

I226

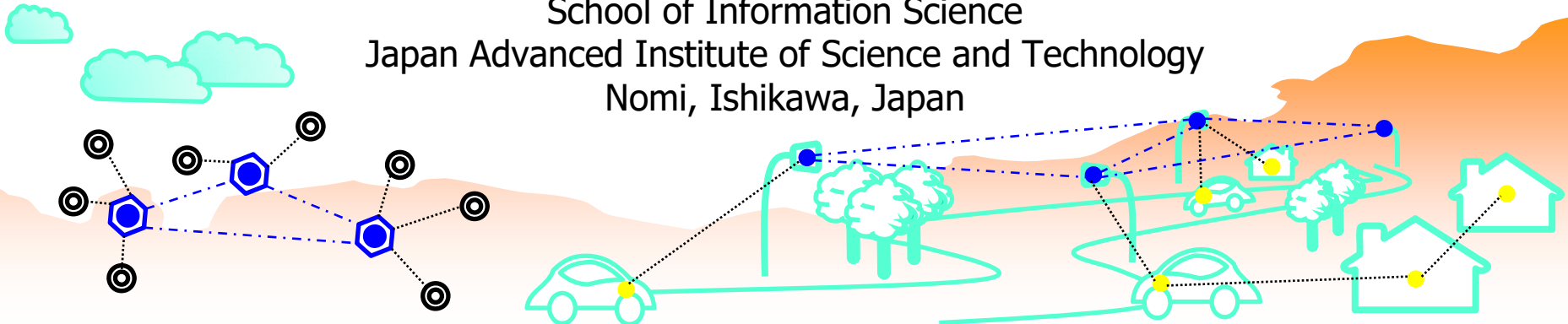
Computer Networks

Chapter 6

Transport Layer

Assoc. Prof. Yuto Lim

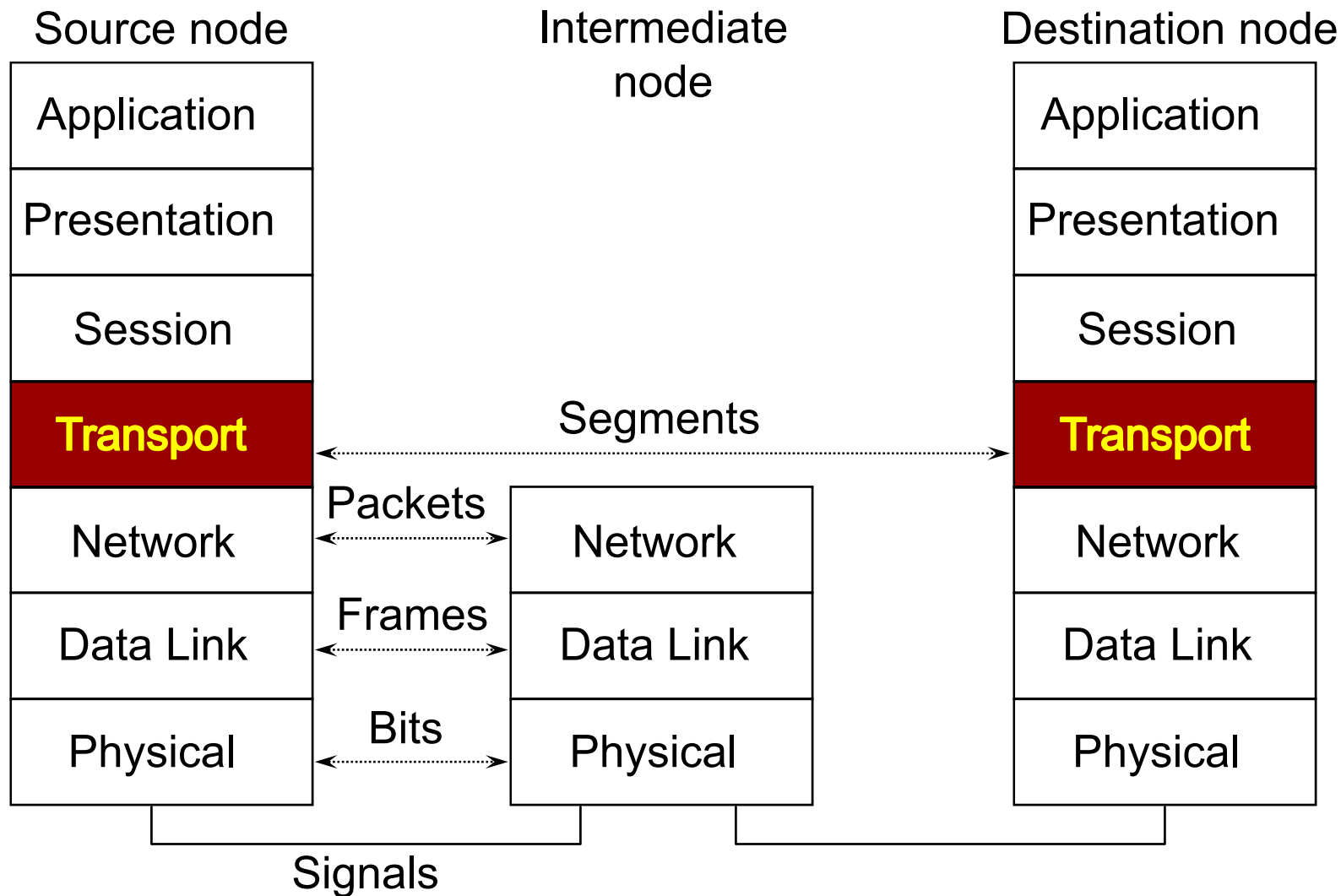
School of Information Science
Japan Advanced Institute of Science and Technology
Nomi, Ishikawa, Japan



Objectives of this Chapter

- Provide an understanding what is the transport layer and its functions
- Offer the knowledge of two transport layer protocols: UDP and TCP
- Explain the transport layer services for multiplexing, reliable data transfer, flow control and congestion control

Transport Layer

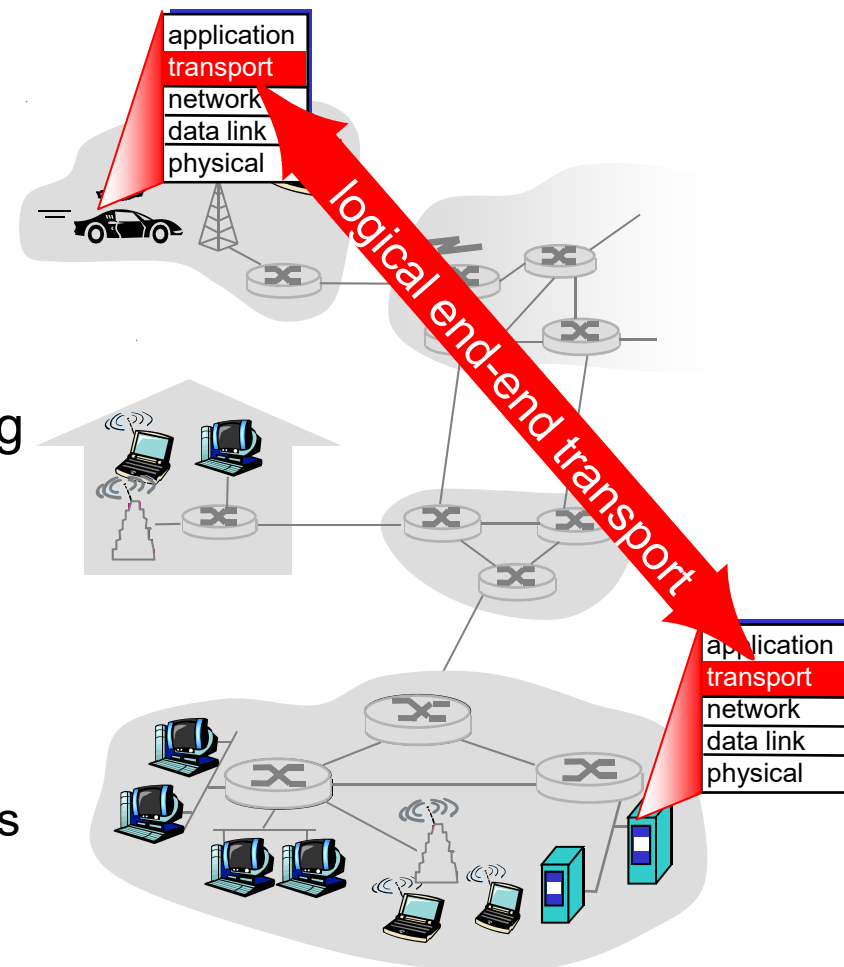


Outline

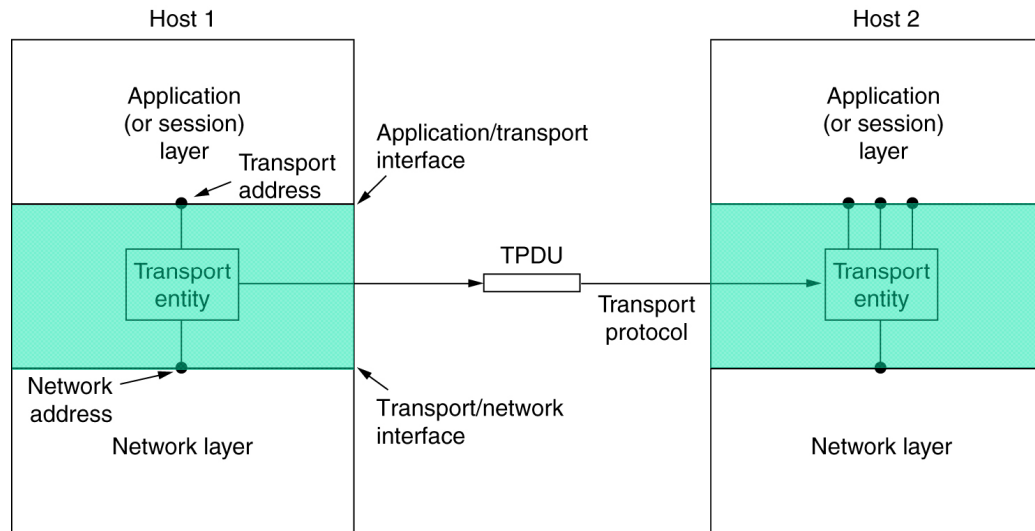
- Introduction to Transport Layer
- Transport Service Primitives
- Elements of Transport Protocols
- Transport Layer Protocols
 - UDP: Connectionless Transport
 - TCP: Connection-oriented Transport
- Transport Layer Services
 - Multiplexing/Demultiplexing
 - Reliable Data Transfer
 - Flow Control
 - Congestion Control

Transport Layer Services & Protocols

- Purpose: Ensure that data gets between A and B
- Provide **logical communication** between application processes running on different hosts
- Provide **flow control** if necessary (send data faster or slower depending on the network conditions)
- Transport protocols run in end systems
 - Sender side: breaks application messages into segments, passes to network layer
 - Receiver side: reassembles segments into messages, passes to application layer



Network, Transport & Application Layers



- Functions in the transport layer are
 - Connection setup and release
 - Addressing
 - Segmentation and reassembly
 - Multiplexing and demultiplexing
 - Error handling
 - Flow control
 - Congestion handling

Transport Layer vs. Network Layer

- Network layer: logical communication between **hosts**
- Transport layer: logical communication between **processes**
 - ▣ Relies on, enhances, network layer services

