Telematics I

Chapter 8
Transport Layer

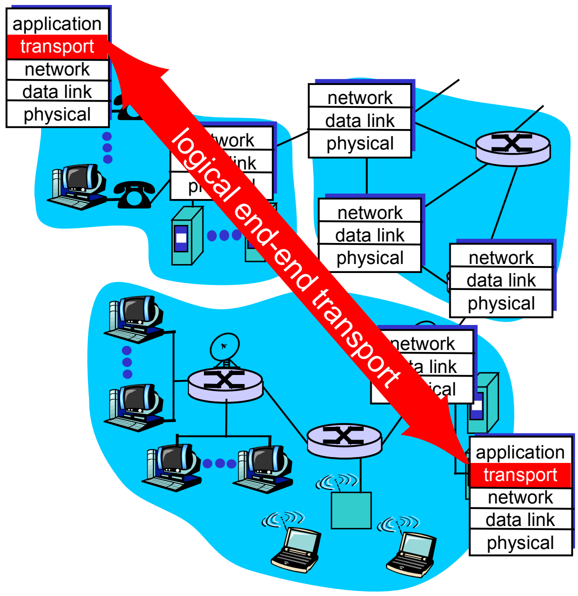
(Acknowledgement: these slides have mostly been compiled from [KR04, Kar04, Sch04])

Chapter Overview

- **Transport Layer Services and Protocols**
- Addressing and Multiplexing
- Connection Control
- Flow Control
- Congestion Control
- Transport protocols in the Internet:
  - User Datagram Protocol (UDP)
  - Transport Control Protocol (TCP)
    - Connection Management
    - Reliable Data Transfer
    - Flow Control
    - Congestion Control
    - Performance

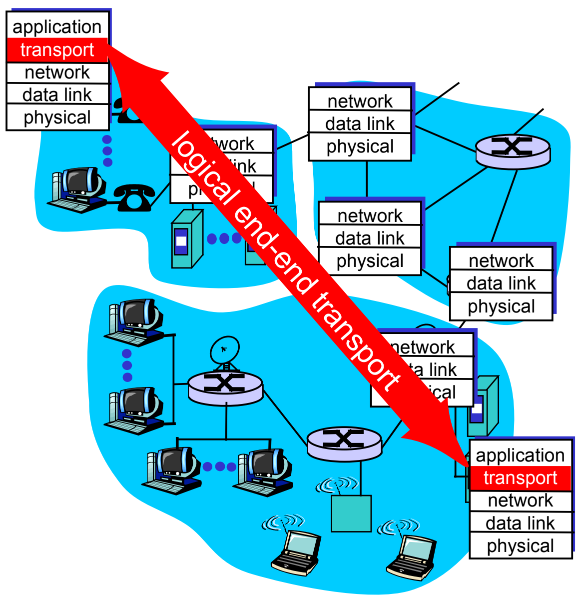
Transport Services and Protocols

- Provide *logical communication* between app processes running on different hosts
- Transport protocols run in end systems
  - Sending side: breaks app messages into segments, passes to network layer
  - Receiving side: reassembles segments into messages, passes to app layer
- More than one transport protocol available to applications
  - Internet: TCP and UDP



Internet Transport-Layer Protocols

- Reliable, in-order delivery (TCP)
  - Congestion control
  - Flow control
  - Connection setup
- Unreliable, unordered delivery: UDP
  - No-frills extension of “best-effort” IP
- Services not available:
  - Delay guarantees
  - Bandwidth guarantees

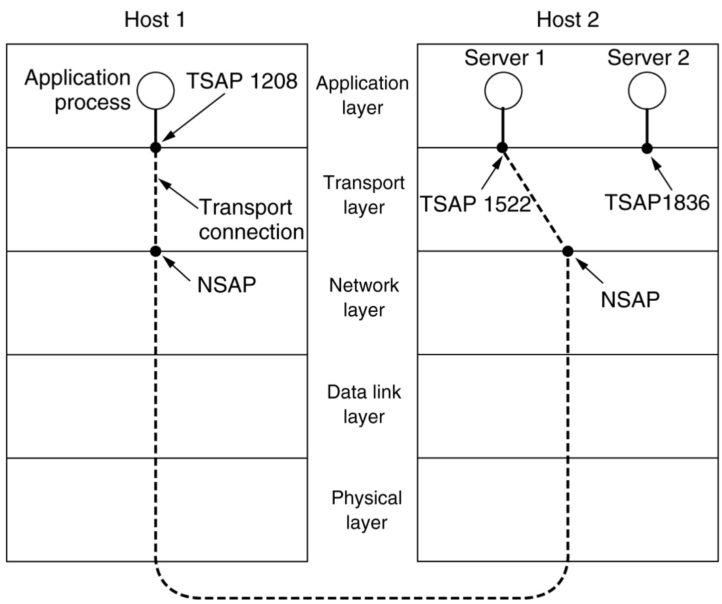


Chapter Overview

- Transport Layer Services and Protocols
- **Addressing and Multiplexing**
- Connection Control
- Flow Control
- Congestion Control
- Transport protocols in the Internet:
  - User Datagram Protocol (UDP)
  - Transport Control Protocol (TCP)
    - Connection Management
    - Reliable Data Transfer
    - Flow Control
    - Congestion Control
    - Performance

Addressing and Multiplexing

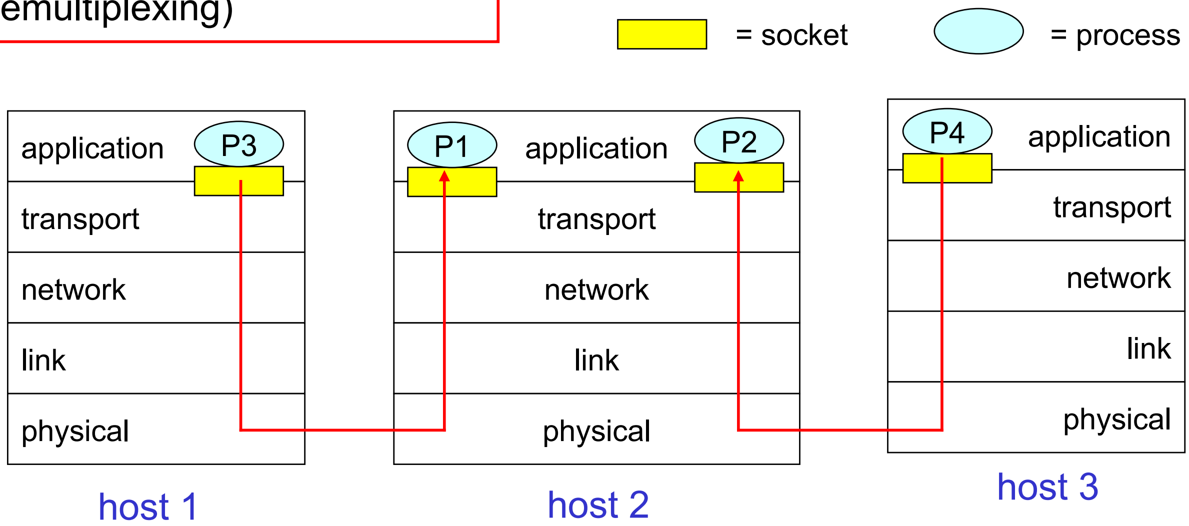
- Provide multiple *service access points (SAP)* to multiplex several applications
  - SAPs can identify connections or data flows
- E.g., *“port numbers”*
  - Dynamically allocated
  - Predefined for “well-known services” – port 80 for Web server



Multiplexing/Demultiplexing

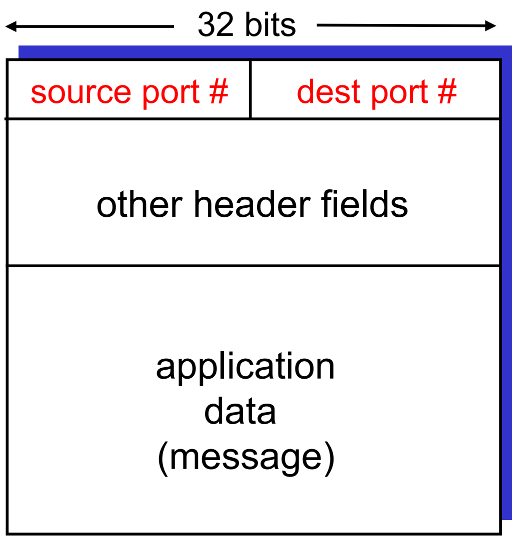
Multiplexing at send host:
Gathering data from multiple sockets, enveloping data with header (later used for demultiplexing)

Demultiplexing at rcv host:
Delivering received segments to correct socket



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 (recall: well-known port numbers for specific applications)
- Host uses IP addresses & port numbers to direct segment to appropriate socket



TCP/UDP segment format

Connectionless Demultiplexing

- Create sockets with port numbers:

```
DatagramSocket mySocket1 =
new DatagramSocket(9111);
DatagramSocket mySocket2 =
new DatagramSocket(9222);
```

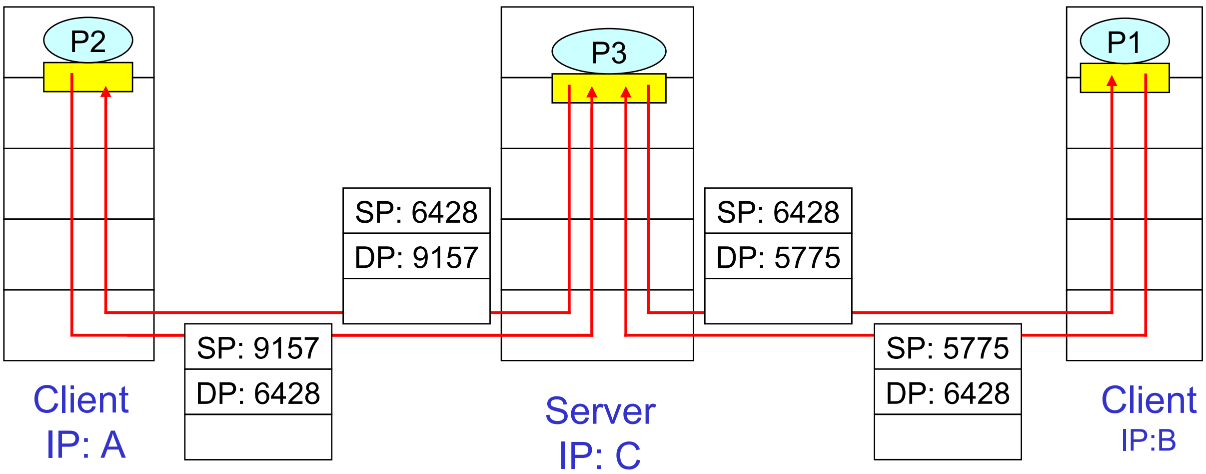
- UDP socket identified by two-tuple:

  (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

Connectionless Demultiplexing (continued)

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

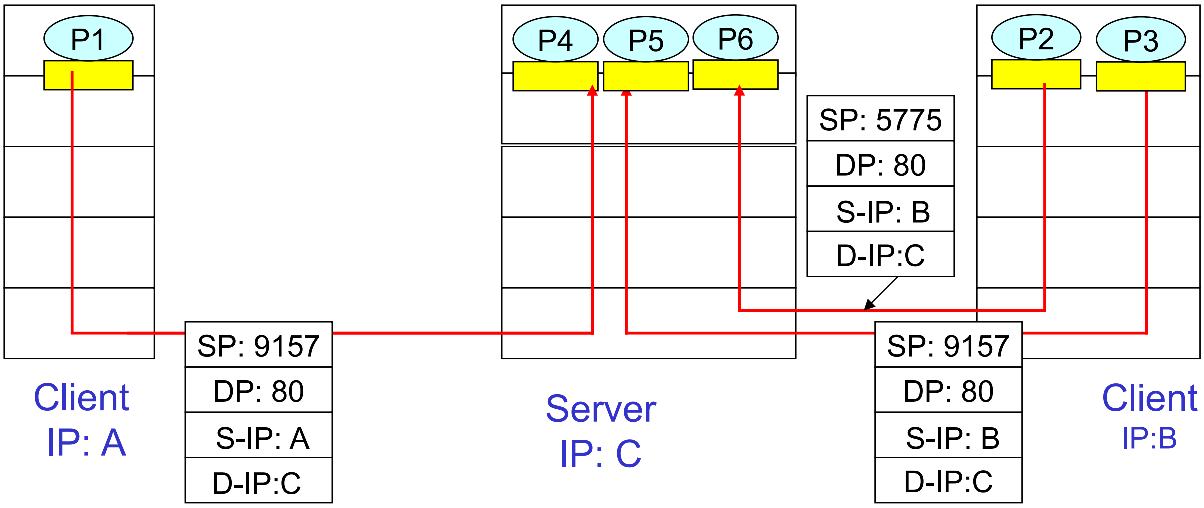


Source port (SP) provides “return address”

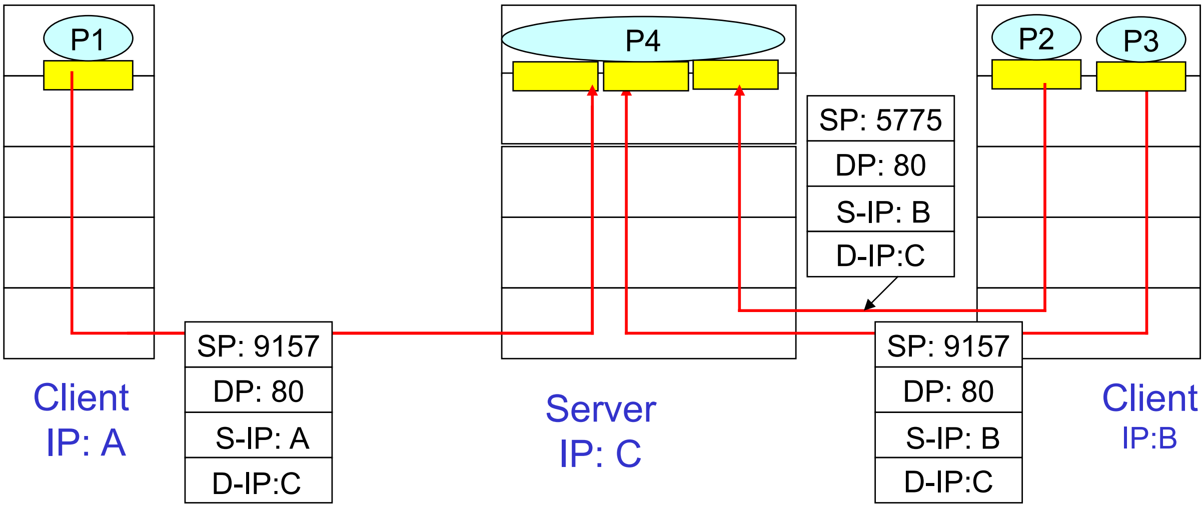
Connection-Oriented Demultiplexing

- TCP socket identified by 4-tuple:
  - Source IP address
  - Source port number
  - Dest IP address
  - Dest port number
- Receiving 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

