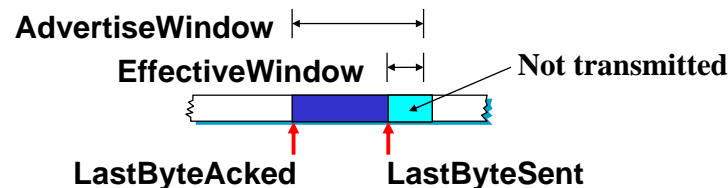


## Flow Control (Receive Buffer)

- If the local process is reading data just **as fast as it arrives**
  - The advertised window stays open
    - AdvertiseWindow = MaxRcvBuffer**
- If the receiving process **falls behind**
  - The advertised window grows smaller until it goes to **0**

## Flow Control (Receive Buffer)

- TCP on the sending side must ensure that
  - $\text{LastByteSent} - \text{LastByteAked} \leq \text{AdvertiseWindow}$
- To avoid overflowing the **receive buffer**, the sender computes an effective window that limits how much data it can send:
  - $\text{EffectiveWindow} = \text{AdvertiseWindow} - (\text{LastByteSent} - \text{LastByteAked})$

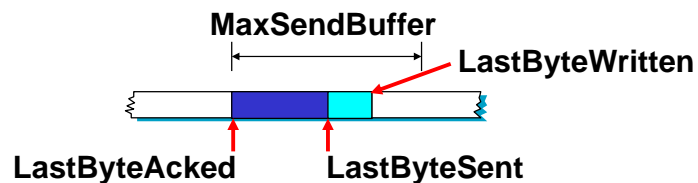


## Flow Control (Receive Buffer)

- EffectiveWindow** must be **greater than 0** before the source can send more data
- If a segment arrives acknowledging **x** bytes and the receiving process **was not** reading any data
  - The receive buffer **does not** free any buffer space
  - The advertise window is **x** bytes smaller
  - The sender can increase **LastByteAked** by **x**
  - The sender would be able to **free** buffer space, but **not to send** any more data

## Flow Control (Send Buffer)

- The sending side must also make sure that the local application process **does not overflow** the send buffer
  - $\text{LastByteWritten} - \text{LastByteAked} \leq \text{MaxSendBuffer}$
- If the sending process tries to write  $y$  bytes to TCP, but
  - $\text{LastByteWritten} - \text{LastByteAked} + y > \text{MaxSendBuffer}$
  - Then TCP **blocks** the sending process



## Flow Control (Send Buffer)

- A **slow** receiving process ultimately stops a **fast** sending process
  - The receive buffer fills up
  - ⇒ The advertise window shrinks to 0
  - ⇒ The sending side cannot transmit any data
  - ⇒ The send buffer fills up
  - ⇒ TCP blocks the sending process
- TCP is designed to make the receive side **as simple as possible**
  - It simply responds to segments from the sender

## Flow Control

- How does the sending side know that **the advertised window is no longer 0**?
- TCP **always** sends a segment in response to a received segment
  - Contains the latest values for the **Acknowledge** and **AdvertiseWindow** fields
- Whenever the receiving side advertises a window size of 0
  - The sending side persists in sending a **probe segment** with **1 byte** of data
  - Each probe segment **triggers a response** containing the **current** advertised window
  - Eventually, a response reports a **nonzero** advertised window

## Protection Against Wrap Around

- 32-bit **SequenceNum**

Bandwidth	Time Until Wrap Around
T1 (1.5 Mbps)	6.4 hours
Ethernet (10 Mbps)	57 minutes
T3 (45 Mbps)	13 minutes
FDDI (100 Mbps)	6 minutes
STS-3 (155 Mbps)	4 minutes
STS-12 (622 Mbps)	55 seconds
STS-24 (1.2 Gbps)	28 seconds