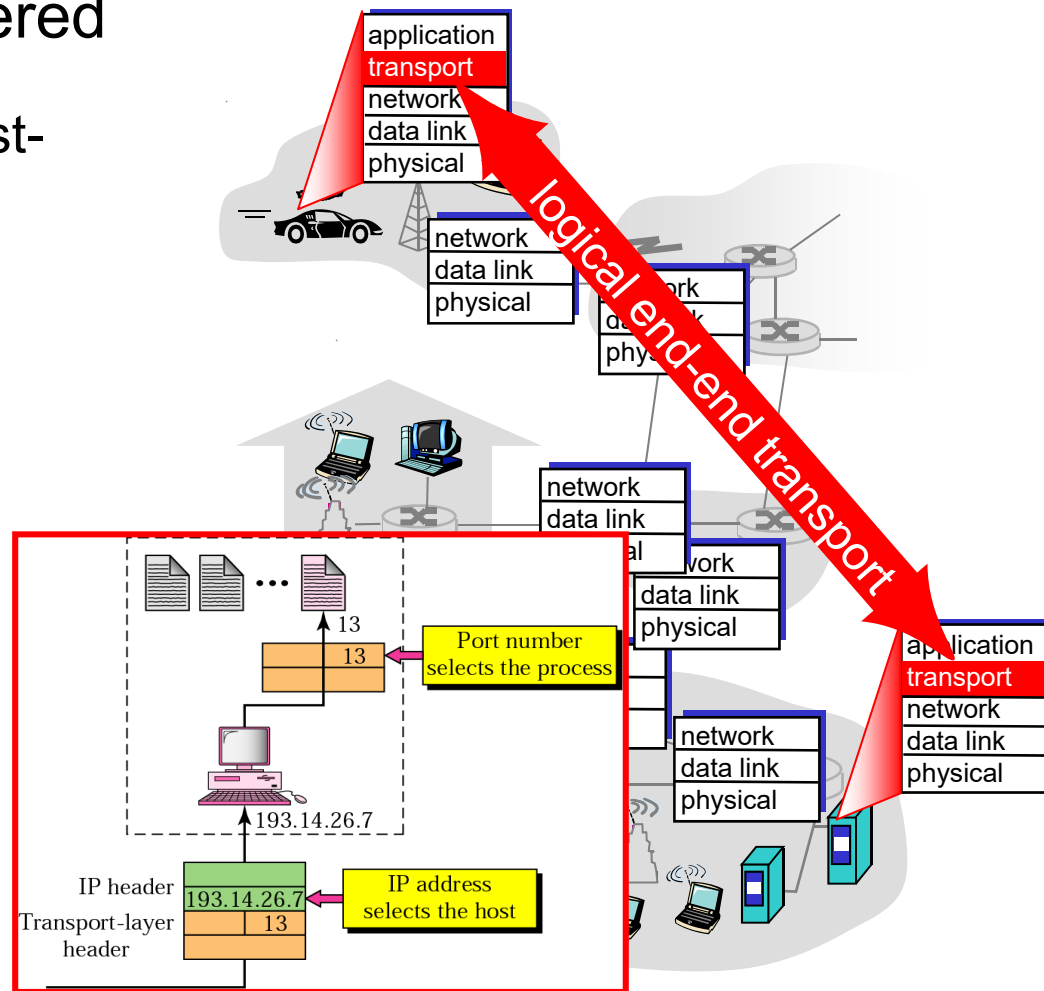
Household analogy:

12 kids sending letters to 12 kids

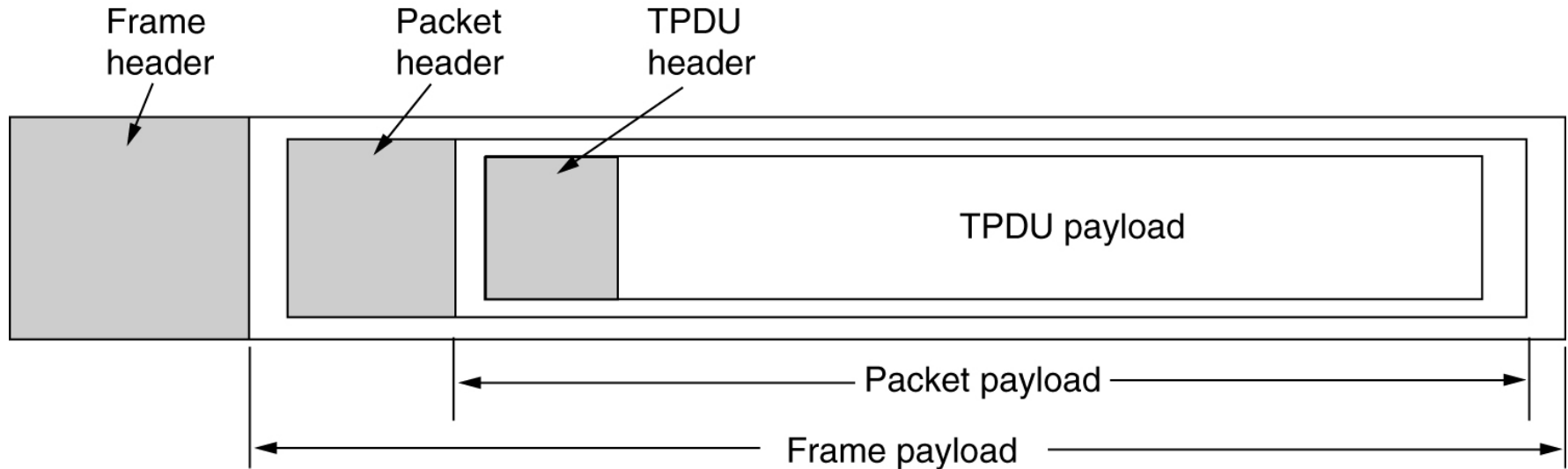
- Processes = kids
- Application messages = letters in envelopes
- hosts = houses
- Transport-layer protocol = Ann and Bill
- Network-layer protocol = postal service

Internet Transport Layer Protocols

- **UDP**: unreliable, unordered delivery
 - ▣ No-frills extension of “best-effort” IP
- **TCP**: reliable, in-order delivery
 - ▣ Connection setup
 - ▣ Flow control
 - ▣ Congestion control
- Services not available
 - ▣ Delay guarantees
 - ▣ Bandwidth guarantees



Transport Protocol Data Unit (TPDU)



- TPDU is sent from transport entity to transport entity
- TPDU is contained in **packets** (exchanged by network layer)
- Packets are contained in **frames** (exchanged by the data link layer)

Transport Service Primitives

- Transport layer can provide
 - ▣ **Connection-oriented** transport service, providing an error free bit/byte stream; reliable service on top of unreliable network
 - ▣ **Connectionless** unreliable transport (datagram) service
- Example of basic transport service primitives (see p33, ch1):

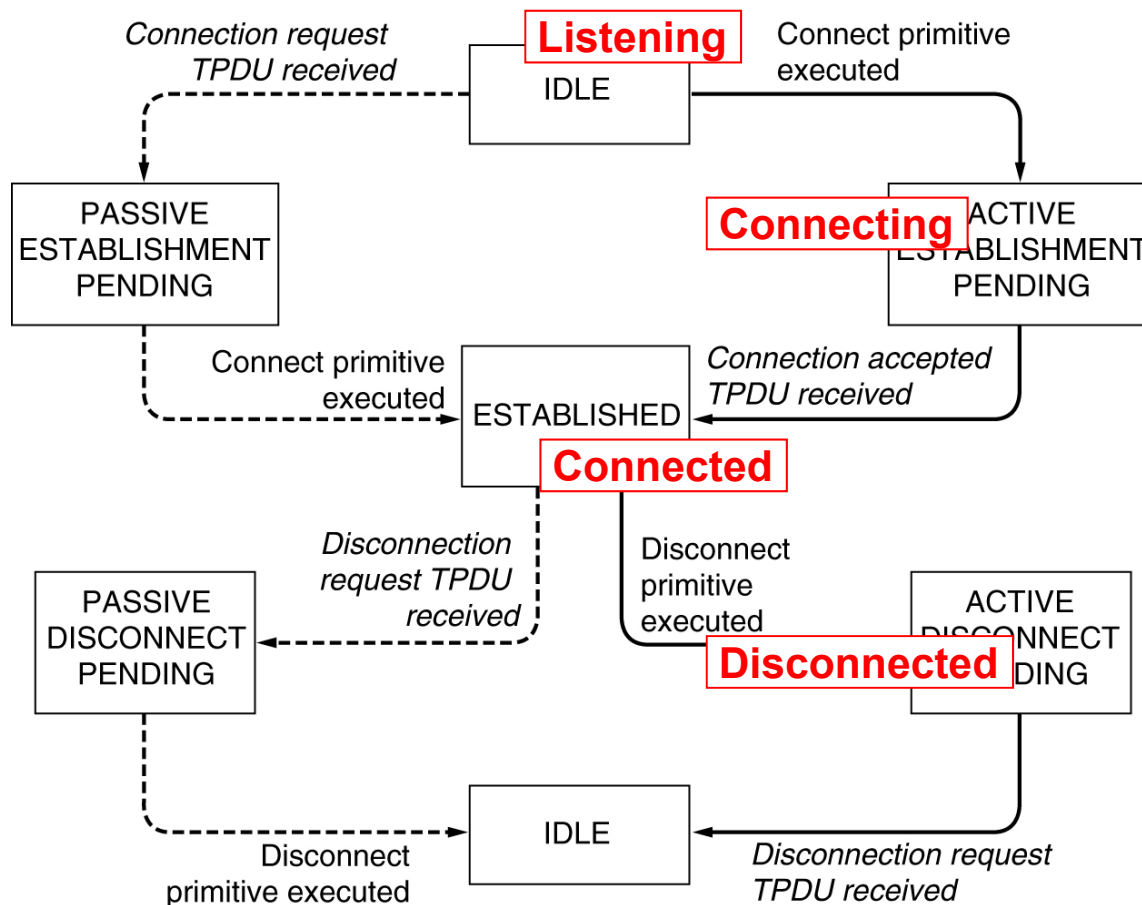
Primitive	Packet sent	Meaning
LISTEN	(none)	Block until some process tries to connect
CONNECT	CONNECTION REQ.	Actively attempt to establish a connection
SEND	DATA	Send information
RECEIVE	(none)	Block until a DATA packet arrives
DISCONNECT	DISCONNECTION REQ.	This side wants to release the connection

DISCONNECT – when a connection is no longer needed, it must be released in order to free up tables in the transport entities; it can be asymmetric (either end sends a *disconnection* TPDU to the remote transport entity; upon arrival, the connection is released) or symmetric (each direction is closed separately)

Finite State Machine (FSM) Modeling

- A basic system can be one of a finite no. of conditions (**states**)
- In the transport layer, the service primitives can be categorized into 4 main conditions:
 - Listening – waiting for something to happen
 - Connecting
 - Connected – whilst connected you can either be
 - sending
 - Receiving
 - Disconnected
- In this way, the problem is defined to be a limited no. of states with a finite no. of transitions between them, called **finite state machine** (FSM)
 - FSM is further discussed on **Chapter 11**

Connection Establishment and Release

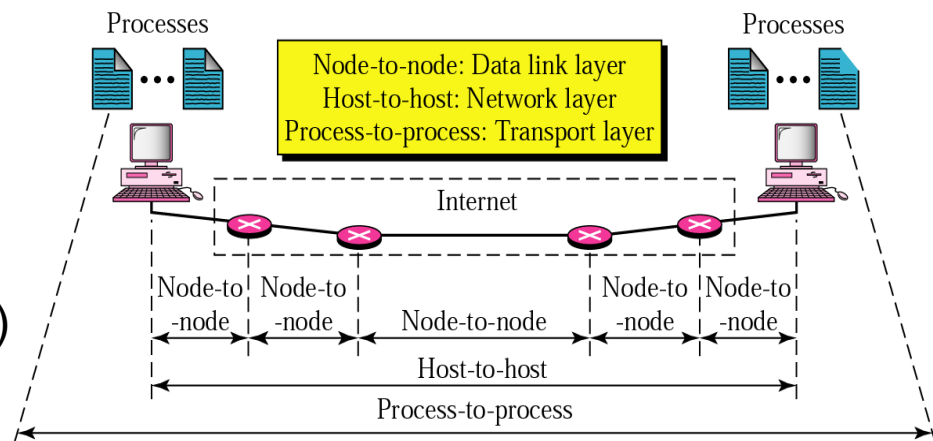
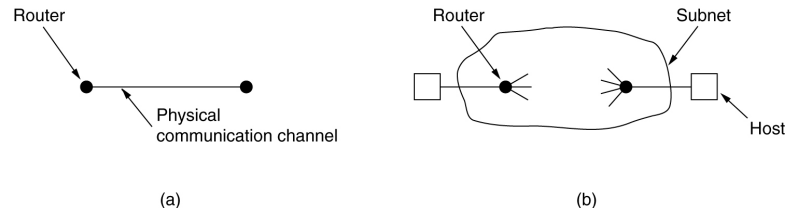