Connection-Oriented Demultiplexing (continued)



Connection-Oriented Demux: Threaded Web Server



One process can have multiple simultaneous connections

Chapter Overview

- Transport Layer Services and Protocols
- Addressing and Multiplexing
- **Connection Control**
- Flow Control
- Congestion Control
- Transport protocols in the Internet:
  - User Datagram Protocol (UDP)
  - Transport Control Protocol (TCP)
    - Connection Management
    - Reliable Data Transfer
    - Flow Control
    - Congestion Control
    - Performance

Connection Control

- Recall the two types of communication services to be distinguished:
  - **Connection-oriented service**
  - **Connectionless service**

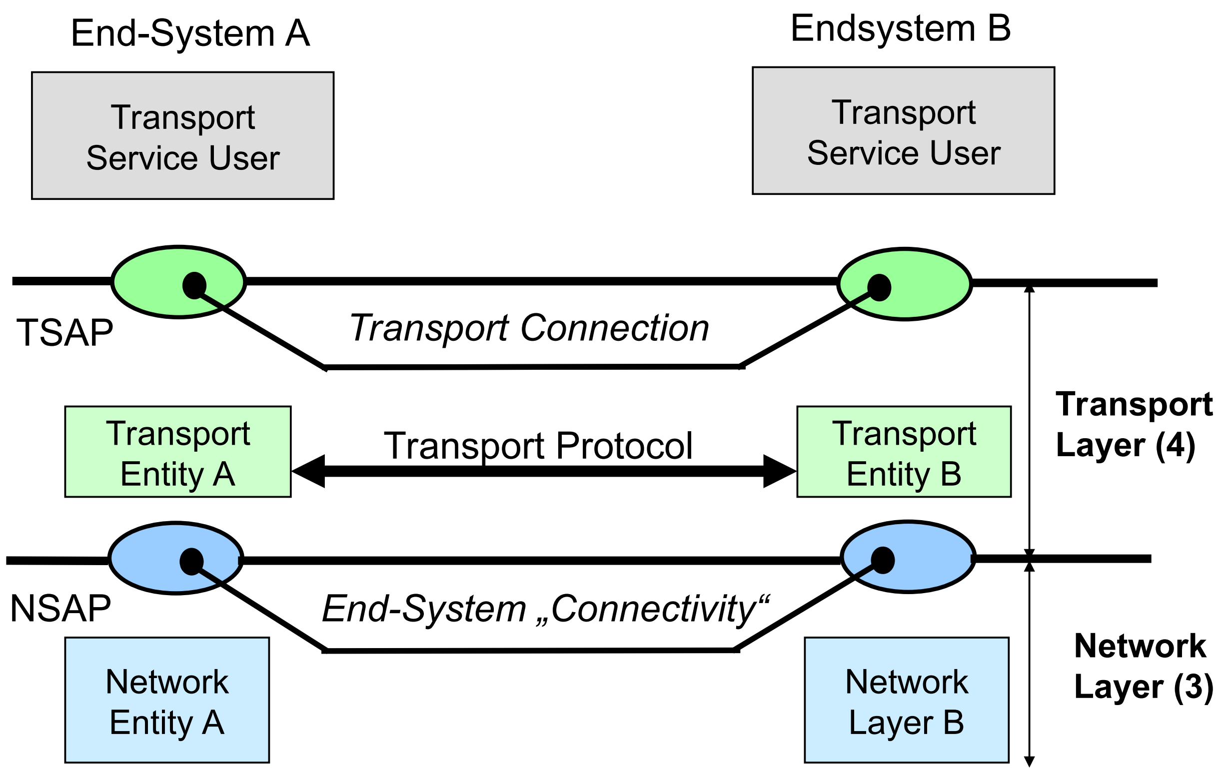
In the following, we will deal with connection-oriented services

- In principle, there are three phases of a connection:
  - Connection establishment phase (Connect)
  - Data transfer phase (Data)
  - Connection release phase (Disconnect)

  For every phase there are specific service primitives

- When talking about the service of a specific layer, we usually add a layer specific prefix to the primitives, e.g.:
  - Transport Layer: T-Connect, T-Data, T-Disconnect
  - Network Layer: N-Connect, N-Data, N-Disconnect (note, however, that the network layer of the Internet provides a connectionless service)

Transport Connections and End-System „Connectivity“



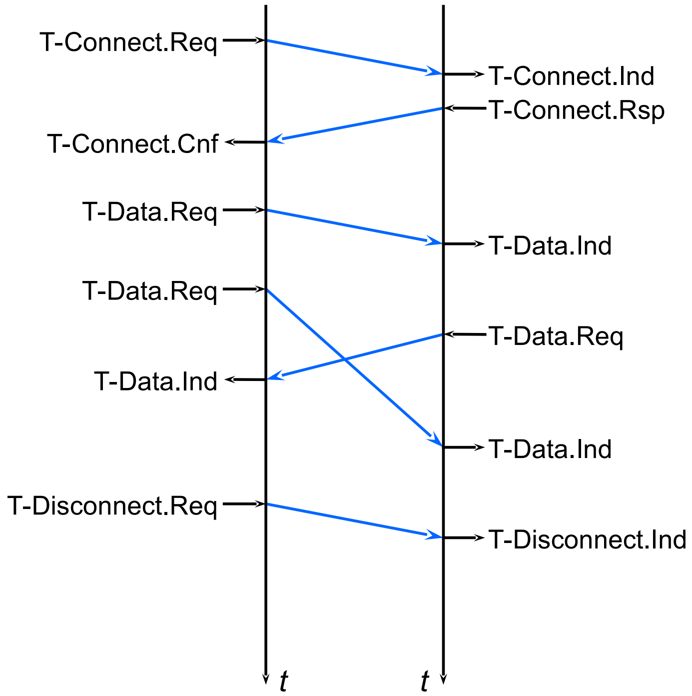
Transport Connection Establishment (OSI Terminology)

- Confirmed service primitive: T-Connect
- Primitive:
  - T-Connect.Request (Destination Address, Source Address)
  - T-Connect.Indication (Destination Address, Source Address)
  - T-Connect.Response (Responding Address)
  - T-Connect.Confirmation (Responding Address)
- Parameters:
  - Destination Address: Address of the called transport service user (= application)
  - Source Address: Address of the calling service user
  - Responding Address: Address of the responding service user (in general, this is the address of the called service user)

Transport Layer Services in a Message Sequence Chart

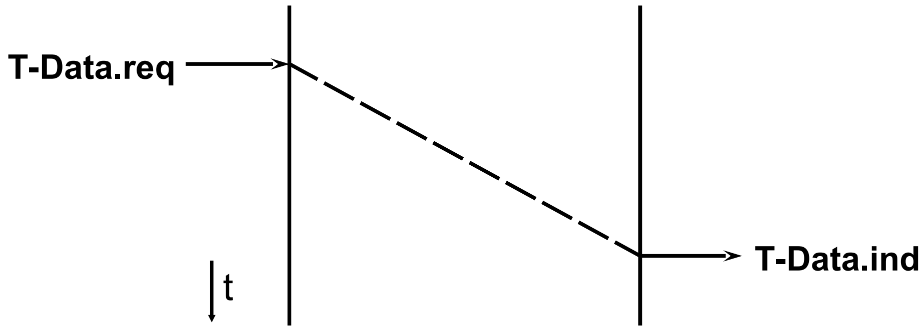
Primitive	**Type**
T-Connect	confirmed
T-Data	unconfirmed
T-Disconnect	unconfirmed (or confirmed)

Example Run:



Data Transfer Service

- Data Transfer Service: T-Data
  - unconfirmed service
- Primitive:
  - T-Data.req (userdata)
  - T-Data.ind (userdata)
- Parameter:
  - Userdata: transport service data unit to be transfered (TSDU, can have arbitrary length)

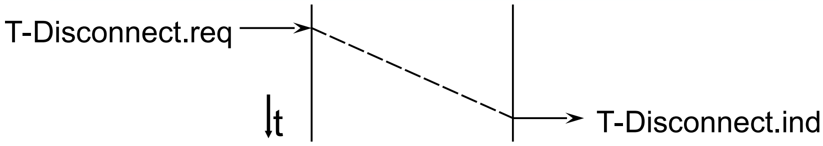


Connection Release (1)

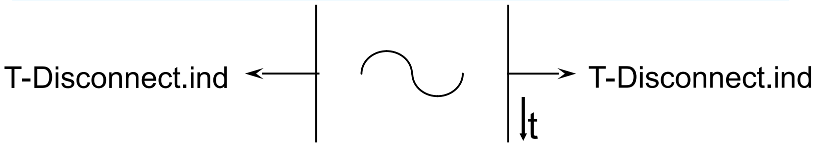
- Unconfirmed release service: T-Disconnect
- Usage:
  - Abrupt teardown of a connection, loss of TSDUs is possible
  - Rejection of a connection establishment request
- Primitives:
  - T-Disconnect.req (userdata)
  - T-Disconnect.ind (cause, userdata)
- Parameters:
  - Cause of the teardown, e.g.:
    - unknown
    - requested by remote user
    - lack of local or remote resources for the transport service provider
    - Quality of service below minimal level
    - error occured in transport service provider
    - can not reach remote transport service user
  - User Data: TSDU to be transfered (max. length e.g. 64 Byte)

Connection Release (2)

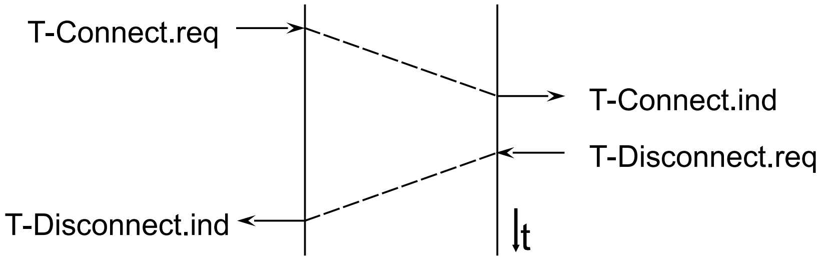
Teardown of a connection by a *service user*:



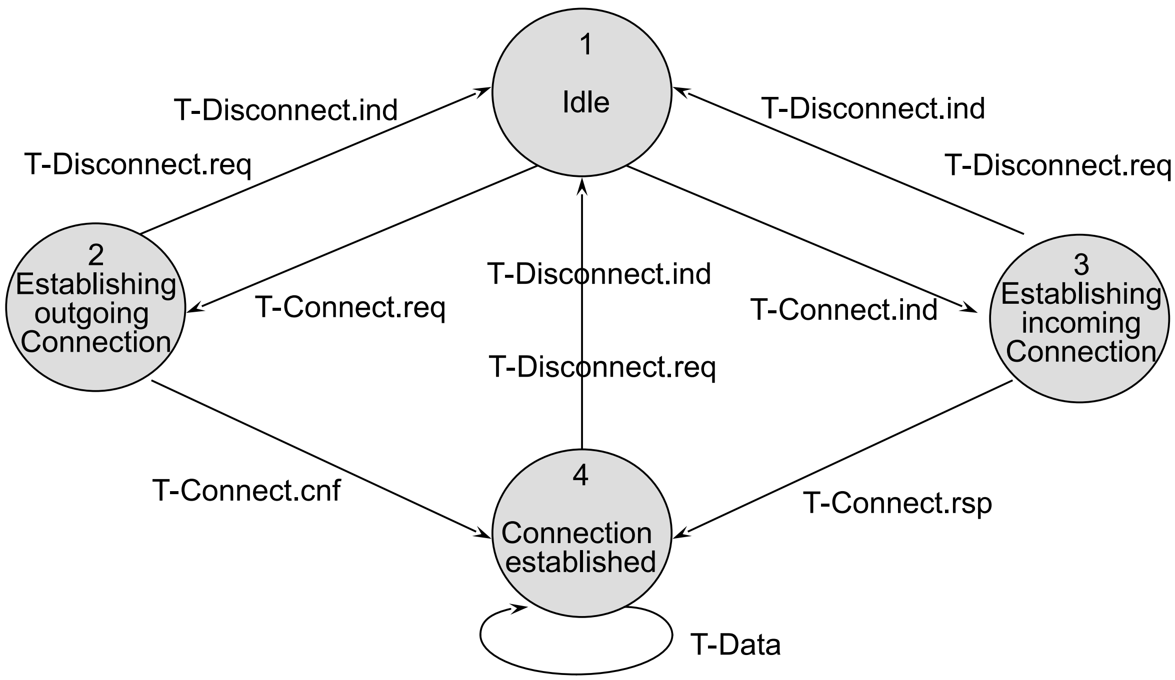
Teardown of a connection by a *service provider*:



Rejection of connection establishment:

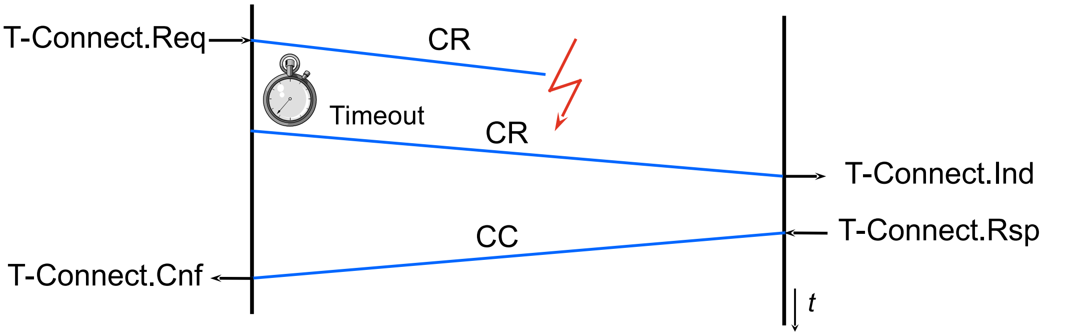


State Diagram for a Transport Service Access Point

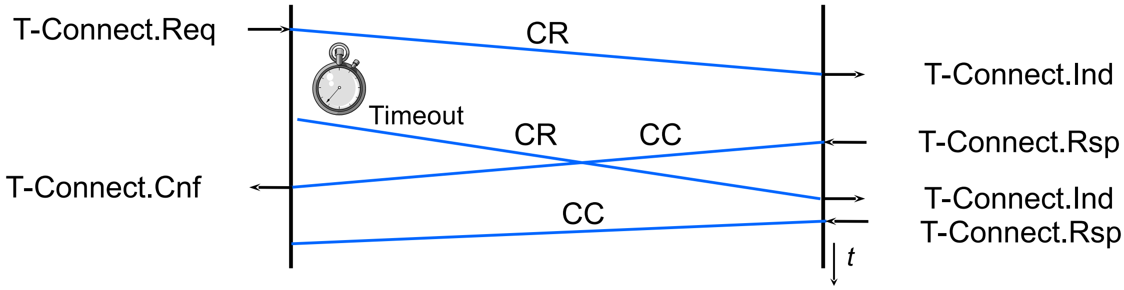


Errors during Connection Establishment

- Loss of CR oder CC TPDU:

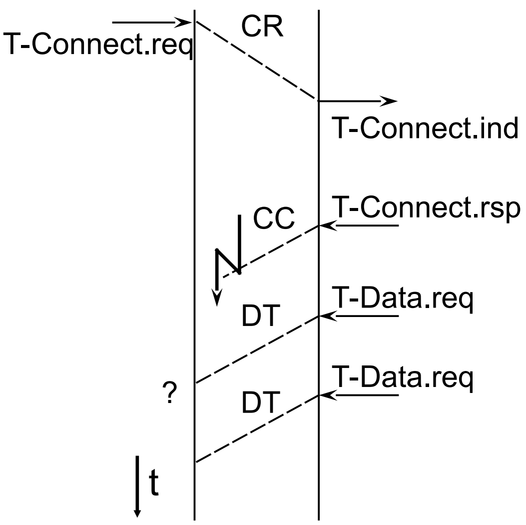


- Duplication of TPDUs:

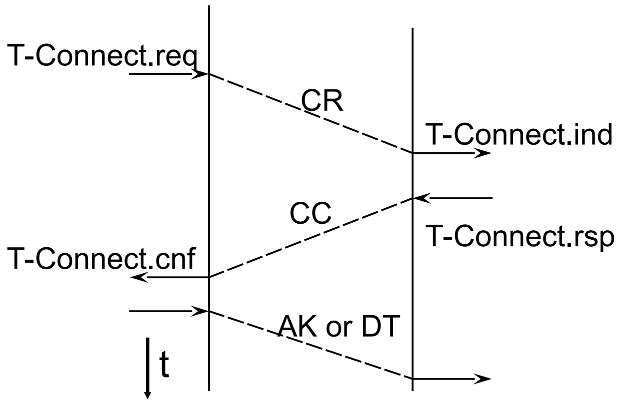


Three-Way Handshake

Problem: Loss of CC TPDU

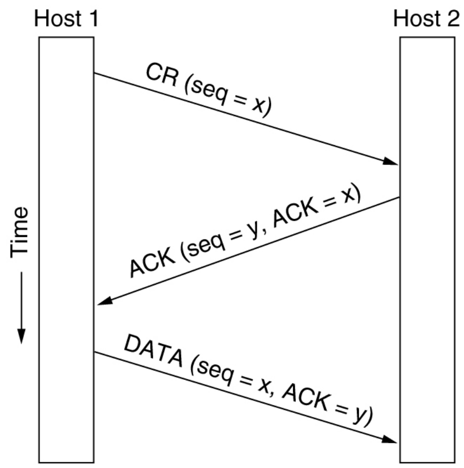


- Solution - Three-Way Handshake during connection establishment:
  - Connection is established, when both connection establishment TPDUs (CR and CC) have been acknowledged
  - Requires an additional AK (Acknowledge) or DT (Data) TPDU



Is Three-Way Handshake Sufficient?

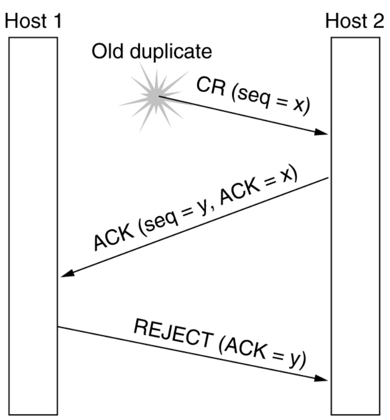
- No, it does not protect against delayed duplicates!
  - Problem: If both the connection request and the connection confirmation are duplicated and delayed, receiver again has no way to ascertain whether this is fresh or an old copy
- Solution: Have the sender answer a question that the receiver asks!
  - Actually: Put ***sequence numbers*** into
    - connection request
    - connection acknowledgement,
    - and connection confirmation
  - Have to be copied by the receiving party to the other side
  - Connection only established if the correct number is provided
  - Sequence numbers should not be re-used too quickly (start with number higher than in last connection; wrap-around)



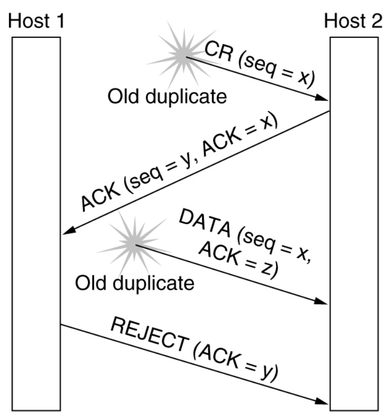
Three-Way Handshake + Sequence Numbers

- Two examples for critical cases (which are handled correctly):

- Connection request appears as an old duplicate:

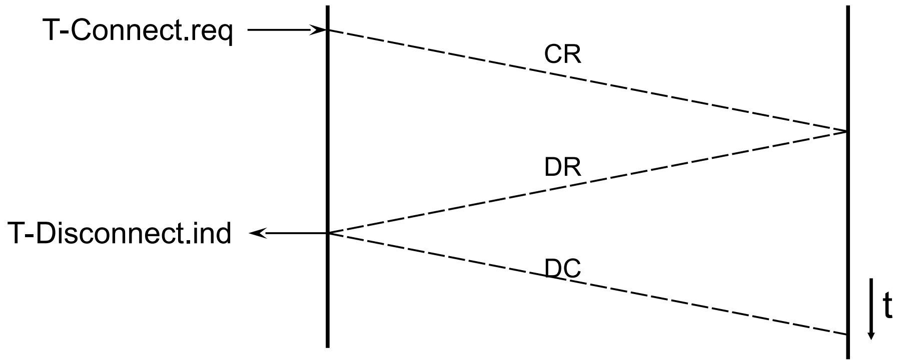


- Connection request & confirmation appear as old duplicates:



Connection Rejection

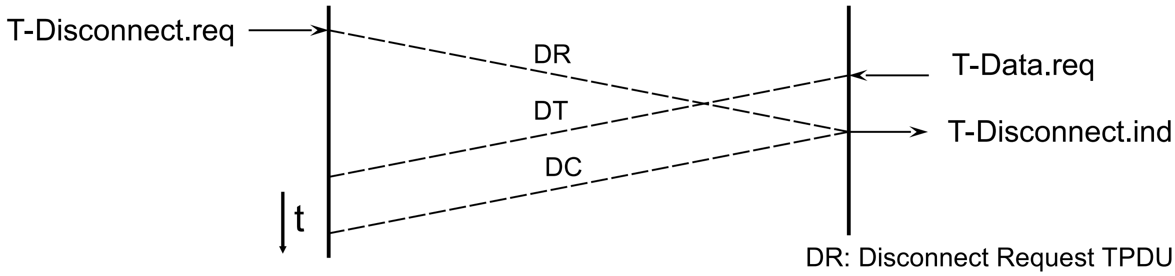
- Refusing an incoming connection request with a Disconnect-Request (DR) or Error-TPDU (reasons for this can be communicated)
  - Reasons:
    - Rejection by transport service user
    - Desired service requirements can not be fulfilled



DR: Disconnect Request TPDU
DC: Disconnect Confirm TPDU

Connection Release (1)

- Normal Release:
  - Teardown of an existing transport connection
  - This can cause loss of data that has not yet been acknowledged
    - The Internet transport protocol TCP avoids loss of data by requiring all sent PDUs to be acknowledged before a connection is closed
  - Variants:
    - Implicit: Teardown of network layer connection (not in the Internet, however, the remote peer entity might become unreachable)
    - Explicit: connection release with Disconnect-TPDUs



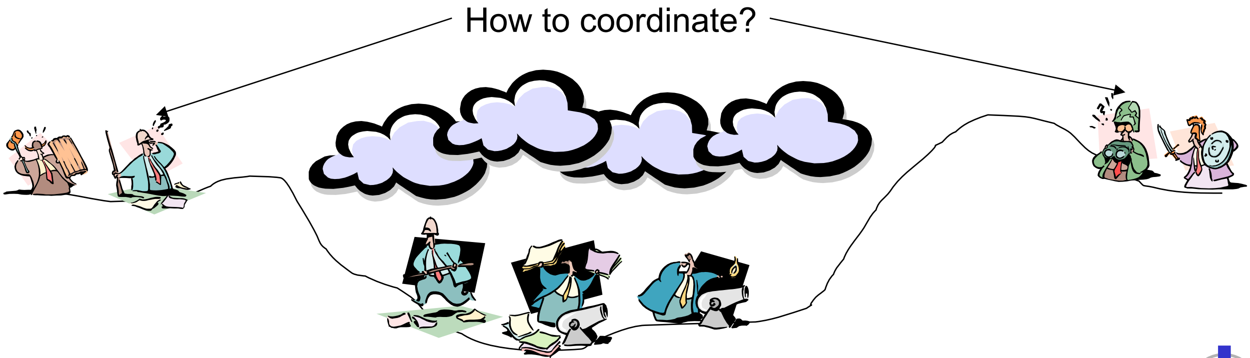
DR: Disconnect Request TPDU
DC: Disconnect Confirm TPDU

Connection Release (2)

- Once connection context between two peers is established, releasing a connection should be easy
  - Goal: Only release connection when both peers have agreed that they have received all data and have nothing more to say
  - I.e., both sides must have invoked a “Close”-like service primitive
- It fact, it is impossible
  - Problem: How to be sure that the other peer knows that you know that it knows that you know … that all data have been transmitted and that the connection can now safely be terminated?
- Analogy: Two army problem

Two Army Problem

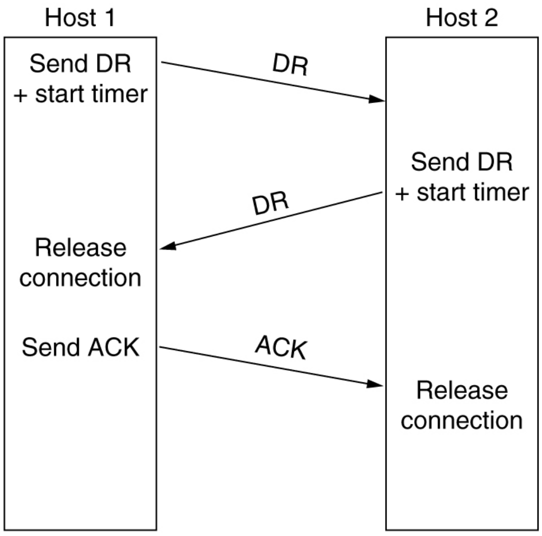
- Coordinated attack
  - Two armies form up for an attack against each other
  - One army is split into two parts that have to attack together – alone they will lose
  - Commanders of the parts communicate via messengers who can be captured
- Which rules shall the commanders use to agree on an attack date?
- Provably unsolvable if the network can loose messages



Connection Release in Practice

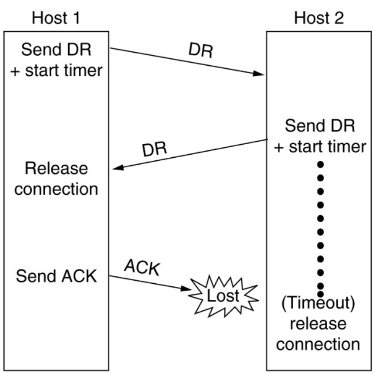
- Two army problem equivalent to connection release
- But: when releasing a connection, bigger risks can be taken

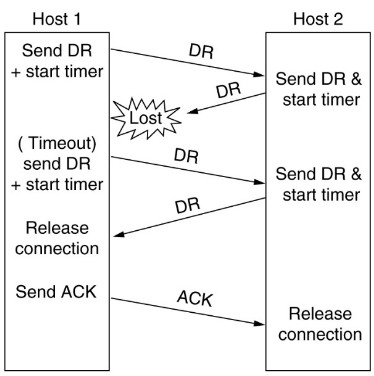
- Usual approach: Three-way handshake again
  - Send disconnect request (DR), set timer, wait for DR from peer, acknowledge it

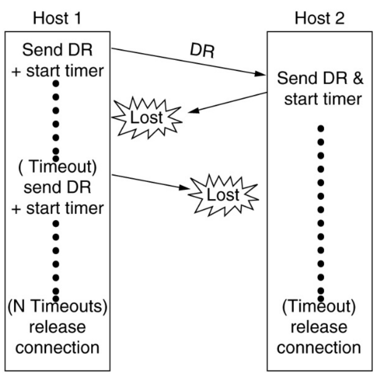


Problem Cases for Connection Release with 3WHS

- Lost ACK solved by (optimistic) timer in Host 2
- Lost second DR solved by retransmission of first DR
- Timer solves (optimistically) case when 2nd DR and ACK are lost







Chapter Overview

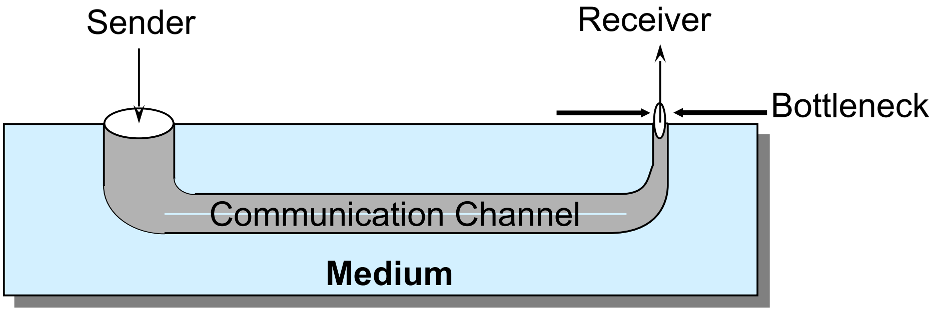
- Transport Layer Services and Protocols
- Addressing and Multiplexing
- Connection Control
- **Flow Control**
- Congestion Control
- Transport protocols in the Internet:
  - User Datagram Protocol (UDP)
  - Transport Control Protocol (TCP)
    - Connection Management
    - Reliable Data Transfer
    - Flow Control
    - Congestion Control
    - Performance

Motivation: Controlling Overload Situations

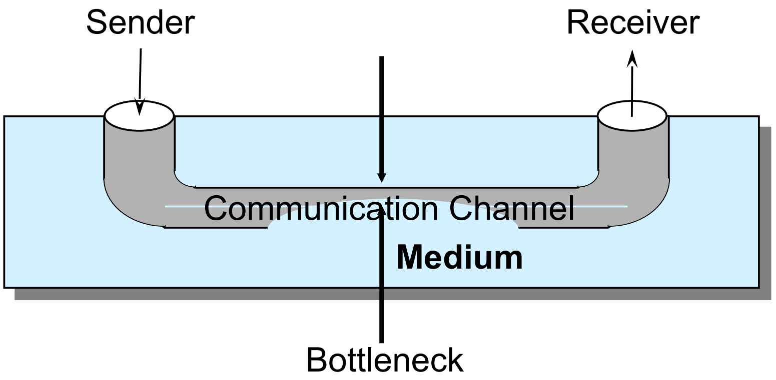
- Usually, multiple systems are involved in a communication taking place:
  - the system initiating the communication
  - the responding system
  - the network between initiator and responder with its intermediate systems
- In order to avoid overload situations:
  - The amount of data exchanged has to be adapted to the current capabilities (i.e. available resources) of the systems involved
  - Otherwise a couple of problems may arise (*performance bottlenecks*; see following slides)