## Keeping the Pipe Full

- 16-bit **AdvertisedWindow**

Bandwidth	Delay x Bandwidth Product
T1 (1.5 Mbps)	18KB
Ethernet (10 Mbps)	122KB
T3 (45 Mbps)	549KB
FDDI (100 Mbps)	1.2MB
STS-3 (155 Mbps)	1.8MB
STS-12 (622 Mbps)	7.4MB
STS-24 (1.2 Gbps)	14.8MB

41

## Triggering Transmission

Dr. Tsai

## Triggering Transmission

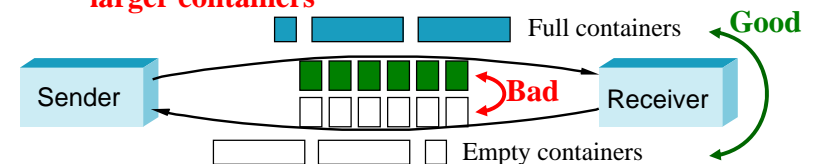
- TCP has three mechanisms to trigger the transmission of a segment
  - It sends a segment as soon as it has **collected MSS (maximum segment size)** bytes from the sending process
    - MSS is generally set to the size of the largest segment TCP can send **without causing IP fragmentation**
  - It sends a segment when the **sending process** has **asked** it to do so
    - TCP supports a **PUSH operation** and the sending process invokes it to **flush** the buffer of unsent bytes
  - It sends a segment when a **timer fires**
    - The resulting segment contains **all** bytes that are currently buffered for transmission

Dr. Tsai

43

## Triggering Transmission

- Data segment:** full containers; **ACKs:** empty containers;
  - MSS-sized** segments: large container; **1-byte** segments: small container
- Silly window syndrome:** If the sender aggressively fills an empty container **as soon as it arrives**
  - Any **small container** introduced into the system remains in the system indefinitely
  - It never coalesces with adjacent containers to create **larger containers**



Dr. Tsai

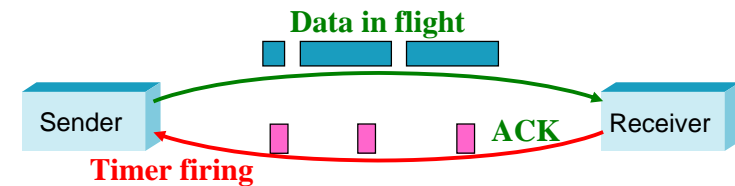
44

## Triggering Transmission (Window Size)

- Triggering transmission is applied to keep the receiver from introducing a small container:
  - After advertising a zero window, the **receiver** must **wait for space equal to an MSS** before it advertises an open window
- Some mechanisms are also introduced to coalesce small containers
  - The receiver can do this by **delaying ACKs** — sending one **combined ACK** rather than multiple smaller ones
    - Reply a large window size

## Triggering Transmission (Sender)

- If there is data to send but the window is open **less** than MSS
  - It waits some amount of time before sending the data:
    - Introduce a **timer**
  - It transmits when the timer expires
- A **self-clocking** solution: **Nagle's algorithm**
  - If TCP has any data in flight, the sender will eventually receive an ACK — treated like a timer firing



## Triggering Transmission (Sender)

- **Nagle's algorithm:**
  - It's **always OK** to send a **full** segment if the window allows
  - It's OK to send a small amount of data if there are currently **no segments in transit**
  - If there is anything in flight, the sender must **wait for an ACK** before transmitting the next segment

## Adaptive Retransmission

## Adaptive Retransmission

- TCP **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 the RTTs are **various with time**, TCP uses an **adaptive retransmission mechanism**
  - To keep a **running average** of the RTT
  - Then compute the timeout as a function of this RTT