- Transitions in **italics** are caused by packet arrivals
- **Solid lines** show the client's state sequence
- **Dashed lines** show the server's sequence

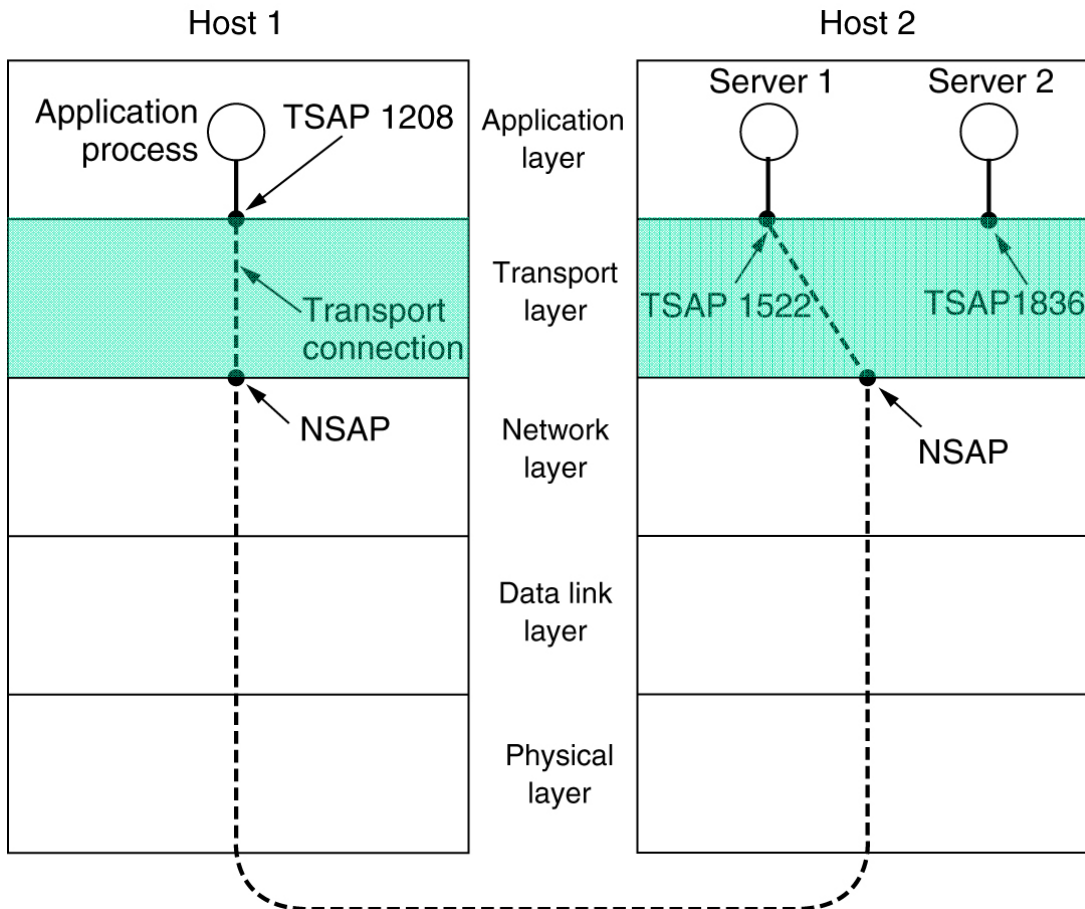
Figure: State diagram for connection establishment and release of a simple transport manager

Transport Protocol

- Transport protocols reassemble the data link protocols
 - ▣ Both have to deal with error control, sequencing and flow control
- Significant differences
 - ▣ At **data-link layer**, two routers communicate directly over a physical channel, see Fig. (a)
 - ▣ At **transport layer**, the physical channel is replaced by the subnet, see Fig. (b)
- Explicit (clear) addressing of the destination is required for transport layer
- Need for connection establishment and disconnection (in the data-link case the connection is always there)
- Packets get lost, duplicated or delayed in the subnet, the transport layer has to deal with this



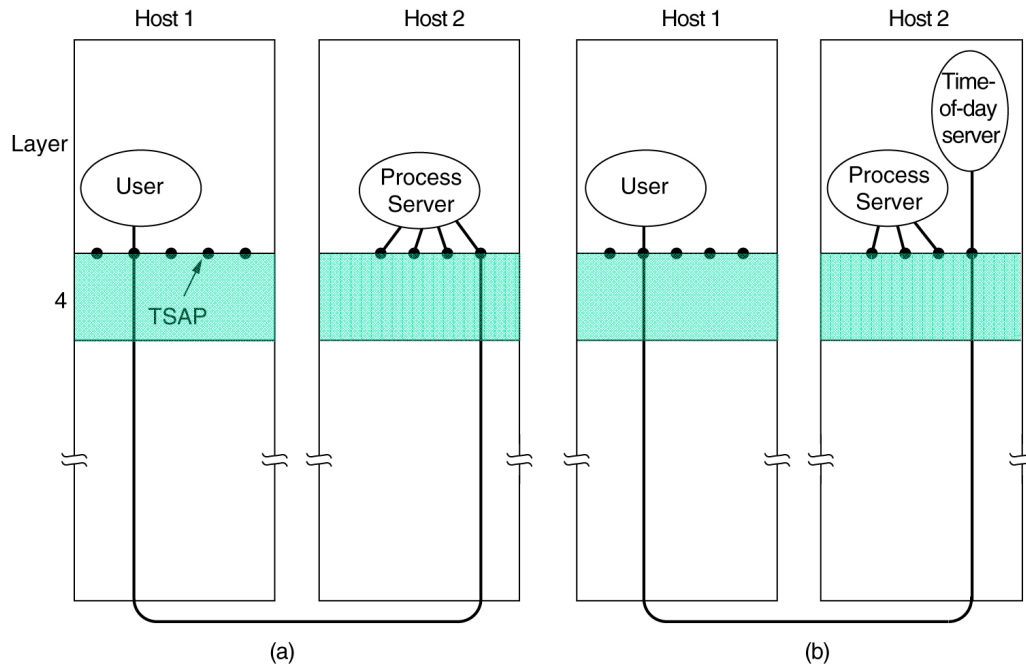
Addressing: Fixed TSAP



- To whom should each message be sent?
 - Have transport address to which processes can listen for connection requests (or datagrams in connectionless transports)
 - Host and process are identified Service Access Points, i.e., TSAP, NSAP
 - In Internet, TSAP is the **port** and NSAP is the **IP**

TSAP: **T**ransport Service Access Point
NSAP: **N**etwork Service Access Point

Addressing: Dynamic TSAP



How a user process in host 1 establishes a connection with a time-of-day server in host 2?

- Initial connection protocol
 - Process server that acts as a **proxy** server for less-heavily used servers
 - Listens on a set of ports at the same time for a number of incoming TCP connection requests
 - Spawns the requested server, allowing it to inherit the existing connection with the user
- Name server (directory server)
 - Handles situations where the services exist independently of the process server (i.e. file server, needs to run on special hardware and cannot just be created on the fly when someone wants to talk to it)
 - Listens to a well known TSAP
 - Gets message specifying the service name and sends back the TSAP address for the given service
 - Services need to register themselves with the name server, giving both its service name and its TSAP address

Connection Establishment

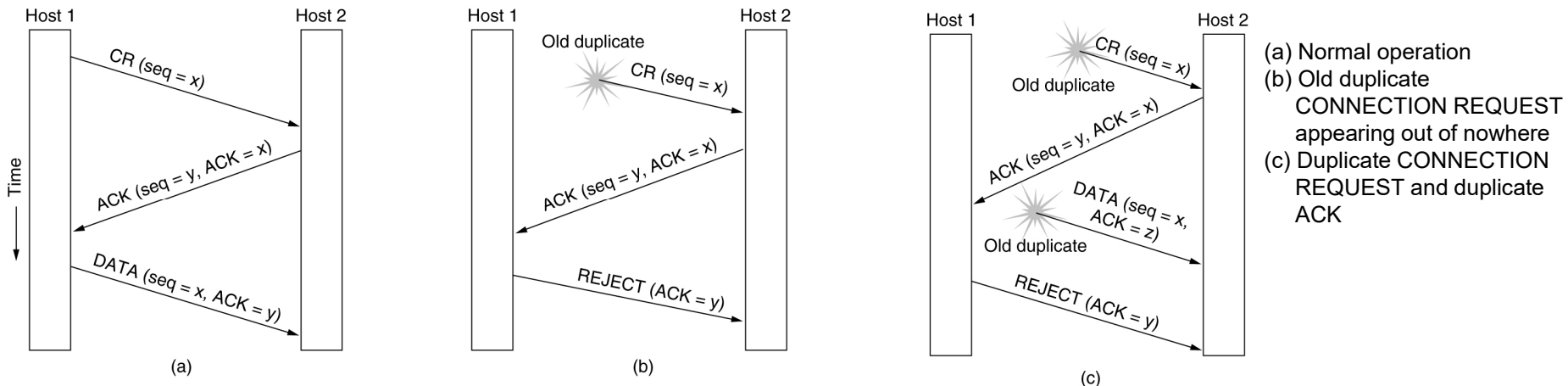
- When a communication link is made over a network (Internet) problems can arise
 - The network can lose, store and duplicate packets
- Use case scenario:
 - User establishes a connection with a bank, instructs the bank to transfer large amount of money, closes connection
 - The network delivers a delayed set of packets, in same sequence, getting the bank to perform another transfer
- For establishing a connection, we
 - Deal with **duplicate**d packets
 - Use **handshake** to establish a connection

Duplicated Packets

- Use of throwaway TSAP addresses
 - Not good, because the client-server paradigm will not work anymore
- Give each connection an **unique identifier** (chosen by the initiating party and attached in each TPDU)
 - After disconnection, each transport entity to update a used connections table (source transport entity, connection identifier) pair
 - Requires each transport entity to maintain history information
- Packet **lifetime** has to be restricted to a known one
 - Restricted subnet design – Any method that prevents packets from looping
 - Hop counter in each packet – Having a hop counter incremented every time the packet is forwarded
 - Time-stamping each packet – Each packet carries the time it was created, with routers agreeing to discard any packets older than a given time. Routers have to have sync clocks (not an easy task)
 - In practice, if the packet has to be dead, all the sequent acknowledges to it are also dead
- **Tomlinson** (1975) proposed that each host will have a binary counter that increases itself at uniform intervals
 - The no. of bits in the counter has to exceed the no. of bits used in the sequence number (**seq_no**)
 - The counter will run even if the host goes down
 - The **idea** is to ensure that two identically numbered TPDU's are never outstanding at the same time
 - Each connection starts numbering its TPDU's with a difference **seq_no**; the sequence space should be so large that by the time **seq_no** will wrap around, old TPDU's with the same **seq_no** are long gone