Bottlenecks in Communication Systems

Bottleneck at the Receiver

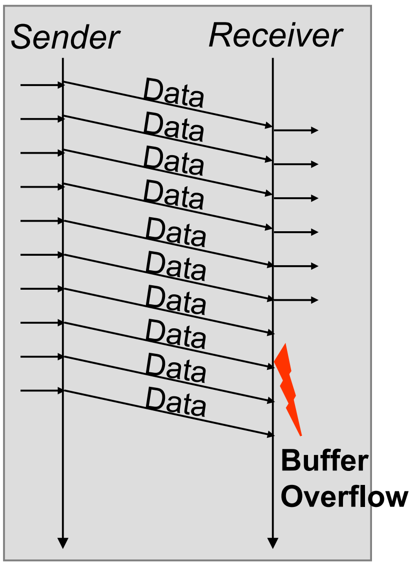


Bottleneck in the Communication Channel



Bottleneck in Receiver

- *Assumption:*
  - The network does not represent a bottleneck; it can deliver all packets sent by the sender
- *Reasons for bottleneck in receiver:*
  - Communicating end systems have different performance characteristics (fast sender & slow receiver)
  - Receiver has to receive packets from many senders
- *Consequences:*
  - Receiver can not keep up with processing all incoming packets
  - Receive buffer overflow
  - Data gets lost



Example: Buffer Overflow in a Point-to-Point Connection

Fast Sender

Slow Receiver

Flow Control

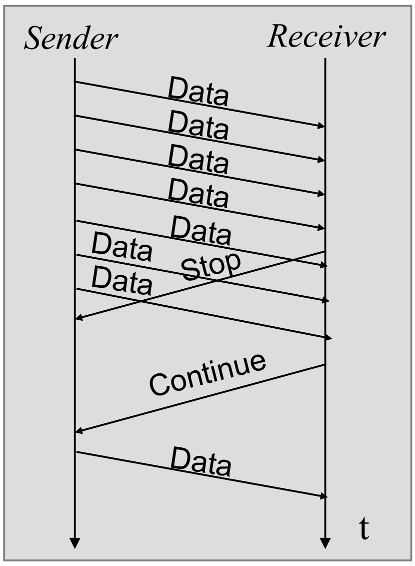
- Task:
  - To protect a receiver from having to process too many packets from a faster sender
- Where provided:
  - At the link layer to prevent overload of „forwarding segments“ (consisting of node-link-node)
  - At higher layers (e.g. network and transport layers) in order to protect overload of connections
- But, flow control in transport layer is more complicated:
  - Many connections, need to adapt the amount of buffer per connection dynamically (instead of just simply allocating a fixed amount of buffer space per outgoing link)
  - Transport layer PDUs can differ widely in size, unlike link layer frames
  - Network’s packet buffering capability further complicates the picture

Flow Control – Buffer Allocation

- Flow control is strongly related to buffer allocation, as the receiver must able to store incoming packets until they can be processed
- Thus, in order to support outstanding packets, the sender either
  - Has to rely on the receiver to process packets as they come in (packets must not be reordered) – unrealistic, or
  - Has to assume that the receiver has sufficient buffer space available
- The more buffer, the more outstanding packets
  - Necessary to obtain highly efficient transmission, recall bandwidth-delay product!
- How does sender have buffer assurance?
  - Receiver slows sender down, when no more buffer space is available (either explicitly or implicitly)
  - Sender can request buffer space
  - Receiver can tell sender: “I have X buffers available at the moment”
    - For sliding window protocols: Influence size of sender’s send window

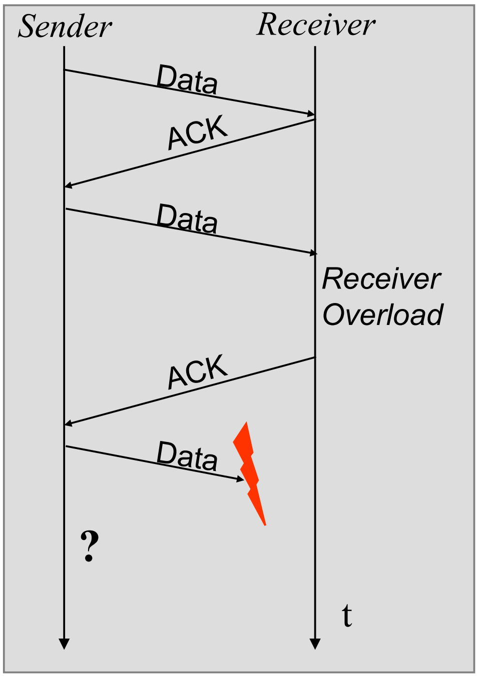
Flow Control with Stop and Continue Messages

- *Easiest Solution*
  - Sender Receiver Flow Control
    - Exchange of explicit notifications
      - **Stop**
      - **Continue**
  - If the receiver can not keep up with the incoming data flow, it sends a **stop** message to the sender
  - If he becomes able to receive again, it sends a **continue** message
- *Example: XON/XOFF Protocol*
    - With ISO 7-Bit-Alphanumerical characters
    - XON is DC_1 (Device Control 1).
    - XOFF is DC_3 (Device Control 3).
    - Can only be used on full duplex communication lines



Implicit Flow Control

- *Idea:*
  - By holding back acknowledgements (ACK or NACK), the sender can be slowed down
  - This basically means, that an error control mechanism is „(ab)used“ for flow control
- *Drawback:*
  - The sender can not distinguish:
    - if his packet(s) got lost, or
    - if the receiver holds back the acknowledgements in order to slow him down (resulting in unnecessary retransmissions)



Credit Based Flow Control

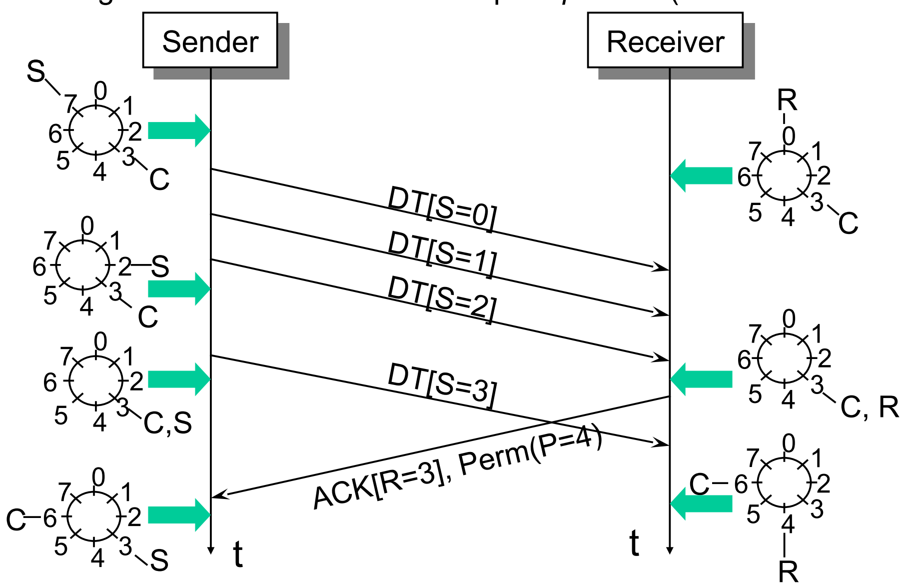
- *Idea:*
  - The receiver gives the sender explicit credit to send multiple packets
  - If the sender runs out of credit (and does not get new credit), it stops sending and waits for new credit
  - However, this requires that explicit error control is provided in order to be able to recover from loss of credit messages
- *Implementation alternatives:*
  - Absolute credit:
    - The receiver gives an absolute credit to the sender (e.g. “you may send 5 more packets”)
    - Drawback: potential ambiguities because the sender receives credit at a different point in time than when the receiver sent it
  - Credit window („sliding window“):
    - Credit is given relatively to an acknowledged packet

Flow Control – Permits and Acknowledgements

- Distinguish:
  - Permits (“Receiver has buffer space, go ahead and send more data”)
  - Acknowledgements (“Receiver has received certain packets”)
- Should be separated in real-world protocols!
- Can be combined with dynamically changing buffer space at the receiver
  - Due to, e.g., different speed with which the application actually retrieves received data from the transport layer
  - Example: TCP
- Please note: some protocols ask for the next packet not yet received in order, while other protocols acknowledge the last packet received in order (see next two examples)

Credit Based Flow Control: Sliding Window

- *Example:* Sliding Window Mechanism with explicit *permits* (= credit notification)



- S: Sequence number (of last sent packet)
- R: Next expected sequence number = Acknowledges up to sequence number R-1
- C: Upper window limit (maximum permitted sequence number)
- P: Number of packets that may be transmitted, starting from next expected packet

One More Example of Flow Control with ACK/Permit Separation

- Arrows show direction of transmission, "…" indicates lost packet
- Potential deadlock in step 16 when control PDU is lost and not retransmitted

	A	Message	B	Comments
1	→	< request 8 buffers>	→	A wants 8 buffers
2	←	<ack = 15, buf = 4>	←	B grants messages 0-3 only
3	→	<seq = 0, data = m0>	→	A has 3 buffers left now
4	→	<seq = 1, data = m1>	→	A has 2 buffers left now
5	→	<seq = 2, data = m2>	• • •	Message lost but A thinks it has 1 left
6	←	<ack = 1, buf = 3>	←	B acknowledges 0 and 1, permits 2-4
7	→	<seq = 3, data = m3>	→	A has 1 buffer left
8	→	<seq = 4, data = m4>	→	A has 0 buffers left, and must stop
9	→	<seq = 2, data = m2>	→	A times out and retransmits
10	←	<ack = 4, buf = 0>	←	Everything acknowledged, but A still blocked
11	←	<ack = 4, buf = 1>	←	A may now send 5
12	←	<ack = 4, buf = 2>	←	B found a new buffer somewhere
13	→	<seq = 5, data = m5>	→	A has 1 buffer left
14	→	<seq = 6, data = m6>	→	A is now blocked again
15	←	<ack = 6, buf = 0>	←	A is still blocked
16	• • •	<ack = 6, buf = 4>	←	Potential deadlock

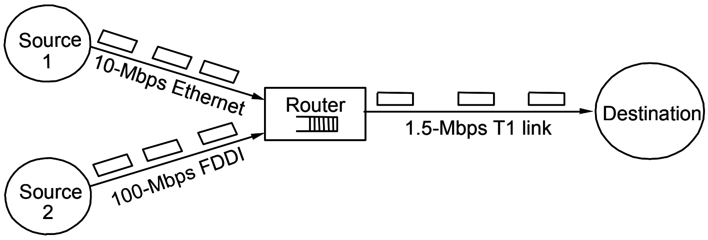
Chapter Overview

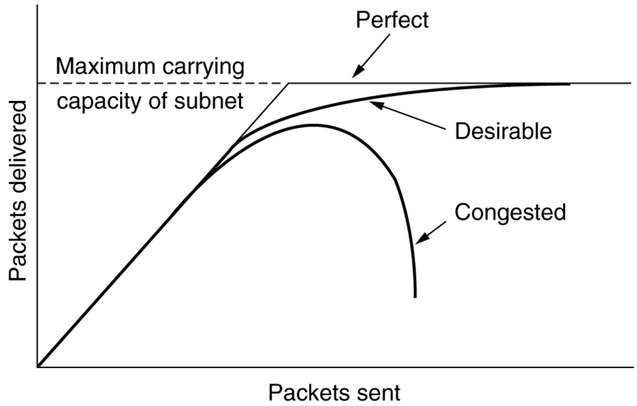
- Transport Layer Services and Protocols
- Addressing and Multiplexing
- Connection Control
- Flow Control
- **Congestion Control**
- Transport protocols in the Internet:
  - User Datagram Protocol (UDP)
  - Transport Control Protocol (TCP)
    - Connection Management
    - Reliable Data Transfer
    - Flow Control
    - Congestion Control
    - Performance

Why Congestion Control?

Recall overload in network:

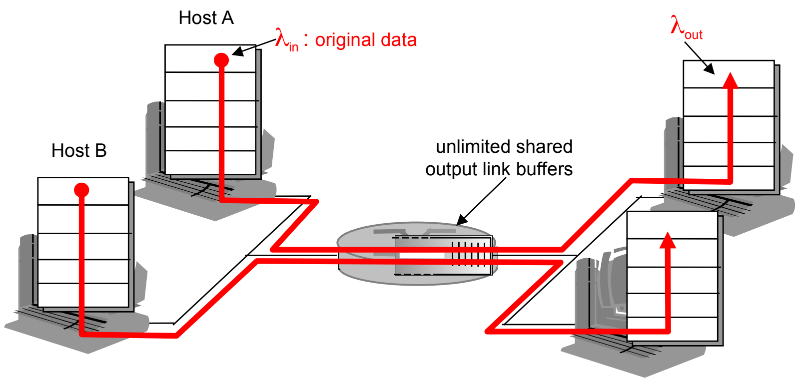
- Any *network* can only transport a bounded amount of traffic per unit time
  - Link capacities are limited, processing speed in routers, buffer space, …
- When sources inject more traffic into the network than its nominal capacity, ***congestive collapse*** (usually) results
- Consequence: packets are lost!