## Adaptive Retransmission

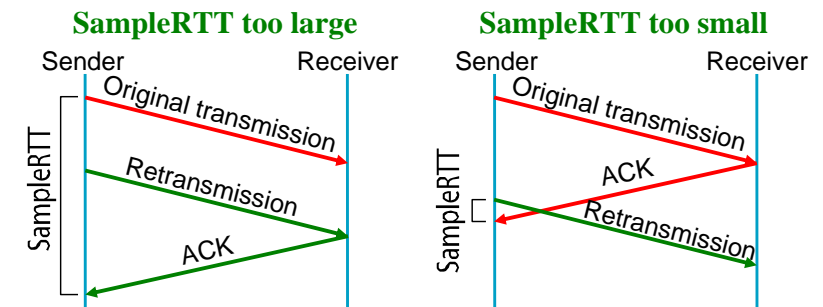
- Every time TCP sends a data segment, it records the time
- When an ACK for that segment arrives, TCP reads the time again and then takes the difference as a **SampleRTT**
- TCP then computes an **EstimatedRTT** as a **weighted average** between the previous estimate and this new sample
  - **EstimatedRTT** =  $\alpha \times \text{EstimatedRTT} + (1-\alpha) \times \text{SampleRTT}$
  - $\alpha$  is selected to **smooth** the **EstimatedRTT**
- TCP then uses **EstimatedRTT** to compute the timeout:
  - **TimeOut** =  $2 \times \text{EstimatedRTT}$

## Adaptive Retransmission

- The setting of  $\alpha$ :
  - A **small  $\alpha$**  tracks changes in the RTT but is heavily influenced by **temporary fluctuations**
  - A **large  $\alpha$**  is more **stable** but is not quick enough to adapt to real change
  - It recommended a setting of  $\alpha$  between **0.8 and 0.9**
- Problem: An ACK does not really acknowledge a transmission
  - It actually acknowledges the **receipt** of data

## Adaptive Retransmission

- Whenever a segment is **retransmitted** and then an ACK arrives at the sender
  - It is impossible to determine if this ACK should be associated with the **first** or the **second** transmission



## Adaptive Retransmission

- **Karn/Partridge algorithm:**

- Whenever TCP **retransmits** a segment, it **stops** taking samples of the RTT
- It only measures **SampleRTT** for segments that have been **sent only once**
- Each time TCP retransmits, it sets the next timeout to be **twice the last timeout** (rather than the last EstimatedRTT)
  - TCP use **exponential backoff**
- Problem: If the variation among samples is **small**
  - Then the EstimatedRTT can be **better trusted**
- If the variation among samples is **large**
  - Then the timeout value **should not** be too tightly coupled to the EstimatedRTT

## Adaptive Retransmission

- **Jacobson/Karels algorithm:**

- The sender measures a new **SampleRTT** as before
- The timeout is calculated as follows:  
**Difference = SampleRTT – EstimatedRTT**  
**EstimatedRTT = EstimatedRTT + ( $\delta \times \text{Difference}$ )**  
**Deviation = Deviation +  $\delta(|\text{Difference}| - \text{Deviation})$**
- $\delta$  is a fraction between **0 and 1**
- TCP then computes the timeout value as follows:  
**TimeOut =  $\mu \times \text{EstimatedRTT} + \phi \times \text{Deviation}$**
- $\mu$  is typically set to **1** and  $\phi$  is set to **4**
- When the variance is **small**, TimeOut is close to EstimatedRTT
- When the variance is **large**, Deviation will dominate TimeOut

## Transport for Real-Time Application (RTP)

## Real-time Transport Protocol (RTP)

- RTP contains a considerable amount of functionality that is specific to multimedia applications
  - Runs **on top** of one of the transport-layer protocols **UDP**
  - Provides **common end-to-end functions** to a number of applications
- Multimedia applications are sometimes divided into two classes:
  - **Conferencing applications**
  - **Streaming applications**
- RTP can run over many lower-layer protocols, but **commonly runs over UDP**

Application
RTP
UDP
IP
Subnet

**Protocol stack for multimedia applications using RTP**

## Requirements for RTP

- The most basic requirement for a general-purpose multimedia protocol is that it allow **similar applications** to **interoperate** with each other
  - Two **independently** implemented applications to communicate with each other
- Coding schemes agreement:** A sender tell a receiver the **used coding scheme**, and negotiate until a scheme is identified
  - There are only quite a few different coding schemes