Three-Way Handshake

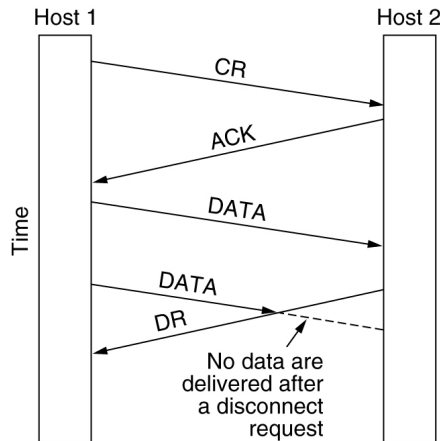
- Control TPDUs may also be delayed. Consider the situation below:
 - Host 1 sends CONNECTION REQ (initial seq_no, destination port no.) to a remote peer host 2
 - Host 2 acknowledges the request by sending CONNECTION ACCEPTED TPDDU back
 - If first request is lost, but a delayed duplicate CONNECTION REQ suddenly shows up at host 2, the connection will be established incorrectly; solution is **3-way handshake**
- 3-way handshake:
 - Each packet is responded to in sequence
 - Duplicates must be rejected
- 3 protocol scenarios for establishing a connection using 3-way handshake:



Note: CR and ACK denote CONNECTION REQUEST and CONNECTION ACCEPTED, respectively

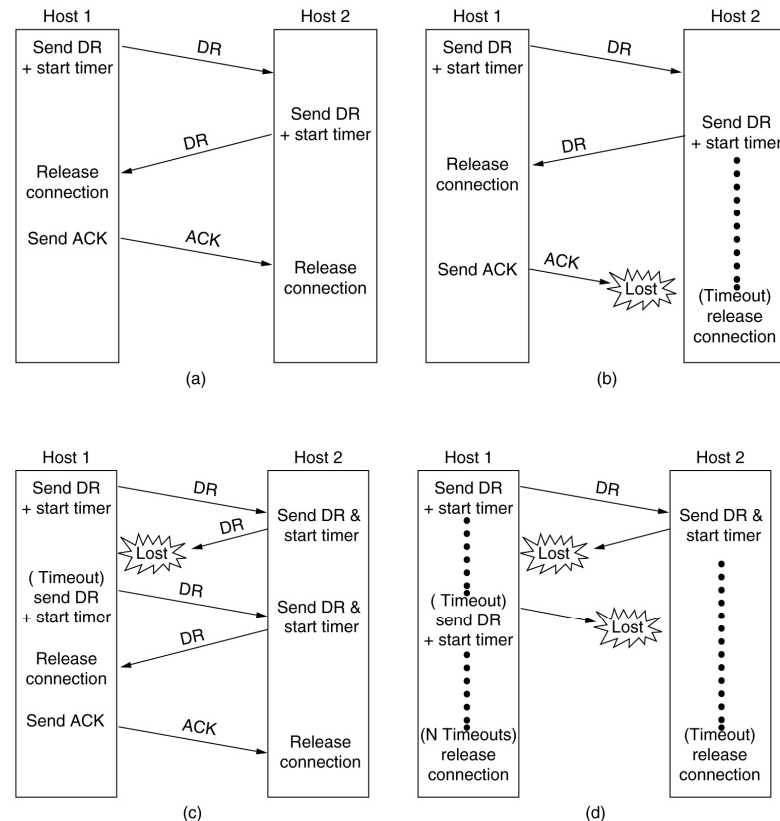
Connection Release

Asymmetrical Connection Release



Is abrupt and may result in loss of data?

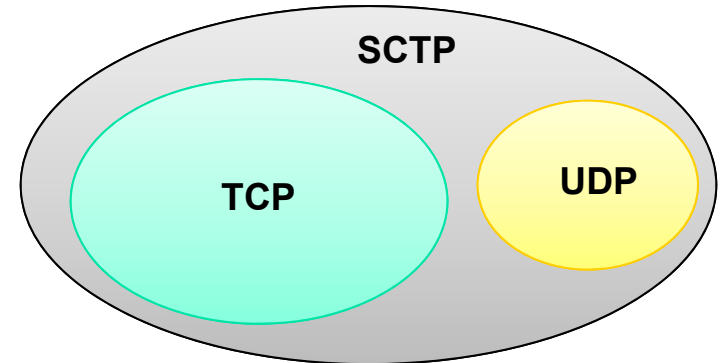
Symmetrical Connection Release



- (a) Normal case of a three-way handshake
- (b) Final ACK lost – situation saved by the timer
- (c) Response lost
- (d) Response lost and subsequent DRs lost

TCP/IP Transport Layer

- Two end-to-end protocols
 - ▣ **UDP** – User Datagram Protocol
 - ▣ **TCP** – Transmission Control Protocol
- **UDP**
 - ▣ Connectionless (unreliable)
 - ▣ Flow control etc provided by application
 - ▣ Client server application
- **TCP**
 - ▣ Connection-oriented (either real or virtual)
 - ▣ Fragments a message for sending
 - ▣ Combines the message on receipt

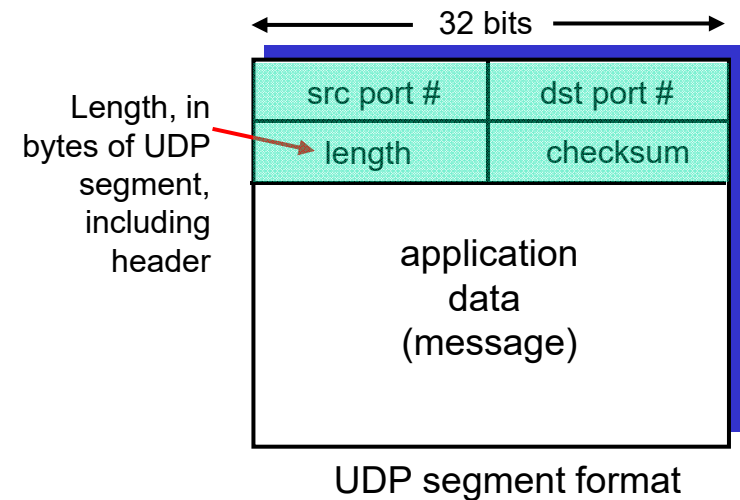


User Datagram Protocol (UDP)

- Best effort service, UDP segments may be:
 - Lost
 - Delivered out of order to application
- **Connectionless:**
 - No handshaking between UDP sender, receiver
 - Each UDP segment handled independently of others
- Often used for streaming multimedia applications
 - Loss tolerant
 - Rate sensitive
- Reliable transfer over UDP: add reliability at application layer
 - Application-specific error recovery
- Other UDP uses DNS and SNMP

Why is there a UDP?

- simple: no connection state at sender, receiver
- no connection establishment (which can add delay)
- small segment header
- no congestion control: UDP can blast away as fast as desired



Note: You can view UDP traffic to and from your PC with the following command:

```
> netstat -snp udp
```

UDP Checksum

- Goal: detect “errors” (e.g., flipped bits) in transmitted segment

Sender:

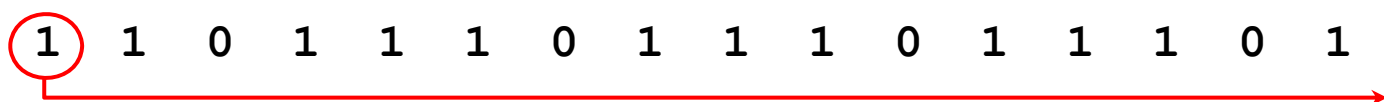
- ▣ Treat segment contents as sequence of 16-bit integers
- ▣ **Checksum**: addition (1’s complement sum) of segment contents
- ▣ Sender puts checksum value into UDP checksum field

Receiver:

- ▣ Compute checksum of received segment
- ▣ Check if computed checksum equals checksum field value:
 - NO – error detected
 - YES – no error detected, *but maybe errors nonetheless?*

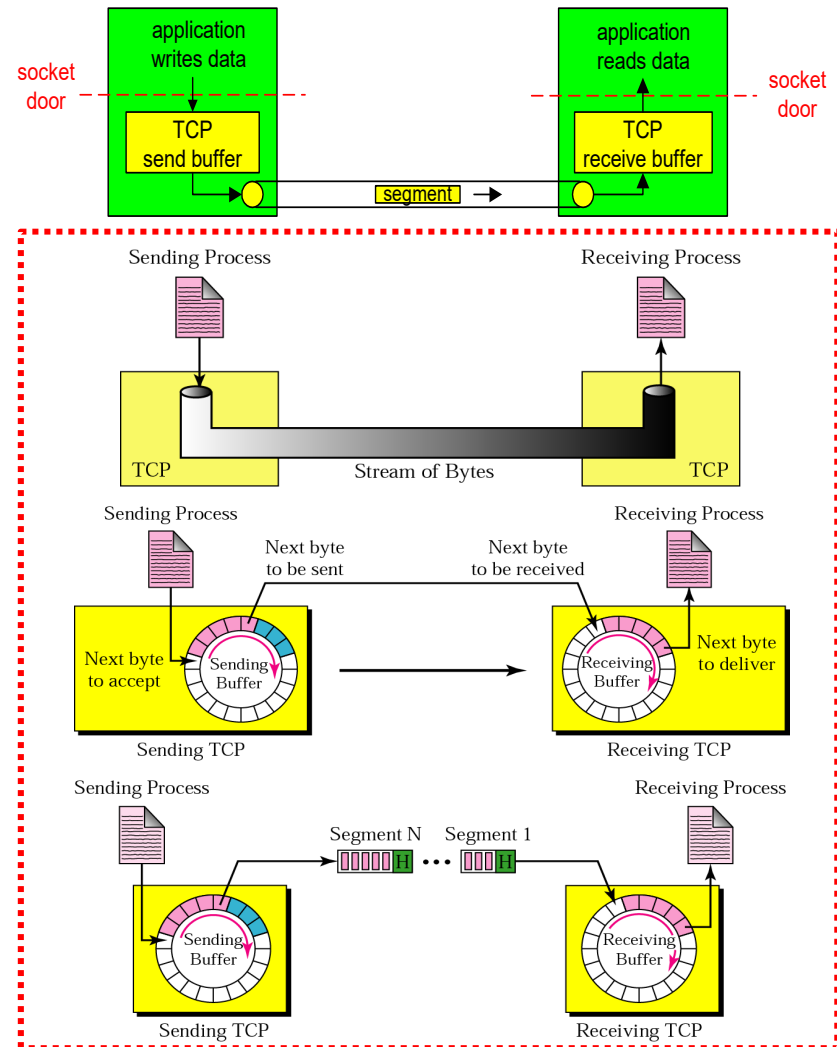
Example

- Note:
 - When adding numbers, a carryout from the **most significant bit** needs to be added to the result
- Example: add two 16-bit integers