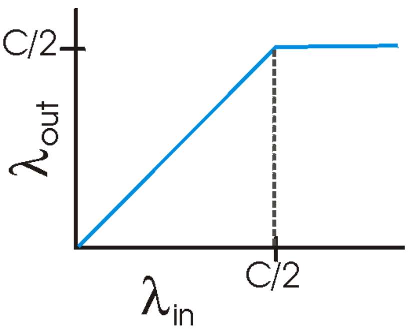


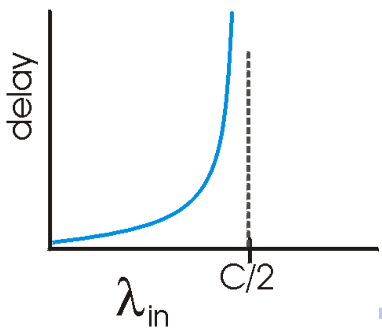


Causes/Costs of Congestion: Scenario 1

- Two senders, two receivers
- One router, infinite buffers
- No retransmission



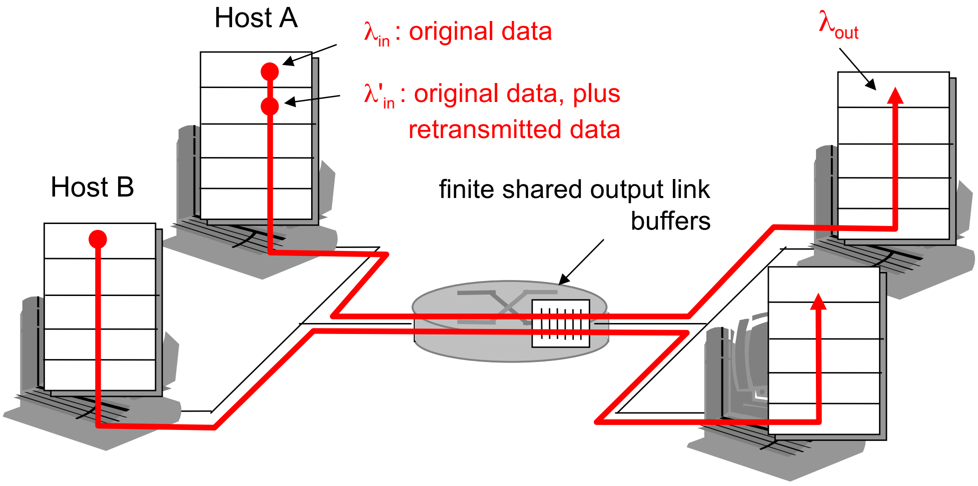




- Large delays when congested
- Maximum possible throughput achieved

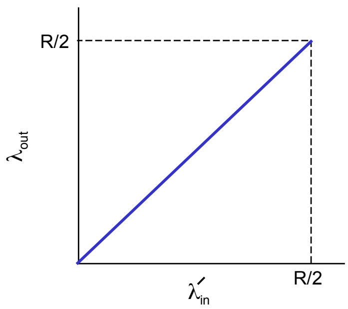
Causes/Costs of Congestion: Scenario 2

- One router, *finite* buffers
- Senders retransmit only lost packets (perfect knowledge)

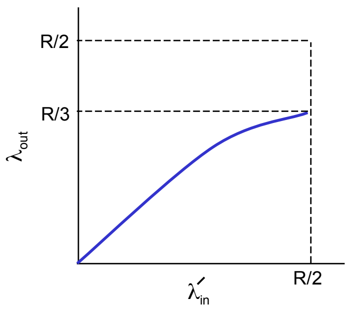


Causes/Costs of Congestion: Scenario 2

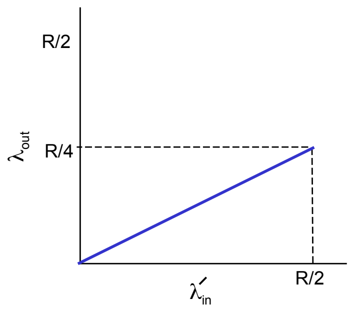
- Always: $\lambda_{in} = \lambda_{out}$ (goodput)
- "Perfect" retransmission only when loss: $\lambda'_{in} > \lambda_{out}$
- Retransmission of delayed (not lost) packet makes λ'_{in} larger (than perfect case) for same λ_{out}



Send only when „room“ in router



Retransmit only lost packets



Retransmit after timeout

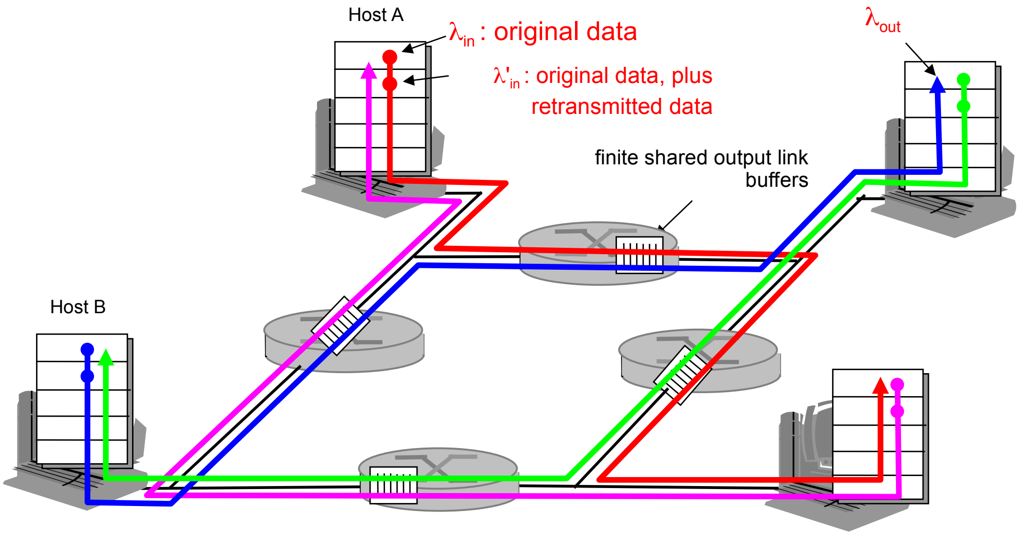
"Costs" of congestion:

- More work (retransmissions) for given "goodput"
- Unneeded retransmissions: link carries multiple copies of packets

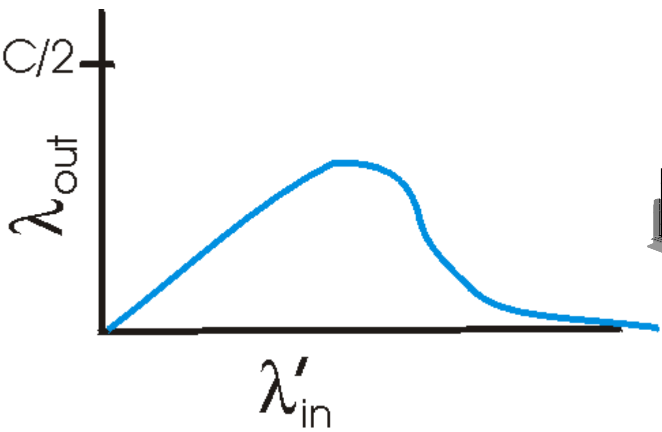
Causes/Costs of Congestion: Scenario 3

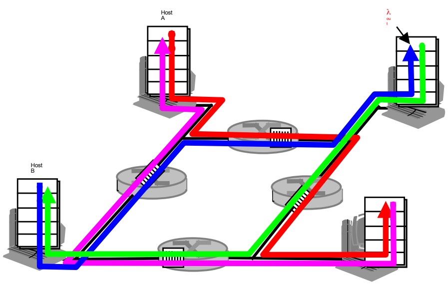
- Four senders
- Multihop paths
- Timeout/Retransmit

Q: What happens as λ_{in} and λ'_{in} increase ?



Causes/Costs of Congestion: Scenario 3





Another "cost" of congestion:

- When packet dropped, any "upstream transmission capacity used for that packet was wasted!

Intermediate Summary: Need For Congestion Control

- Congestion control is essential to avoid *snowball effects*
  - Once a network is overloaded, it will loose packets (buffer overflows, etc.)
  - Once a reliable transport protocol detects packet loss, it will retransmit the lost packets
  - These retransmissions further increase the load in the network
  - More packets will be lost
  - More retransmissions will happen
  - Etc.

- Mechanisms to damper/avoid such oscillations are necessary

Adapt Sending Rate to Network Capacity

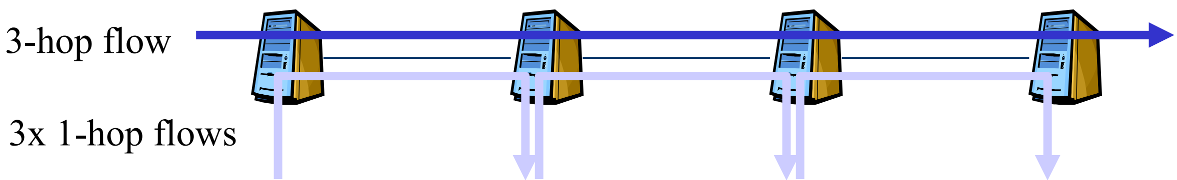
- Sending rate of each source has to be adapted to the network's actual, current capacity
- ***Global issue***: depends on all routers, forwarding disciplines, load injected by other terminals, etc.
- Made complicated by interaction of mechanisms of many different layers

Layer	Policies
Transport	• Retransmission policy • Out-of-order caching policy • Acknowledgement policy • Flow control policy • Timeout determination
Network	• Virtual circuits versus datagram inside the subnet • Packet queueing and service policy • Packet discard policy • Routing algorithm • Packet lifetime management
Data link	• Retransmission policy • Out-of-order caching policy • Acknowledgement policy • Flow control policy

- ***Flow control***, on the other hand: ***local issue***!
  - Source must not overrun its destination
  - Only source and destination are involved (possibly separated by multiple hops, but that is irrelevant)

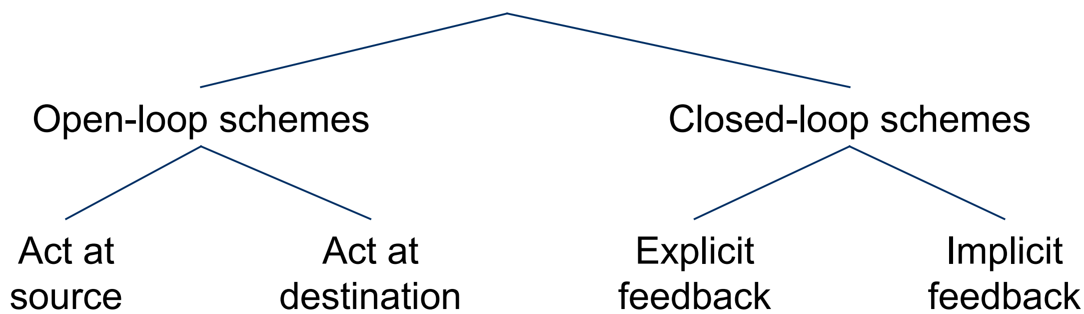
Desirable Properties of Congestion Control

- Congestion control should result in many packets delivered at short delays
  - Protect network from congestive collapse but still transport as much data as possible
- Fairness
  - Give all participating flows a “fair” share of available capacity
  - Does fair mean “equal”? Video conference = telnet session?
  - Should path lengths be considered?



Design Options for Congestion Control Mechanisms

- *Open loop:* design system up front so that it will work correct, no corrections at runtime necessary
- *Closed loop:* use some sort of feedback to allow sender to adapt to current situation
- *Explicit feedback:* point where congestion occurs informs sender
- *Implicit feedback:* no explicit action taken; congestion is deduced by sender from the network’s behavior (e.g., missing acknowledgements)



Possible Actions

- Increase capacity – activate additional links, routers, …
  - Usually not practical, at least on short timescales

- ***Reservations*** and admission control – do not admit additional traffic when network is nearing capacity limit
  - Usually only applicable to circuit-switched (or similar) networks
  - Feedback about network state only relatively rarely – akin to open-loop control

- Reduce load at smaller granularity
  - Have some/all sources reduced their offered load without terminating on-going sessions
  - Usually requires ***feedback*** from the network (closed loop)

Possible Actions – Taxonomy

- Router-centric vs. host-centric
  - Where is/are information gathered, decisions made, actions taken?
  - Usually not either/or, but more a question of emphasis

- Window-based vs. rate-based
  - How is the allowed amount of traffic injected into the network described?
  - By a ***rate*** – so and so many bytes per second?
  - By a ***congestion window*** – as a set of sequence numbers/amount of bytes that may be injected into the network before further permits are received?

  - Further options exist, e.g., credit-based congestion control, but are much less popular

Router Actions: Dropping Packets

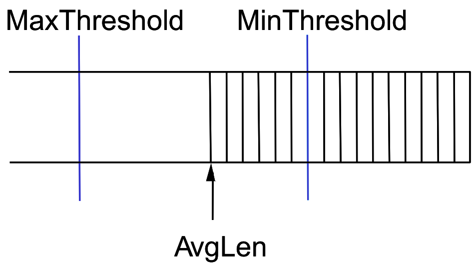
- Suppose a router's buffer space is full and a packet arrives
  - Obviously, there is one packet too many, and one of them has to be dropped
- One candidate: the newly arriving packet
  - Intuition: "old" packets are more valuable than new ones, e.g., for a go-back-n transport protocol
  - A so-called ***drop-tail queue***
- Other option: a packet that is already in the queue for quite some time
  - Intuition: For multi-media traffic, new packets are more important than old ones
  - Maybe even try to drop a packet from the same flow as the newly arriving packet's, but that might not be feasible (overhead)

Dropping Packets = Implicit Feedback

- Dropping a packet constitutes an implicit feedback action
  - The sending transport protocol can detect this packet loss (if it so desires, e.g., by missing acknowledgements)
  - Assumption: Packet loss is ***only*** (or predominantly) caused by congestion
  - Then: Correct action by a transport protocol is to reduce its offered load
  - Assumption is by and large true in wired networks but not in wireless networks
- In open-loop congestion control, packets arriving to a full queue should never happen
  - Else, resource reservations were not done correctly

Avoiding Full Queues – Proactive Actions?

- When packets arrive to a full queue, things are pretty bad already
  - Is there any chance we can try to avoid such a situation, without having to recur to open-loop control?
- Provide ***proactive*** feedback! (***Congestion avoidance***)
  - Do not only react when the queue is full, but already when the "congestion indicator" has crossed some threshold
  - E.g., when the average queue length has exceeded a lower threshold
  - E.g., when the outgoing link utilization is persistently higher than a threshold
  - E.g., …
  - Router is then called to be in a ***warning state***



Proactive Action: Choke Packets

- Once a router decides it is congested (or that it likely will be in the near future):

  Send out choke packets
- A choke packet tells the source of a packet arriving during warning state to slow down its sending rate
- Obvious problem: In an already congested network, more packets are injected to remedy congestion
  - Questionable
- Second problem: How long does it take before source learns about congestion?
  - How much data has already been injected?
  - Think in terms of the data rate-delay product

Proactive Action: Warning Bits

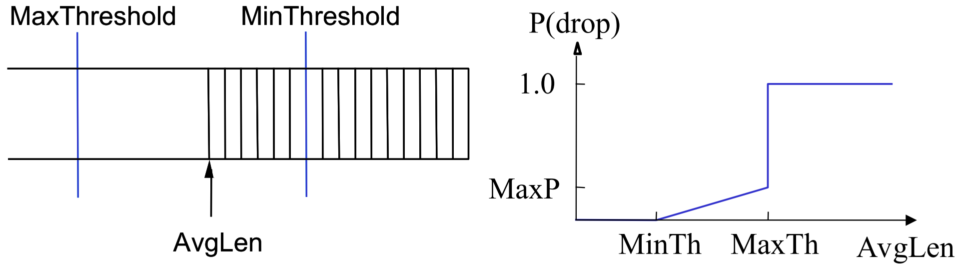
- Once a router decides it is congested (or that it likely will be in the near future):

  Set a ***warning bit*** in all packets that it sends out

  - Destination will copy this warning bit into its acknowledgement packet
  - Source receives the warning bit and reduces its sending rate

Proactive Actions: Random Early Detection (RED)

- Exploit lost packets as implicit feedback, but not only when the queue is already full
- Instead: early on deliberately drop some packets to provide feedback
  - Sounds cruel, but it might save later packets from being dropped
- Dropping probability can be increased as a router becomes more and more congested
  - E.g., as the queue becomes longer and longer



What Happens After Feedback Has Been Received?