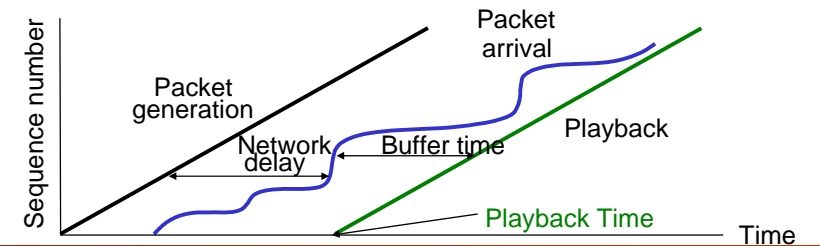


Dr. Tsai

57

## Requirements for RTP

- Timing:** To enable the recipient of a data stream to determine the **timing relationship** among the received data
  - Real-time applications:** need to place received data into a **playback buffer** to smooth out the jitter introduced into the data stream during transmission
    - Some sort of **timestamping** of the data is necessary for the receiver to play it back at the appropriate time



Dr. Tsai

58

## Requirements for RTP

- Synchronization:** To synchronize **multiple media** in a conference
  - For example to synchronize an **audio** and **video** stream that are originating from the same sender
- Indication of packet loss:** An application with **tight** latency bounds generally cannot use a reliable transport like TCP
  - Retransmission** of data to correct for loss would probably cause the packet to **arrive too late** to be useful
  - The application must be able to deal with **missing packets**
  - For example, a video application using MPEG encoding will need to take different actions when a packet is lost
    - Depending on whether the packet came from an **I frame**, a **B frame**, or a **P frame**

Dr. Tsai

59

## Requirements for RTP

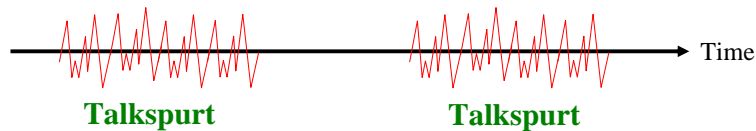
- Congestion-avoidance:** multimedia applications generally **do not run over TCP**
  - Miss out on the congestion-avoidance features of TCP
  - Multimedia applications should **respond to congestion**
    - For example, by changing the parameters of the coding algorithm to **reduce the bandwidth** consumed
  - The receiver needs to notify the sender that **losses** are occurring

Dr. Tsai

60

## Requirements for RTP

- **Frame boundary indication:**
  - Notify a video application that a certain set of packets **correspond to a single frame**
  - Mark the beginning of a “**talkspurt**,” which is a collection of sounds or words followed by **silence**
    - Identify the silences between talkspurts
    - Use them as opportunities to **move the playback point**
    - Slight **shortening** or **lengthening** of the spaces between words are not noticeable to users



## Requirements for RTP

- **Identifying senders:** Should be a way more **user-friendly** than an IP address
  - Such as display strings such as Joe User (**user@domain.com**)
- **Efficient use of bandwidth:** **Do not** introduce a lot of extra bits (**long header**) that need to be sent with every packet
  - Long packets would mean **high latency** due to packetization
  - Audio packets tend to be small
    - Bad bandwidth efficiency is obtained if long header is used

## RTP Details

- The RTP standard actually defines a pair of protocols
  - **Real-time Transport Protocol (RTP):** is used for the exchange of **multimedia data**
  - **Real-time Transport Control Protocol (RTCP):** is used to periodically send **control information** associated with a certain data flow
- When running over UDP, the RTP data stream and the associated RTCP control stream use **consecutive** transport-layer ports
  - The RTP data uses an **even** port number
  - The RTCP control information uses the **next higher (odd)** port number

## RTP Control Protocol

- This **control stream** provides three main functions:
  - To feedback data on the **performance** of the application and the network
  - To **correlate** and **synchronize** different media streams coming from the same sender
  - To convey **the identity of a sender** for display on a user interface
- The performance data is useful for **rate-adaptive** applications
  - Use a more **aggressive compression** scheme to reduce congestion
  - Send a **higher-quality** stream for little congestion