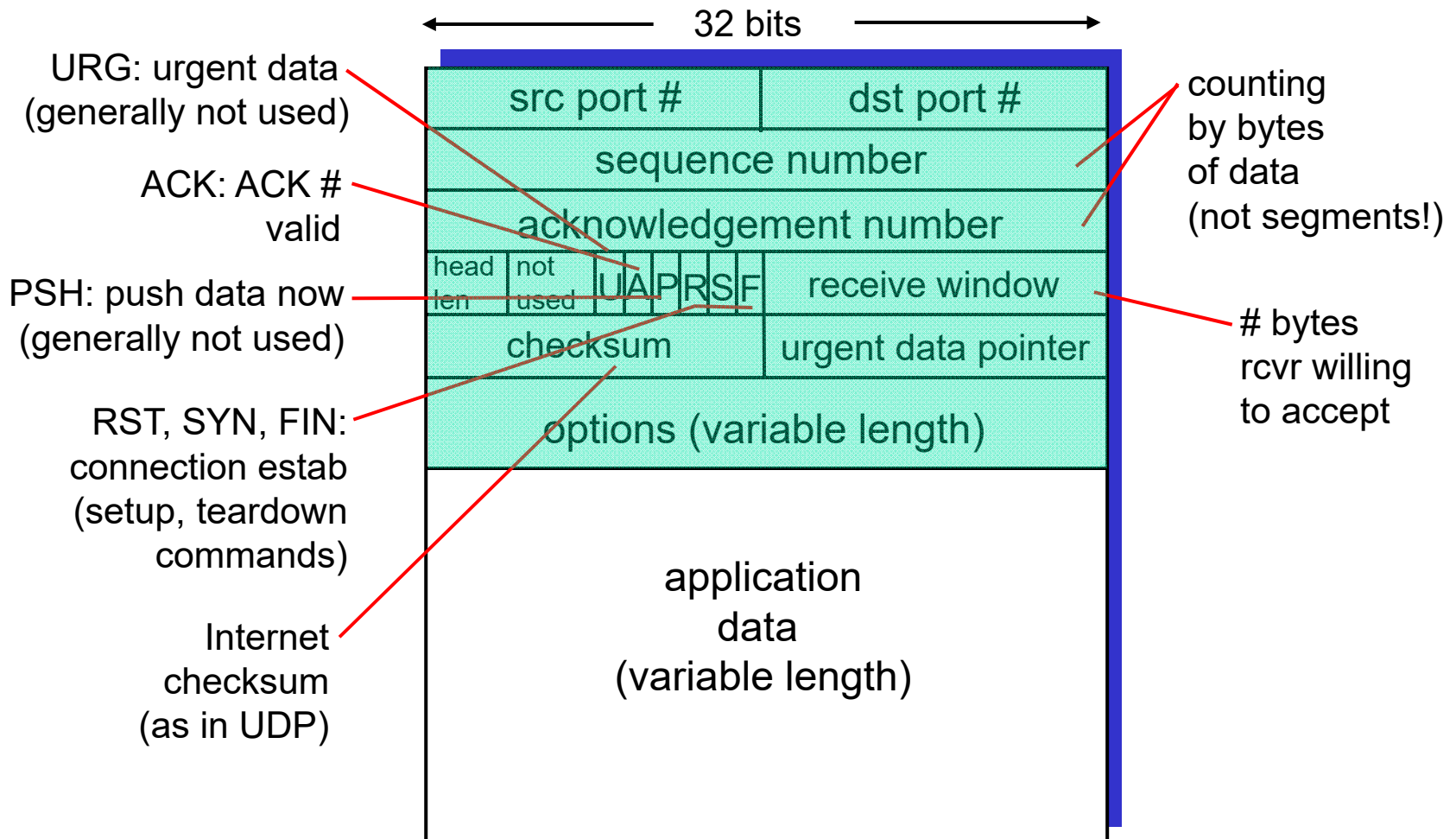
		1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
		1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
<hr/>																	
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
																	
sum		1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum		0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

Transport Control Protocol (TCP)

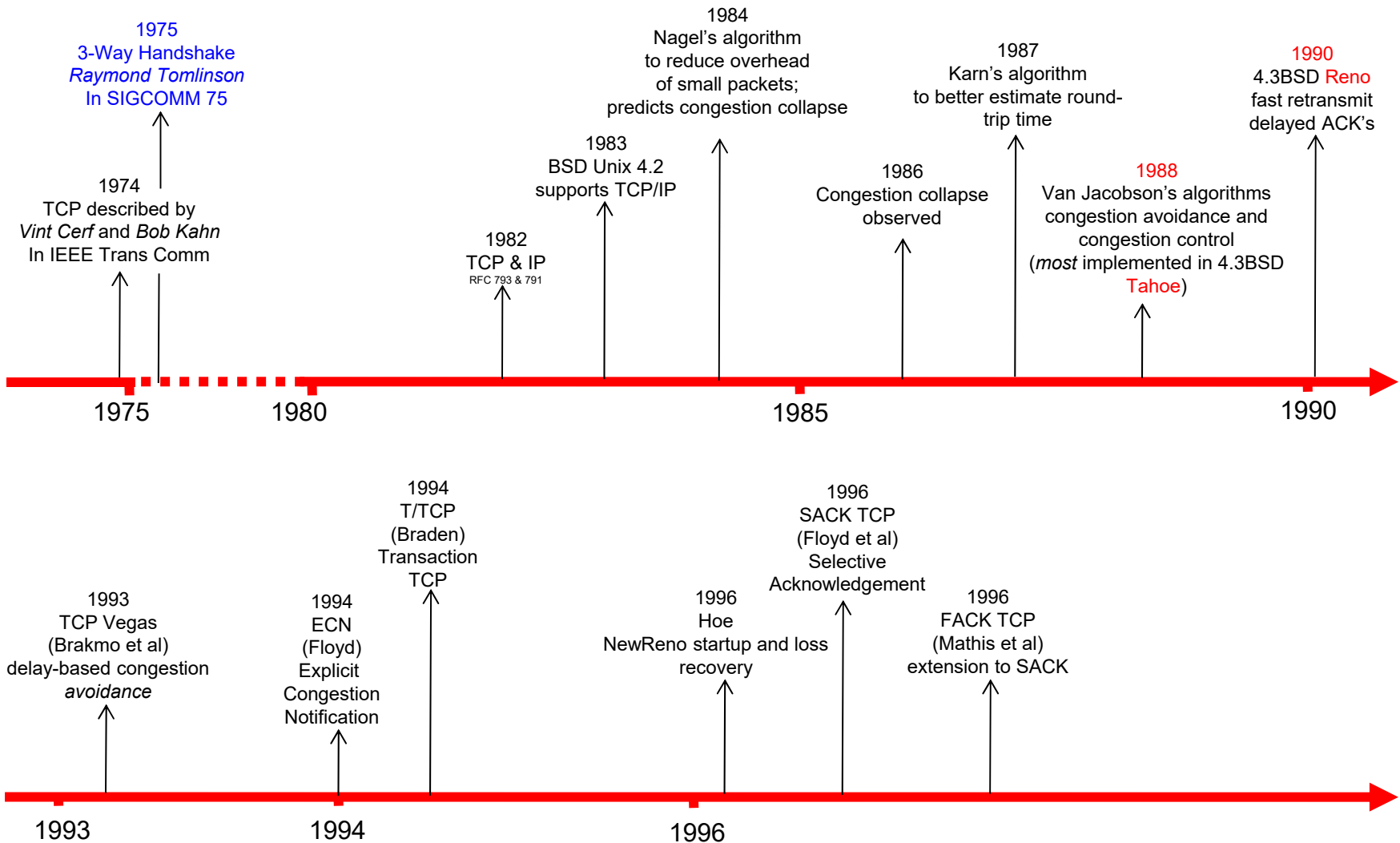
- Point-to-point – 1 sender, 1 receiver
- Reliable, in-order byte stream:
 - No “message boundaries”
- Pipelined – TCP congestion and flow control set window size
- Send & receive buffers
- Full duplex data:
 - Bi-directional data flow in same connection
 - MSS: maximum segment size
- **Connection-oriented:**
 - Handshaking (exchange of control messages) initial sender and receiver state before data exchange
- Flow controlled:
 - Sender will not overwhelm receiver



TCP Segment Format



Evolution of TCP



Sequence Number & ACK

- Sequence number
 - Byte stream
 - “number” of first byte in segment’s data

- ACK

- Sequence number of next byte expected from other side
 - cumulative ACK

- How receiver handles out-of-order segments?

- TCP specification does NOT say – up to implementer

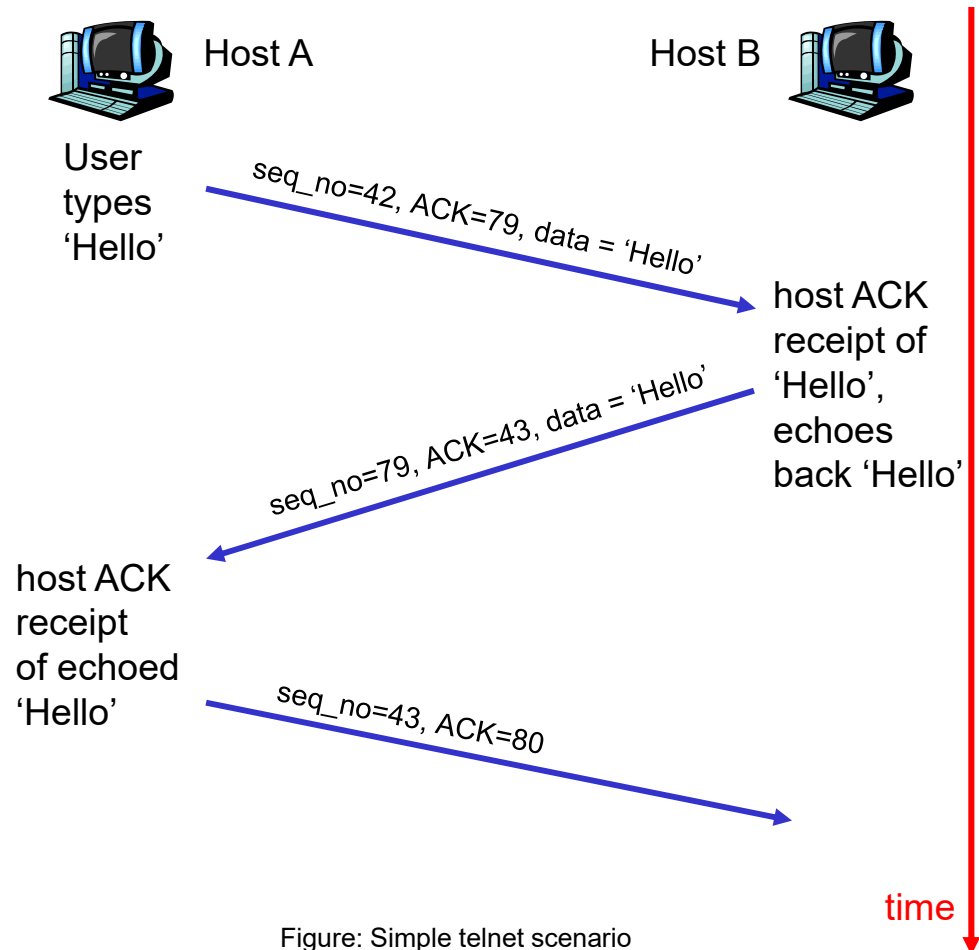


Figure: Simple telnet scenario

Timeout & Round Trip Time (RTT)

How to set TCP timeout value?

- Longer than RTT
 - ▣ But RTT varies
- Too short: premature timeout
 - ▣ Unnecessary retransmissions
- Too long: slow reaction to segment loss