- Once feedback of some sort has been received by a sending transport protocol instance, it has to react on it
- Rate-based protocols: Reduce rate, e.g., by a constant factor
  - Relatively easy
  - Question: How to increase rate again?
- Window-based protocols: Shrink the ***congestion window***
  - By how much?
  - How to grow the window in the first place?
  - What to do with a large window – sending out bursts not a good idea

We will discuss these questions with TCP as a case study

Chapter Overview

- Transport Layer Services and Protocols
- Addressing and Multiplexing
- Connection Control
- Flow Control
- Congestion Control
- Transport protocols in the Internet:
  - **User Datagram Protocol (UDP)**
  - Transport Control Protocol (TCP)
    - Connection Management
    - Reliable Data Transfer
    - Flow Control
    - Congestion Control
    - Performance

UDP: User Datagram Protocol [RFC 768]

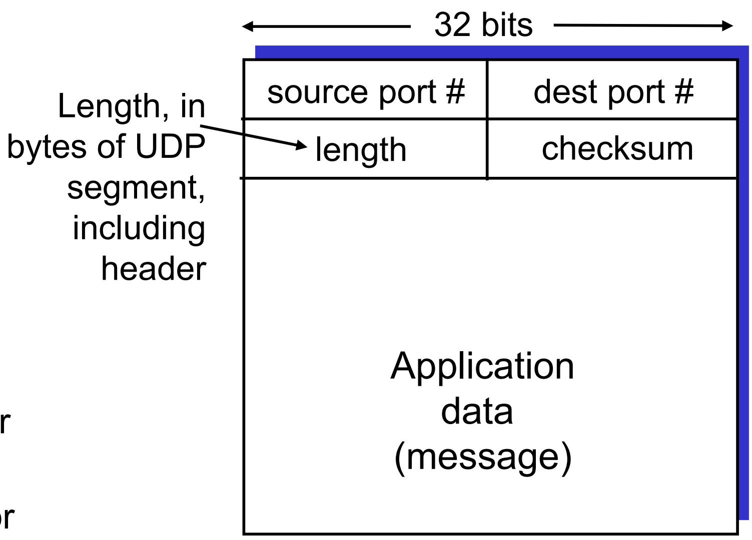
- "No frills," "bare bones" Internet transport protocol
- "Best effort" service, UDP segments may be:
  - Lost
  - Delivered out of order to app
- *Connectionless:*
  - No handshaking between UDP sender, receiver
  - Each UDP segment handled independently of others

Why is there a UDP?

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

User Datagram Protocol (continued)

- Often used for streaming multimedia apps
  - Loss tolerant
  - Rate sensitive
- Other UDP uses
  - DNS
  - SNMP
- Reliable transfer over UDP: add reliability at application layer
  - Application-specific error recovery
  - Please, do not do this for applications that generate large traffic volumes



UDP segment format

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 (or include checksum field in addition and compare to zero):
  - NO - error detected
  - YES - no error detected. *But maybe errors nonetheless?* (recall link layer chapter)

Internet Checksum 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

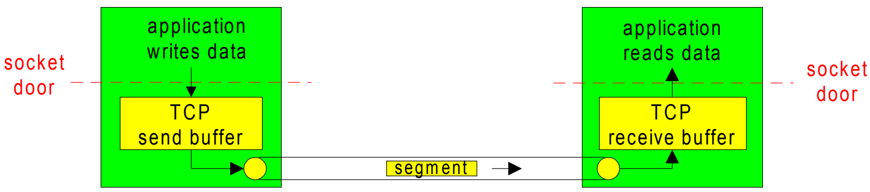
		1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
		1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
sum		1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum		0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

Chapter Overview

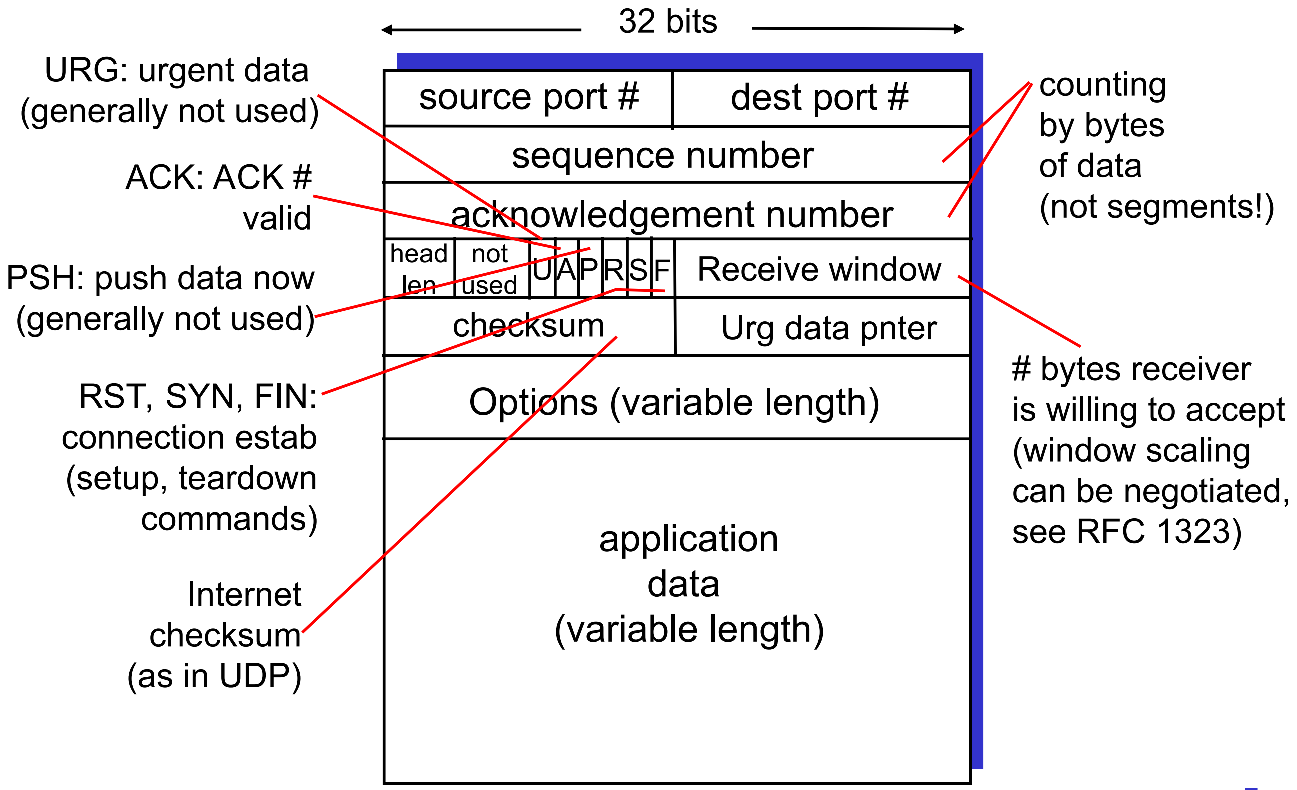
- Transport Layer Services and Protocols
- Addressing and Multiplexing
- Connection Control
- Flow Control
- Congestion Control
- Transport protocols in the Internet:
  - User Datagram Protocol (UDP)
  - **Transport Control Protocol (TCP)**
    - Connection Management
    - Reliable Data Transfer
    - Flow Control
    - Congestion Control
    - Performance

Transport Control Protocol (TCP)

- Point-to-point:
  - one sender, one 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 msgs) initializes sender & receiver state before data exchange
- Flow controlled:
  - Does not overwhelm receiver



TCP Segment Structure

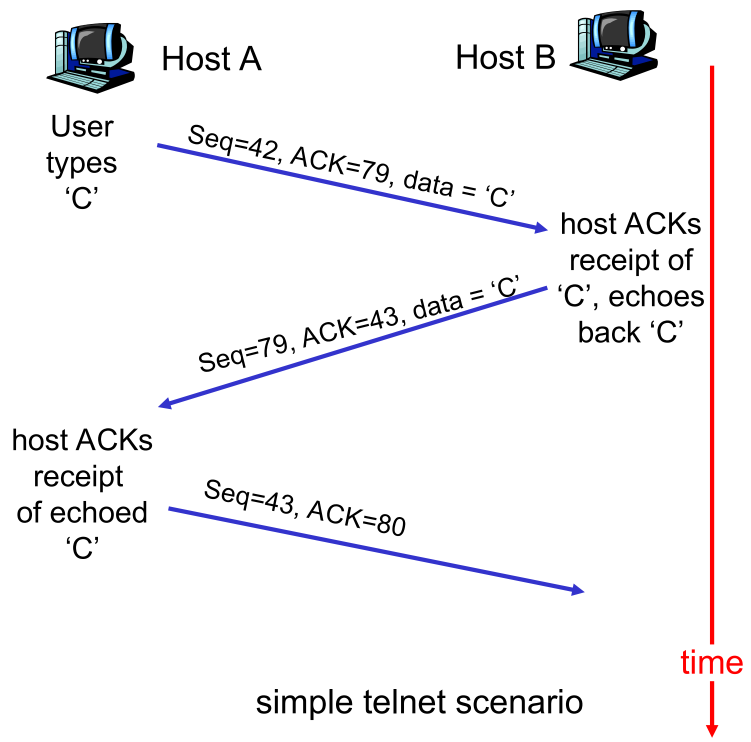


TCP Sequence Numbers and ACKs

Seq. #'s:
- Byte stream "number" of first byte in segment's data

ACKs:
- Seq # of next byte expected from other side
- Cumulative ACK

Q: How does receiver handle out-of-order segments?
- A: TCP spec doesn't say, - up to implementor



simple telnet scenario

TCP Round Trip Time and Timeout

Recall: Reliable data transfer needs to handle timeouts

Q: 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

Q: 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`

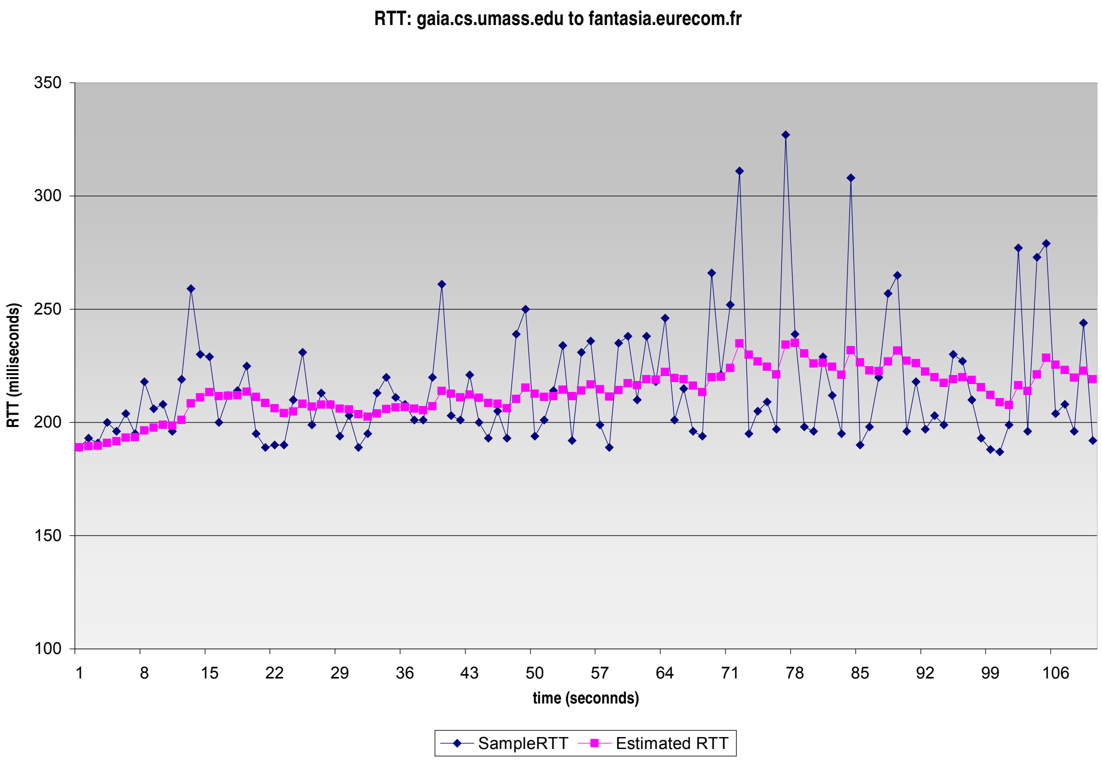
TELEMATIK Rechnernetze

TCP Round Trip Time and Timeout

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

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

Example RTT Estimation



TCP Round Trip Time and Timeout

Setting the timeout

- **`EstimatedRTT`** plus "safety margin"
  - Large variation in **`EstimatedRTT`** **->** larger safety margin
- First estimate of how much SampleRTT deviates from EstimatedRTT:

```
DevRTT = (1-β)*DevRTT +
             β*|SampleRTT-EstimatedRTT|

(typically, β = 0.25)
```