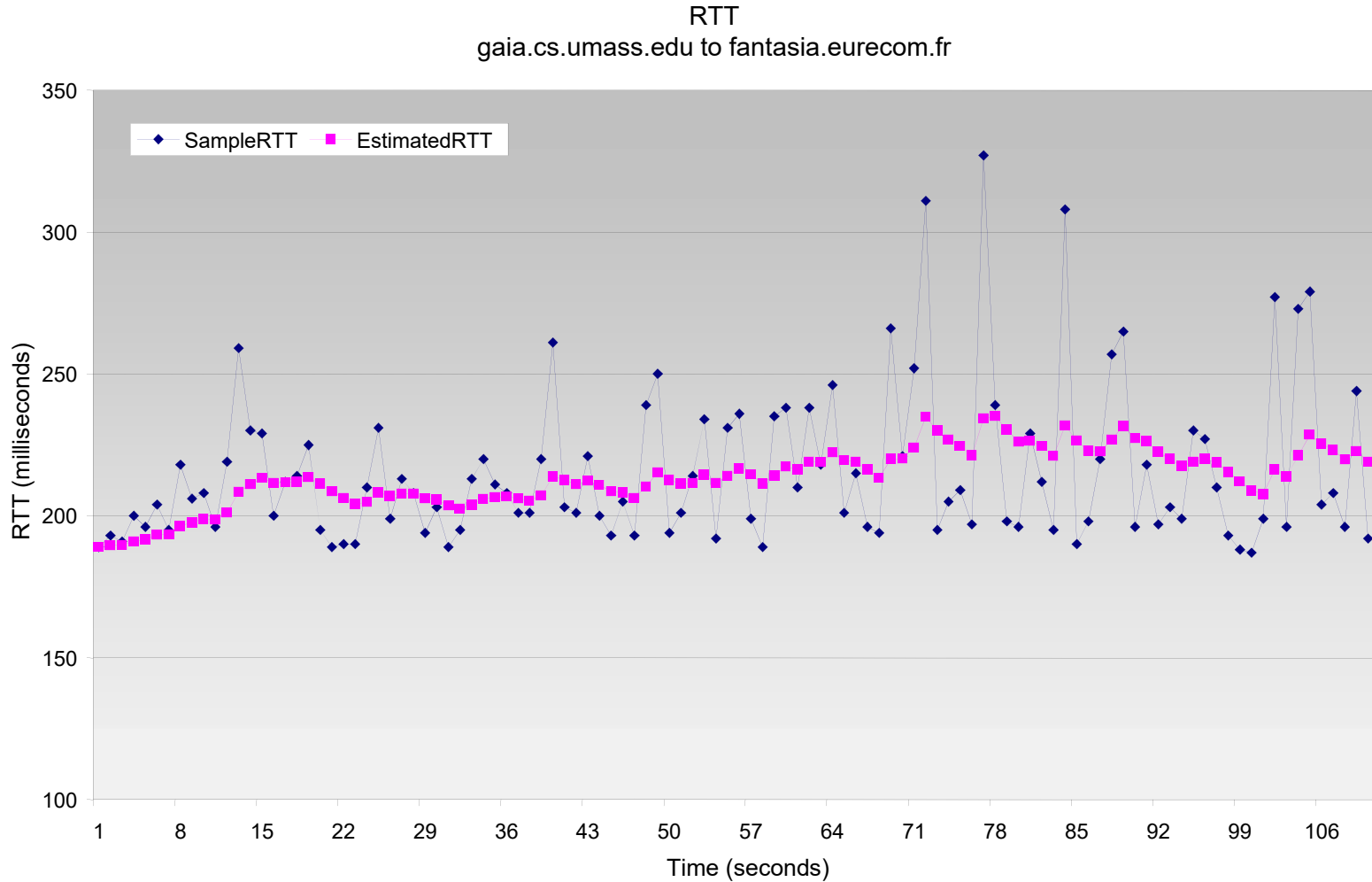
How to estimate RTT?

- SampleRTT: measured time from segment transmission until ACK receipt
 - ▣ Ignore retransmissions
- SampleRTT will vary, want estimated RTT “smoother”
 - ▣ Average several recent measurements, not just current SampleRTT

$$\text{EstimatedRTT} = (1-\alpha) * \text{EstimatedRTT} + \alpha * \text{SampleRTT}$$

- Exponential weighted moving average
- Influence of past sample decreases exponentially fast
- Typical value: $\alpha = 0.125$

Example RTT Estimation



Setting the Timeout

- EstimatedRTT plus “safety margin”
 - ▣ Large variation in EstimatedRTT → larger safety margin
- First estimate of how much SampleRTT **deviates** from EstimatedRTT

$$\text{DevRTT} = (1-\beta) * \text{DevRTT} + \beta * |\text{SampleRTT} - \text{EstimatedRTT}|$$

(note: typical value $\beta = 0.25$)

- Then, set **timeout** interval:

$$\text{TimeoutInterval} = \text{EstimatedRTT} + 4 * \text{DevRTT}$$

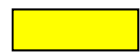
Multiplexing / Demultiplexing

Demultiplexing at receiver side:

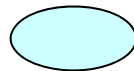
delivering received segments
to correct socket

Multiplexing at sender side:

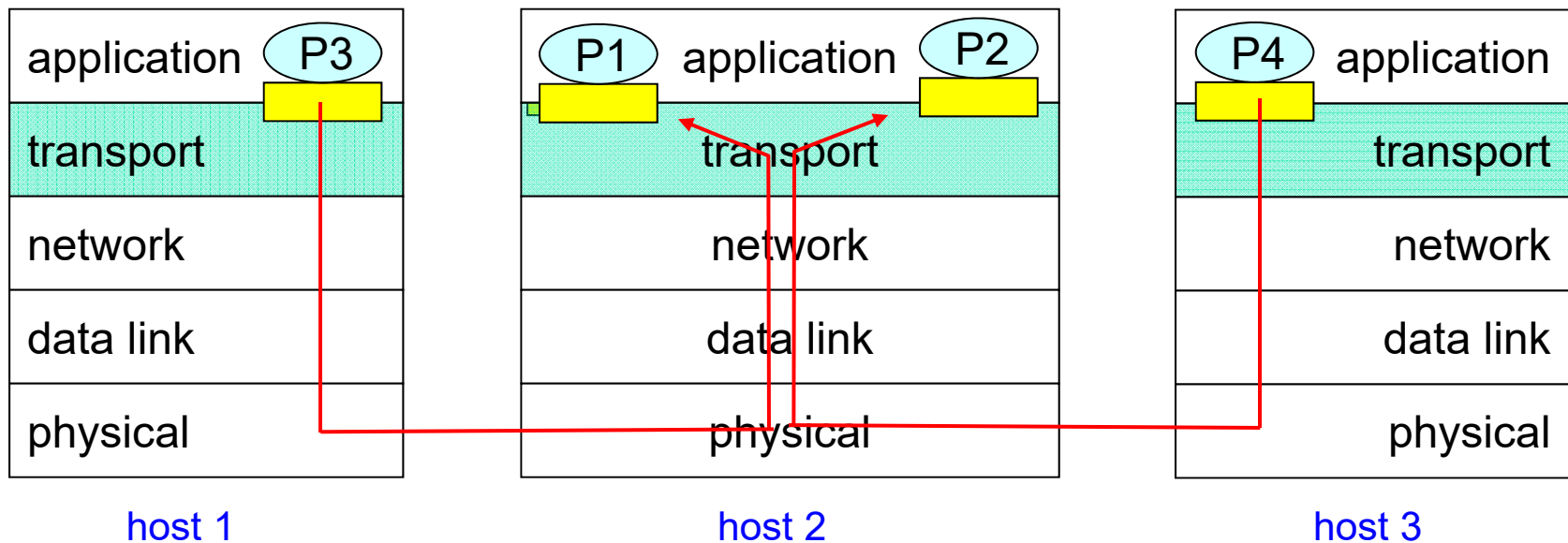
gathering data from multiple
sockets, enveloping data with
header (later used for
demultiplexing)



= socket

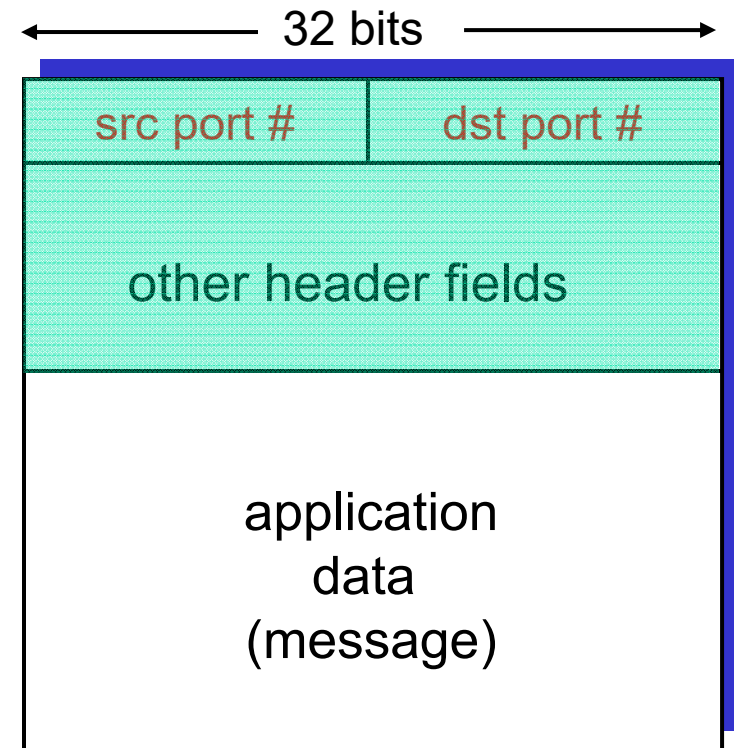


= process



How Demultiplexing Works

- Host receives IP datagrams
 - Each datagram has source IP address, destination IP address
 - Each datagram carries 1 transport-layer segment
 - Each segment has source, destination port number
- Host uses IP addresses & port numbers to direct segment to appropriate socket



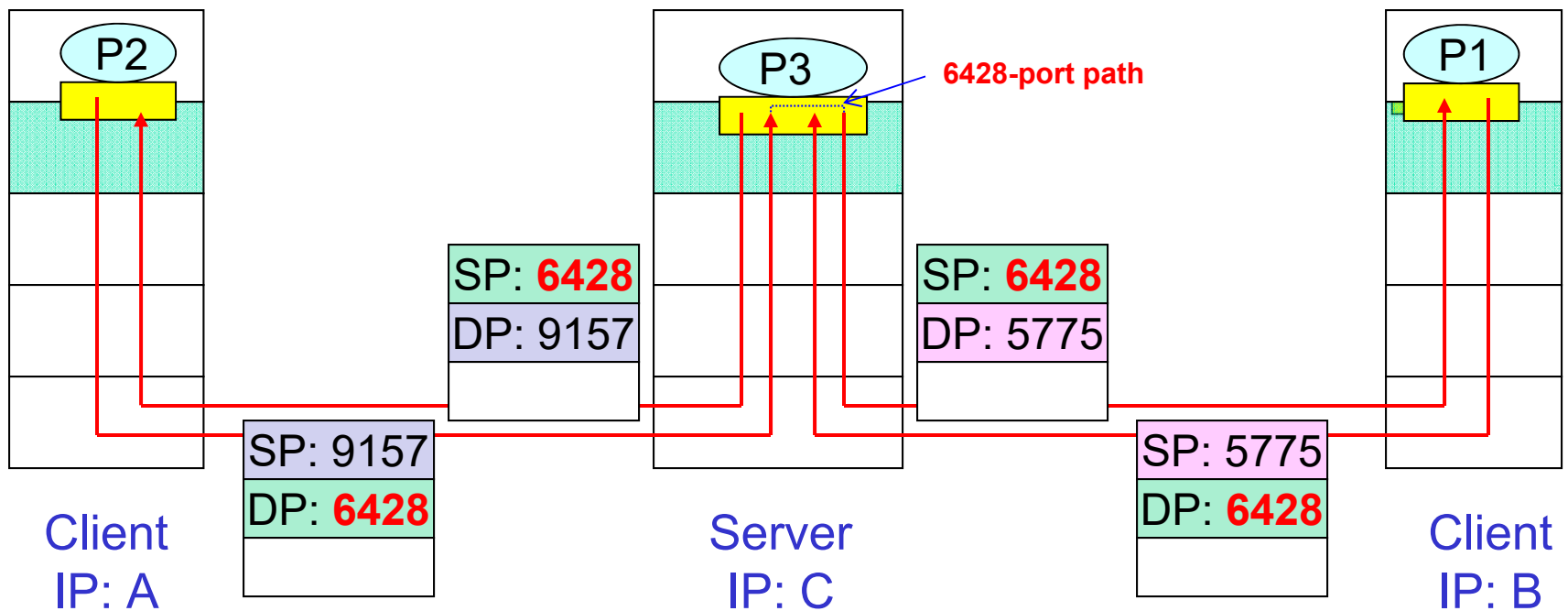
TCP/UDP segment format

Demultiplexing: Connectionless

- Create sockets with port numbers
 - ▣ `DatagramSocket mySocket1 = new DatagramSocket(12534);`
 - ▣ `DatagramSocket mySocket2 = new DatagramSocket(12535);`
- UDP socket identified by **2-tuple**:
 - ▣ Dest IP address
 - ▣ Dest port number
- When host receives UDP segment:
 - ▣ Checks destination port number in segment
 - ▣ Directs UDP segment to socket with that port number
- IP datagrams with different source IP addresses and/or source port numbers directed to same socket

Demultiplexing: Connectionless

```
DatagramSocket serverSocket = new DatagramSocket(6428);
```



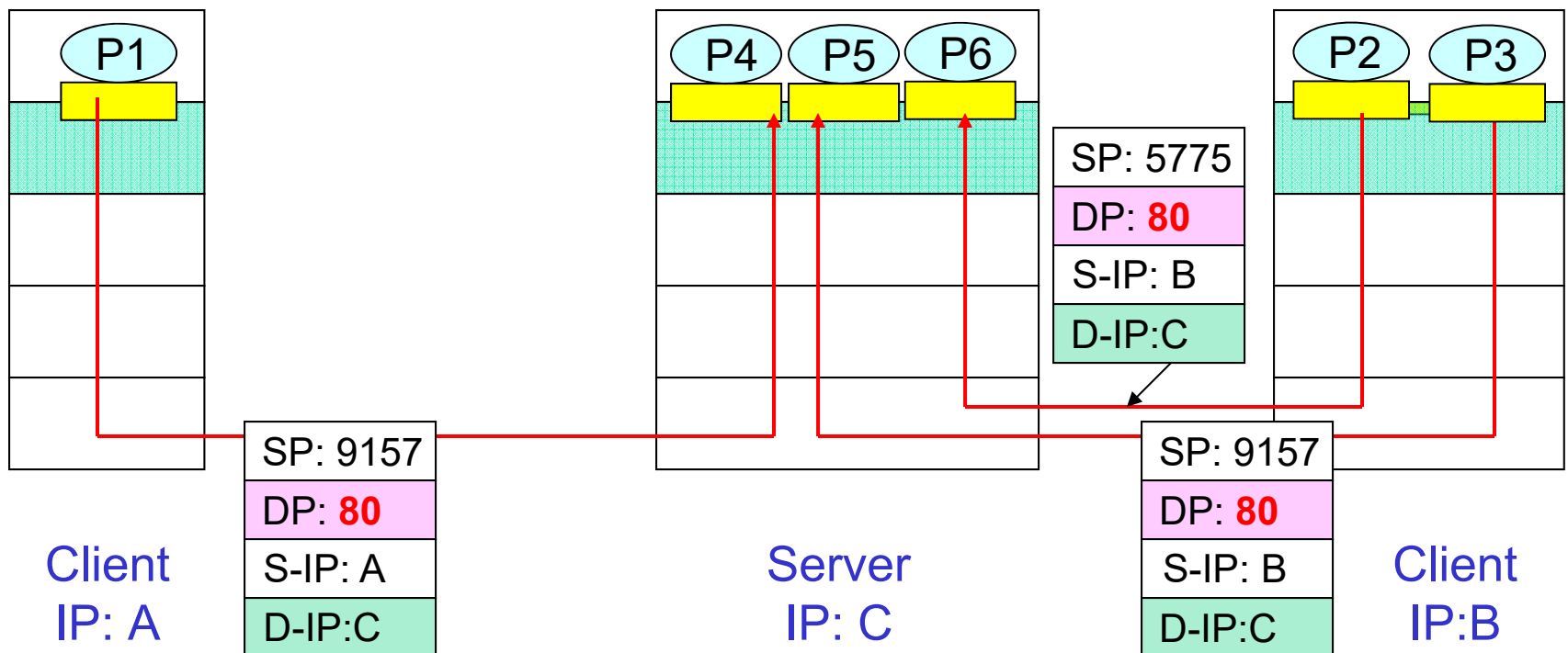
SP provides "return address"

Demultiplexing: Connection-Oriented

- TCP socket identified by **4-tuple**:
 - Source IP address
 - Source port number
 - Dest IP address
 - Dest port number
- Receiver host uses all four values to direct segment to appropriate socket
- Server host may support many simultaneous TCP sockets:
 - Each socket identified by its own 4-tuple
- Web servers have different sockets for each connecting client
 - Non-persistent HTTP will have different socket for each request

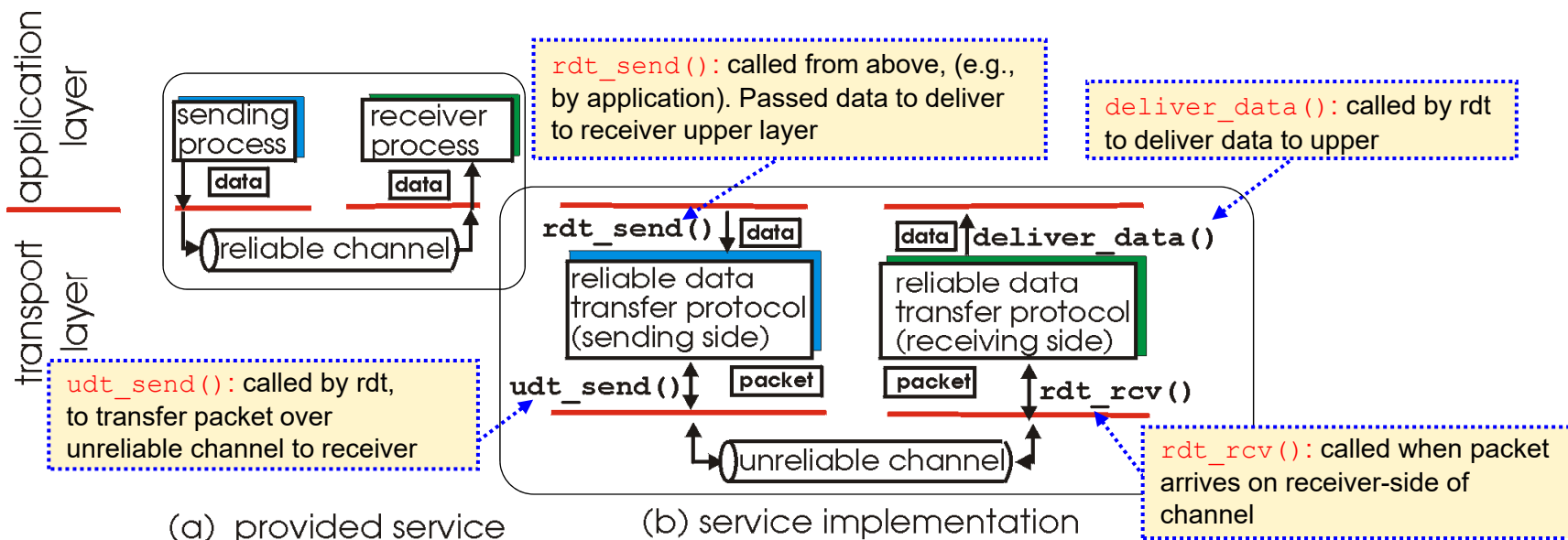
Demultiplexing: Connection-Oriented

```
Socket clientSocket = new Socket("host", 80);
```



Reliable Data Transfer

- Important in the layers of application, transport, data link
- Top-10 list of important networking topics
- Characteristics of unreliable channel will determine complexity of reliable data transfer (**rdt**) protocol



Reliable Data Transfer: TCP

- TCP creates reliable data transfer service on top of IP's unreliable service
- Pipelined segments
- Cumulative ACKs
- TCP uses single retransmission timer
- Retransmissions are triggered by
 - Timeout events
 - Duplicate ACKs
- Initially consider simplified TCP sender
 - Ignore duplicate ACKs
 - Ignore flow control, congestion control

Sender Events

At the sender side:

- When data is received from application layer
 - Create segment with seq_no
 - seq_no is byte-stream number of first data byte in segment
 - Start timer if not already running (think of timer as for oldest unACKed segment)
 - Expiration interval: `TimeoutInterval`
- When timeout is over
 - Retransmit segment that caused timeout
 - Restart timer
- When ACK is received
 - If acknowledges previously unACKed segments
 - Update what is known to be ACKed
 - Start timer if there are outstanding segments

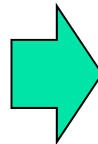
Receiver Events

At the receiver side (ACK generation):

Events at Receiver

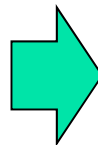
Actions at Receiver

Arrival of in-order segment with expected seq_no. All data up to expected seq_no already ACKed



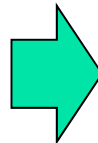
Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK

Arrival of in-order segment with expected seq_no. One other segment has ACK pending



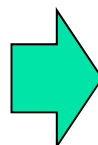
Immediately send single cumulative ACK, ACKing both in-order segments

Arrival of out-of-order segment higher-than-expected seq_no. gap detected



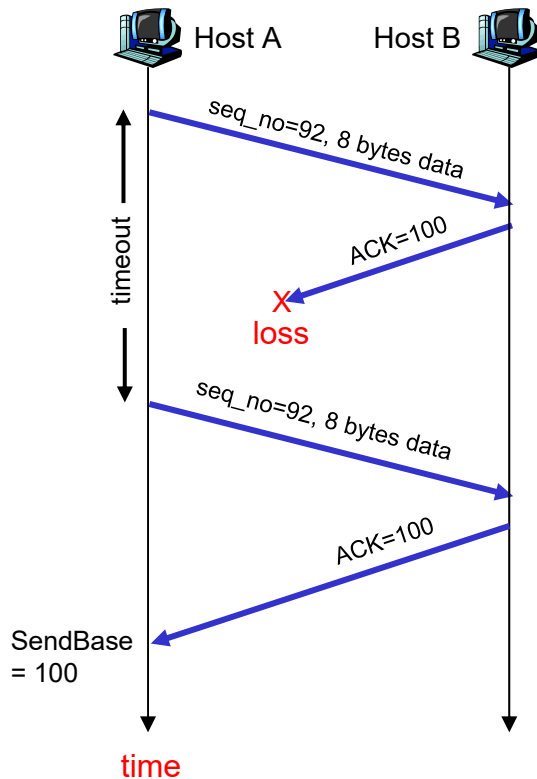
Immediately send *duplicate ACK*, indicating seq_no of next expected byte

Arrival of segment that partially or completely fills gap

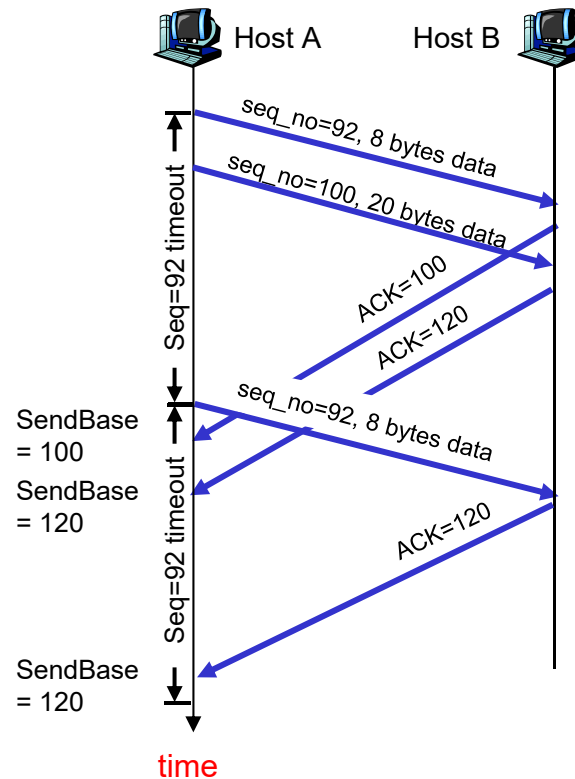


Immediate send ACK, provided that segment starts at lower end of gap

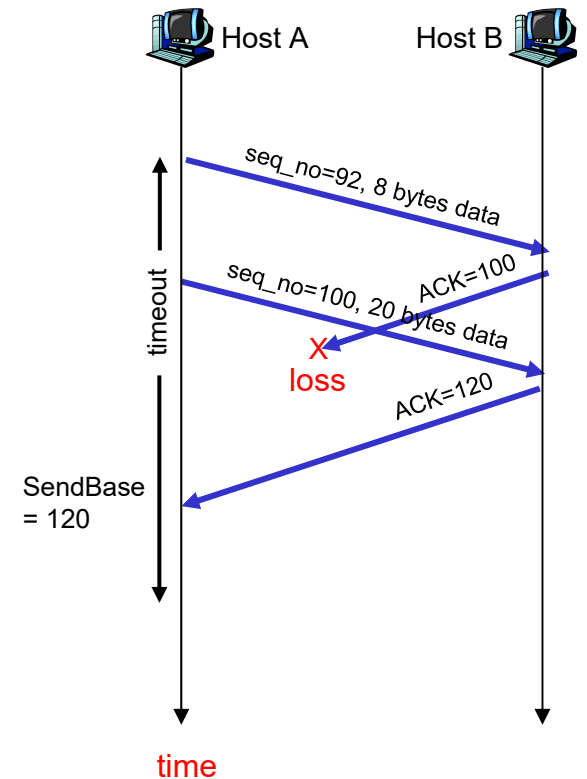
Retransmission Scenarios



Lost ACK scenario

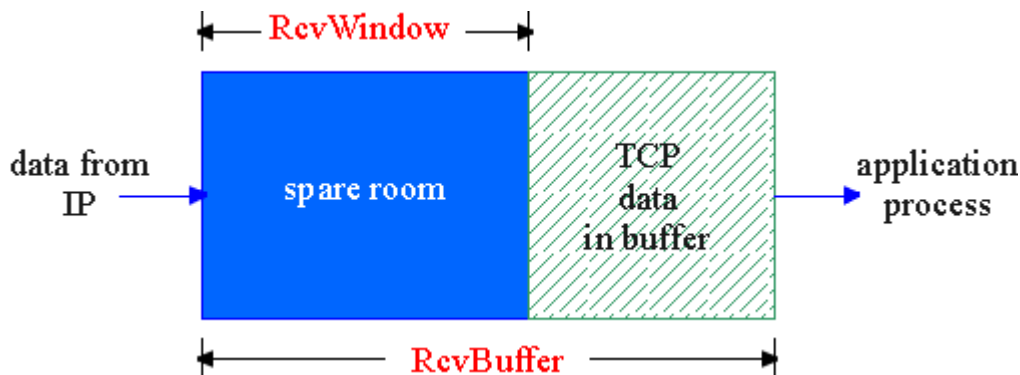


Premature Timeout



Cumulative ACK Scenario

Flow Control



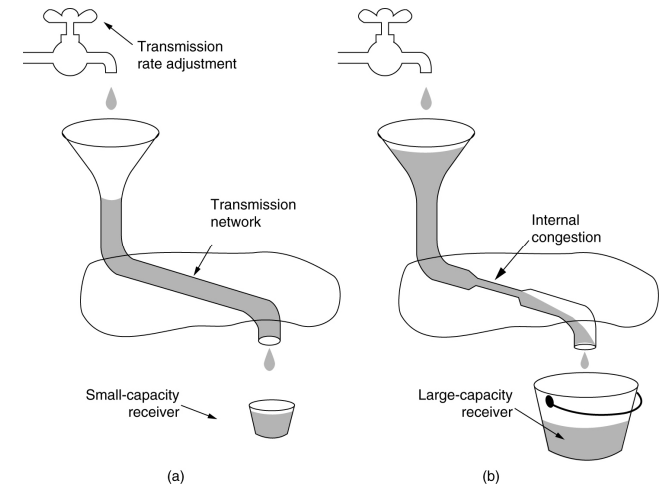
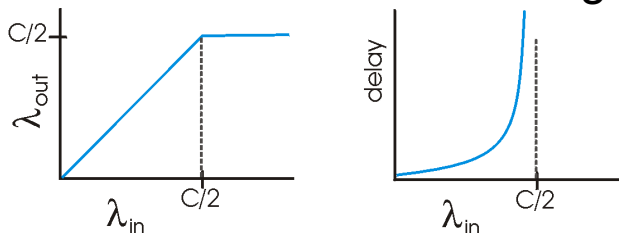
Flow Control

Sender won't overflow receiver's buffer by transmitting too much, too fast

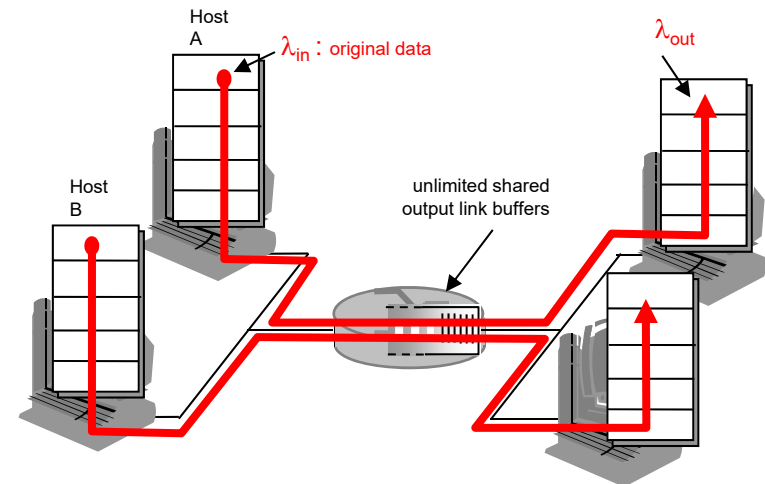
- Receiver side of TCP connection has a receive buffer
- Application process may be slow at reading from buffer
- **Speed-matching service**: matching the send rate to the receiving application's drain rate
- How it works?
 - Suppose TCP receiver discards out-of-order segments
 - Spare room in buffer = **RcvWindow** (is equal to $\text{RcvBuffer} - [\text{LastByteRcvd} - \text{LastByteRead}]$)
 - Receiver advertises spare room by including value of **RcvWindow** in segments
 - Sender limits unACKed data to **RcvWindow**

Congestion Control

- Informally: “too many sources sending too much data too fast for network to handle”
- It is different from flow control
- Manifestations
 - Lost packets (buffer overflow at routers)
 - Long delays (queuing in router buffers)
- Causes/costs of congestion
 - Two senders, two receivers
 - One router, infinite buffers
 - No retransmission
 - Large delays when congested
 - Maximum achievable throughput



(a) A fast network feeding a low capacity receiver.
(b) A slow network feeding a high-capacity receiver.



Congestion Control in TCP Tahoe

- TCP Tahoe refers to the TCP with the original set of congestion control algorithms that were devised in 1988
- Since then, TCP has been improved with more advanced congestion control techniques. Such advanced TCP protocols are TCP Reno, TCP NewReno, TCP Vegas, TCP SACK, TCP Jersey, TCP Peach, TCP Peach+, etc
- Consider TCP Tahoe only
 - Essential strategy: The TCP host sends packets into the network without a reservation and then the host reacts to observable events
 - Originally TCP assumed FIFO queuing
 - Basic **idea**: each source determines how much capacity is available to a given flow in the network
 - ACKs are used to 'pace' the transmission of packets such that TCP is "self-clocking"

TCP Tahoe

- CongestionWindow (**cwnd**) is a variable held by the TCP source for each connection

```
MaxWindow = min(cwnd, AdvertisedWindow)
EffectiveWindow = MaxWindow - (LastByteSent - LastByteAcked)
```

- cwnd is set based on the perceived level of congestion
The Host receives implicit (packet drop) or explicit (mark) indications of internal congestion

- For every segment that is acknowledged

```
{
  if (cwnd < ssthresh) /*if we're still doing slow-start */
    cwnd = 2*cwnd;      /*open window exponentially */
  Else                /*otherwise do congestion avoidance*/
    cwnd += 1;         /*increment-by-1 */
}
```

- On packet loss

```
{
  ssthresh = cwnd/2; /*new threshold is half of current cwnd*/
  cwnd = 1;          /*go back to slow-start phase */
}
```

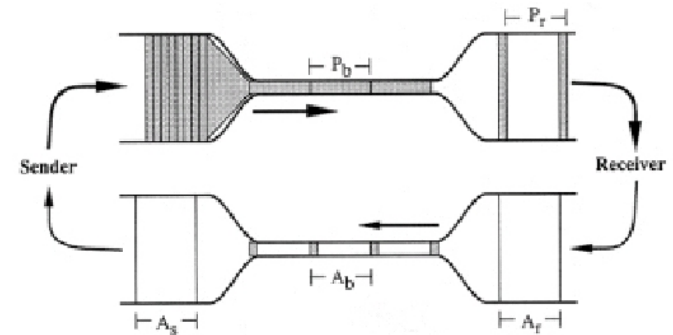
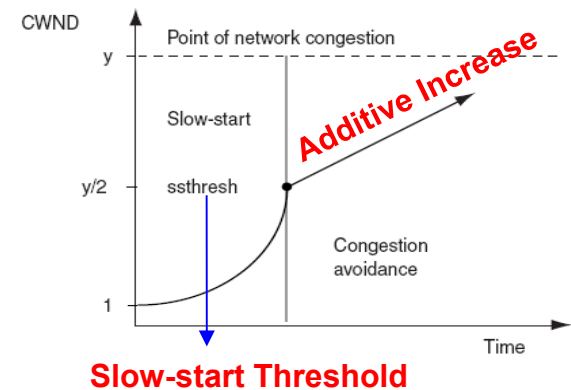


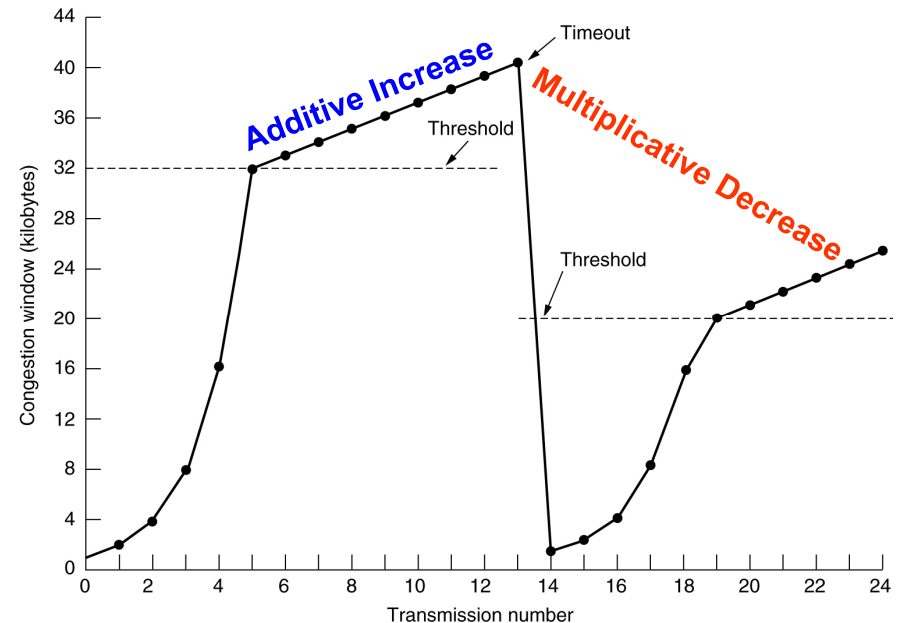
Figure: Self-clocking feature of TCP



TCP Tahoe Performance

TCP Tahoe:

- Start with $cwnd = 1$ (slow start)
- On each successful ACK increment $cwnd$
 $cwnd \leftarrow cwnd + 1$
- Exponential growth of $cwnd$
each RTT: $cwnd \leftarrow 2 \times cwnd$



- Enter CA when $cwnd \geq ssthresh$
- Factors that affect throughput, T :
 - Segment size at a time (after all MTU), B
 - round trip time (RTT), R
 - Packet loss rate, p

$$T \approx \frac{1.5 \sqrt{\frac{2}{3}} B}{R \sqrt{p}}$$

Announcement

- Next is Chapter 7 Application Layer and Network Programming Basics
- 10:50 ~ 12:30 on 02 November (Wednesday)