Then set timeout interval:

```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```

Chapter Overview

- Transport Layer Services and Protocols
- Addressing and Multiplexing
- Connection Control
- Flow Control
- Congestion Control
- Transport protocols in the Internet:
  - User Datagram Protocol (UDP)
  - Transport Control Protocol (TCP)
    - **Connection Management**
    - Reliable Data Transfer
    - Flow Control
    - Congestion Control
    - Performance

TCP Connection Establishment

- TCP connections can be established in **active** (connect) or **passive** mode (using listen/accept)
  - **Active Mode:** Requesting a TCP connection with a specified transport service user (identified via IP address and port number)
  - **Passive Mode:** an application informs TCP, that it is ready to accept an incoming connection
    - Can specify a specific socket, on which an incoming connection is expected, or
    - all incoming connections will be accepted (unspecified passive open)
    - Upon an incoming connection request, a new socket is created that will serve as connection endpoint
  - Note: The connection is established by the TCP-entities without further interaction with the application, i.e. there is no service primitive corresponding to T-CONNECT.Rsp

Connection Identification in TCP

- A TCP connection is setup
  - Between a single sender and a single receiver
  - More precisely, between application processes running on these systems
  - TCP can multiplex several such connections over the network layer, using the port numbers as Transport SAP identifiers

- A TCP connection is thus identified by a four-tuple:

  (Source Port, Source IP Address,
  Destination Port, Destination IP Address)

TCP Connection Management (1)

Three way handshake:

Step 1:

- Client host sends TCP SYN segment (~ CR-PDU) to server
  - Specifies initial seq #
  - No data

Step 2:

- Server host receives SYN, replies with SYNACK segment (~ CC-PDU)
  - Server allocates buffers
  - Specifies server initial seq. #

Step 3:

- Client receives SYNACK, replies with ACK segment, which may contain data

TCP Connection Management (2)

Closing a connection:

Client closes socket (sockets will be treated in next chapter):

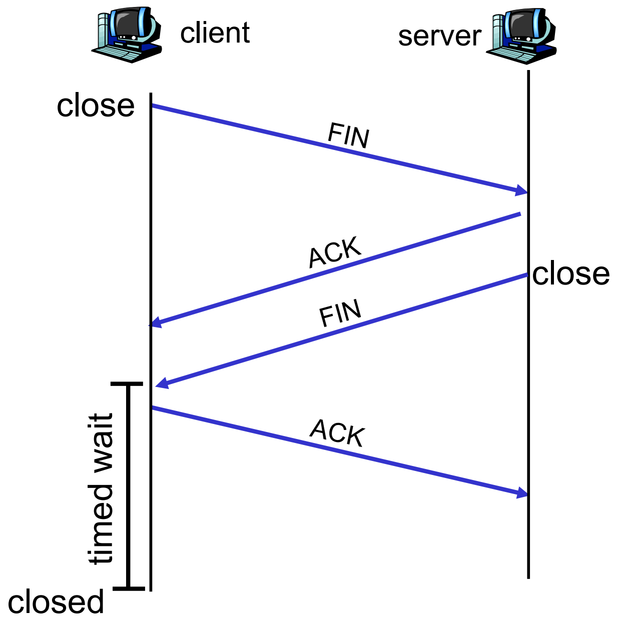
```
clientSocket.close();
```

Step 1:

Client end system sends TCP FIN (~ DR-PDU) control segment to server

Step 2:

Server receives FIN, replies with ACK
Closes connection, sends FIN (~ DC-PDU)



TCP Connection Management (3)

Step 3:

Client receives FIN, replies with ACK.

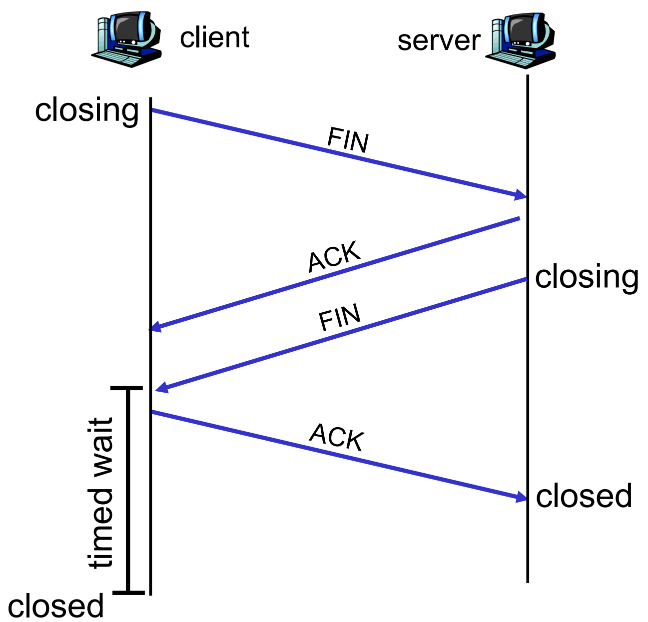
- Enters "timed wait" - will respond with ACK to received FINs

Step 4:

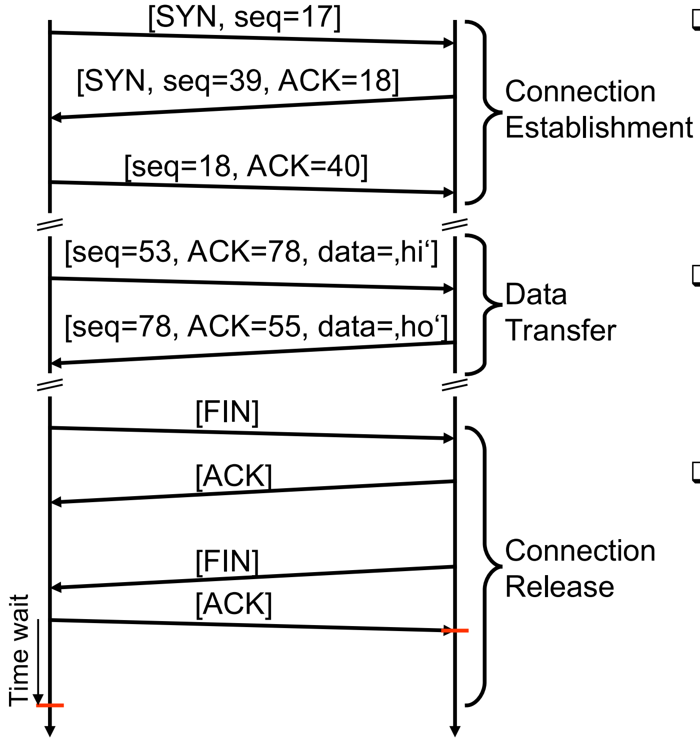
Server, receives ACK. Connection closed

Note:

With small modification, can handle simultaneous FINs

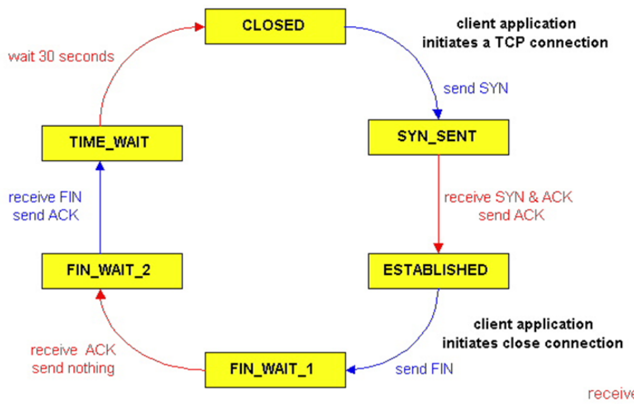


A TCP Connection in all Three Phases



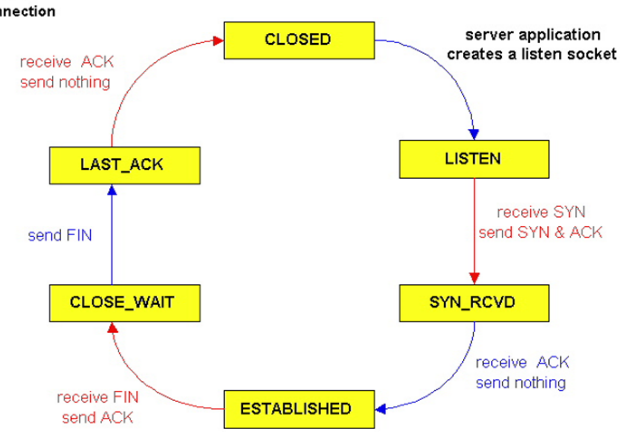
- ❑ Connection Establishement
  - ❑ 3-Way-Handshake
  - ❑ Negotiation of window size and sequence numbers
- ❑ Data transfer
  - ❑ Piggybacking of acknowlegements
- ❑ Connection Release
  - ❑ Confirmed (!)
  - ❑ Avoids loss of data that has already been sent

TCP Connection Management: State Transitions

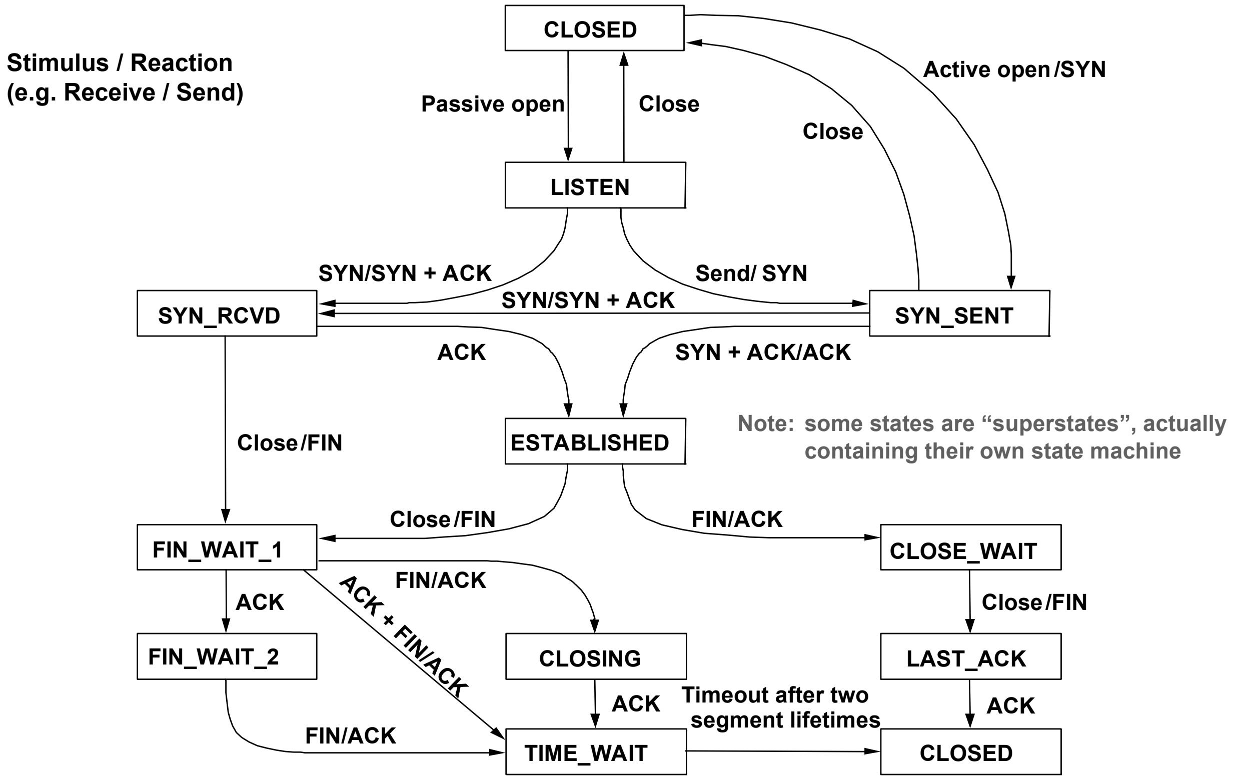


TCP Client Lifecycle

TCP Server Lifecycle



TCP Connection Management: State Diagram



Chapter Overview

- Transport Layer Services and Protocols
- Addressing and Multiplexing
- Connection Control
- Flow Control
- Congestion Control
- Transport protocols in the Internet:
  - User Datagram Protocol (UDP)
  - Transport Control Protocol (TCP)
    - Connection Management
    - **Reliable Data Transfer**
    - Flow Control
    - Congestion Control
    - Performance

TCP Reliable Data Transfer

- TCP creates reliable data 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, we consider a simplified TCP sender:
  - Ignore duplicate acks
  - Ignore flow control, congestion control

TCP Sender Events:

Data received from application:

- Create segment with seq #
- Seq # 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`**

Timeout:

- Retransmit segment that caused timeout
- Restart timer

Ack received:

- If it acknowledges previously unacked segments
  - Update what is known to be acked
  - Start timer if there are outstanding segments

TCP Sender (simplified)

```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum
 loop (forever) {
   switch(event)
   event: data received from application above
       create TCP segment with sequence number NextSeqNum
       if (timer currently not running)
           start timer
       pass segment to IP
       NextSeqNum = NextSeqNum + length(data)

   event: timer timeout
       retransmit not-yet-acknowledged segment with
            smallest sequence number
       start timer

   event: ACK received, with ACK field value of y
       if (y > SendBase) {
           SendBase = y
           if (there are currently not-yet-acknowledged segments)
                start timer
           }

 }  /* end of loop forever */
```

Comments:

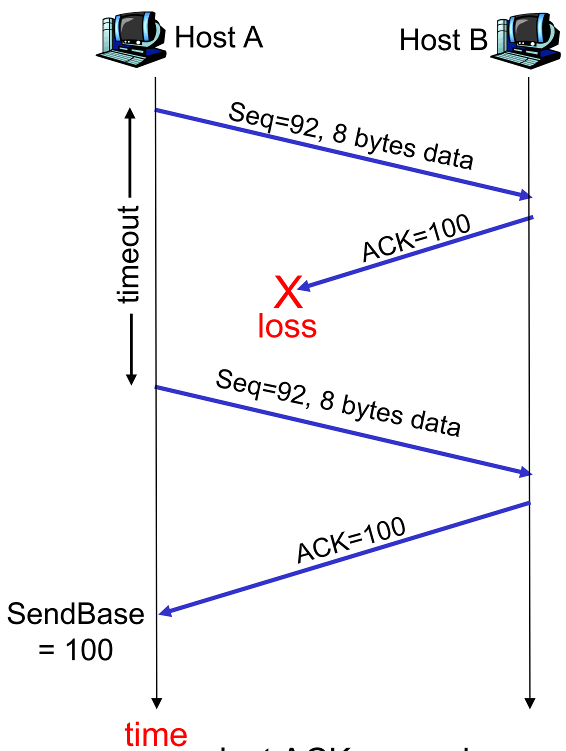
- SendBase-1: last cumulatively ack'ed byte

  (so SendBase is next expected pkt)

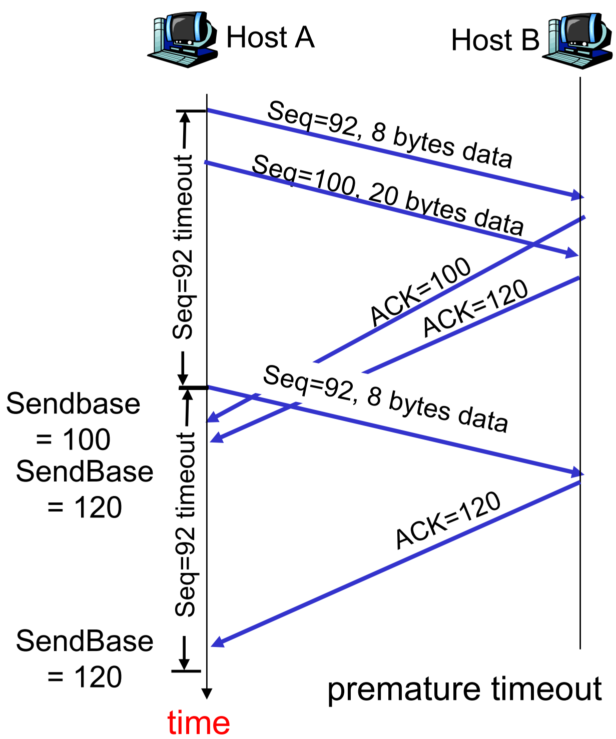
Example:

- SendBase = 72; y= 73, so the rcvr wants 73+ ; y > SendBase, so that new data is acked

TCP: Retransmission Scenarios

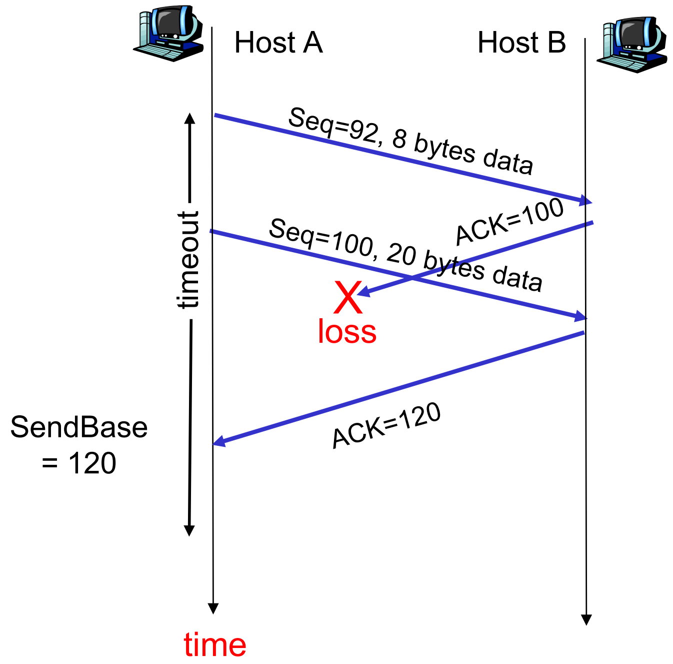


lost ACK scenario



premature timeout

TCP Retransmission Scenarios (more)



Cumulative ACK scenario

TCP ACK Generation [RFC 1122, RFC 2581]

Event at Receiver	TCP Receiver action
Arrival of in-order segment with expected seq #. All data up to expected seq # 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 #. One other segment has ACK pending	Immediately send single cumulative ACK, ACKing both in-order segments
Arrival of out-of-order segment higher-than-expect seq. # . Gap detected	Immediately send duplicate ACK, indicating seq. # 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

Fast Retransmit

- Time-out period often relatively long:
  - Long delay before resending lost packet
- Detect lost segments via duplicate ACKs.
  - Sender often sends many segments back-to-back
  - If segment is lost, there will likely be many duplicate ACKs.
- If sender receives 3 ACKs for the same data, it supposes that segment after ACKed data was lost:
  - Fast retransmit: resend segment before timer expires

Fast Retransmit Algorithm:

```
event: ACK received, with ACK field value of y
            if (y > SendBase) {
                 SendBase = y
                 if (there are currently not-yet-acknowledged segments)
                      start timer
              }
            else {
                 increment count of dup ACKs received for y
                 if (count of dup ACKs received for y = 3) {
                      resend segment with sequence number y
                 }
```

a duplicate ACK for already ACKed segment

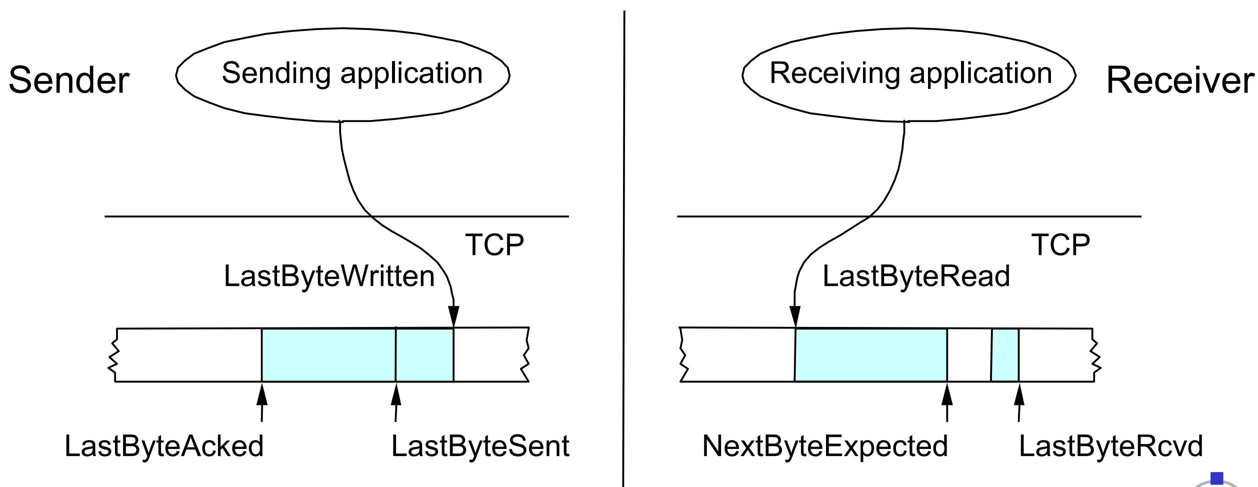
fast retransmit

Chapter Overview

- Transport Layer Services and Protocols
- Addressing and Multiplexing
- Connection Control
- Flow Control
- Congestion Control
- Transport protocols in the Internet:
  - User Datagram Protocol (UDP)
  - Transport Control Protocol (TCP)
    - Connection Management
    - Reliable Data Transfer
    - **Flow Control**
    - Congestion Control
    - Performance

Send and Receive Buffers in TCP

- TCP maintains buffer at:
  - Sender, to service for error control
  - Receiver, to store packets not yet retrieved by application or received out of order
    - Old TCP implementations used GoBack-N, and discarded out-of-order packets



TCP Flow Control: Advertised Window

- In TCP, receiver can ***advertise*** size of its receiving buffer
  - Buffer space occupied:
    (NextByteExpected-1) – LastByteRead
  - Maximum buffer space available: MaxRcvdBuffer
  - Advertised buffer space ***(Advertised window):***
    MaxRcvdBuffer – ((NextByteExpected-1) – LastByteRead)
- Recall: Advertised window limits the amount of data that a sender will inject into the network
  - TCP sender ensures that:
    LastByteSent – LastByteAcked $\leq$ AdvertisedWindow
  - Equivalently:
    EffectiveWindow = AdvertisedWindow – (LastByteSent - LastByteAcked)

Nagle's Algorithm – Self-Clocking and Windows

- TCP self-clocking: Arrival of an ACK is an indication that new data can be injected into the network (see also later)
- What happens when an ACK for only small amount of data (e.g., 1 byte arrives)?
  - Send immediately? Network will be burdened by small packets *("silly window syndrome")*
- Nagle's algorithm describes ***how much*** data TCP is allowed to send
  - When application produces data to send
    if both available data and advertised window $\geq$ MSS
      send a full segment
    else
      if there is unacked data in flight, buffer new data until MSS is full
      else send all the new data now

Chapter Overview

- Transport Layer Services and Protocols
- Addressing and Multiplexing
- Connection Control
- Flow Control
- Congestion Control
- Transport protocols in the Internet:
  - User Datagram Protocol (UDP)
  - Transport Control Protocol (TCP)
    - Connection Management
    - Reliable Data Transfer
    - Flow Control
    - **Congestion Control**
    - Performance

Congestion Control in TCP

- TCP's mechanism for congestion control
  - Implicit feedback by dropped packets
    - Whether the packets were dropped because queues were full or by a mechanism like RED is indistinguishable (and immaterial) to TCP
    - There are some proposals for explicit router feedback as well, but not part of original TCP
    - Assumption: Congestion is the only important source of packet drops!
  - Window-based congestion control:
    - I.e., TCP keeps track of how many bytes it is allowed to inject into the network as a window that grows and shrinks
    - Sender limits transmission (in addition to limit due to flow control):

LastByteSent - LastByteAcked ≤ CongWin

Note: in the following discussion the flow control window will be ignored

TCP ACK/Self-Clocking

- Suppose TCP has somehow determined a correct size of its congestion window
  - Suppose also that the TCP source has injected this entire amount of data into the network but still has more data to send
- When to send more data?
  - Only acceptable when there is space in the network again
  - Space is available when packets leave the network
  - Sender can learn about packets leaving the network by receiving an acknowledgement!
- Thus: ACK not only serves as a confirmation, but also as a ***permit*** to inject a corresponding amount of data into the network

! ***ACK-clocking*** (***self-clocking***) behavior of TCP

Good and Bad News

- Good news: ACK arrival
  - Network could cope with the currently offered load; it did not drop the packet
  - Let's be greedy and try to offer a bit more load – and see if it works

    ⇒ Increase congestion window

- Bad news: No ACK, timeout occurs
  - Packet has been dropped, network is overloaded
  - Put less load onto the network

    ⇒ Reduce congestion window

Reduce Congestion Window by How Much?

- Overloaded network is bad situation – quick and drastic response necessary

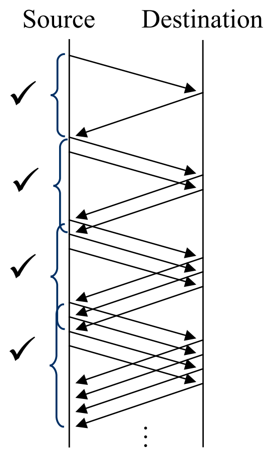
⇒ Upon timeout, cut congestion window in half
  - Reduce load by 50%
  - A minimum congestion window of one packet is always allowed

- A ***multiplicative decrease***

- If a packet happens to be dropped because of a transmission error (not due to overload), TCP misinterprets and overreacts
  - But this is a rare occurrence in wired networks
  - Leads to various problems in wireless networks

Increase Congestion Window by How Much?

- When increasing congestion window, sender cannot be sure that additional capacity is actually available
  - Asymmetric situation to decreasing of congestion window!
- Hence: Be careful, only increase a little!
  - Think in term of round trip times (RTT)
  - If all packets sent out within the last RTT arrived, try to send one more packet per RTT
    - There's a little bit of rounding up involved to account for packet generation times



- This adds constant amounts of load: ***additive increase***

Additive Increase – Details

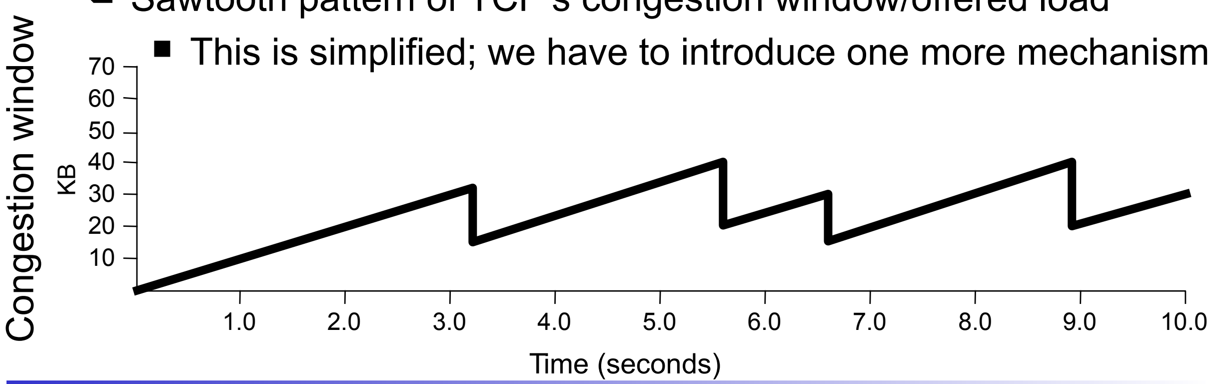
- Additive increase does not wait for a full RTT before it adds additional load
- Instead, each arriving ACK is used to add a little more load (not a full packet)

- Specifically:
  - Increment = MSS x (MSS / Congestion Window)
  - Congestion Window += Increment

  - Where MSS is the Maximum Segment Size, the size of the largest allowed packet

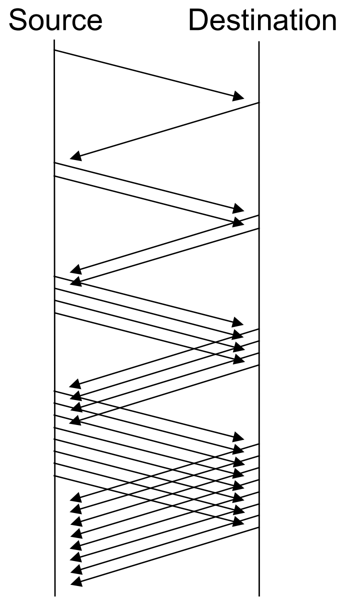
AIMD – Sawtooth Pattern of TCP's Offered Load

- In summary: TCP uses an ***additive increase multiplicative decrease*** (***AIMD***) scheme
- Consequence
  - A TCP connection perpetually ***probes*** the network to check for additional bandwidth
  - Will repeatedly exceed it and fall back, owing to multiplicative decrease
  - Sawtooth pattern of TCP's congestion window/offered load
    - This is simplified; we have to introduce one more mechanism!



Quickly Initialize a Connection: Slow Start

- Additive increase nice and well when operating close to network capacity
- But takes a ***long*** time to converge to it for a new connection
  - Starting at congestion window of, say, 1 or 2
- Idea: Quickly ramp up the congestion window in such an initialization phase
  - One option: ***double congestion window*** each RTT
  - Equivalently: add one packet per ACK
  - Instead of just adding a single packet per RTT



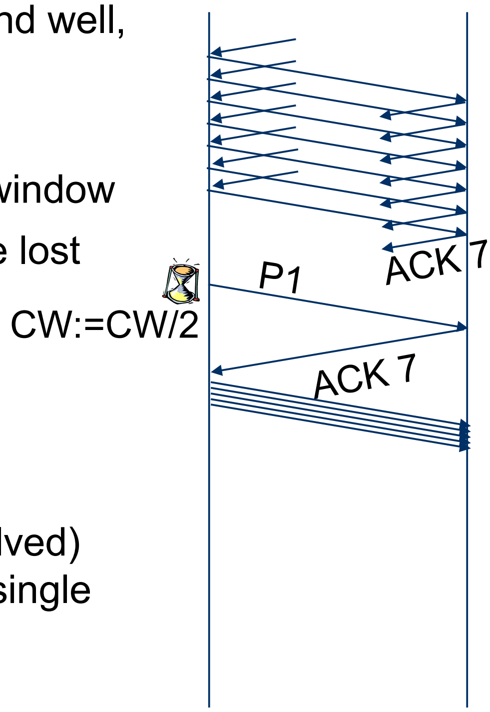
Name "slow start" is historic – it was slow compared to some earlier, too aggressive scheme

Leaving Slow Start

- When doubling congestion window, network capacity will eventually be exceeded
- Packet loss and timeout will result
- Congestion window is halved and TCP switches to "normal", linear increase of congestion window
- The "***congestion avoidance***" phase

Remaining Problem: Packet Bursts

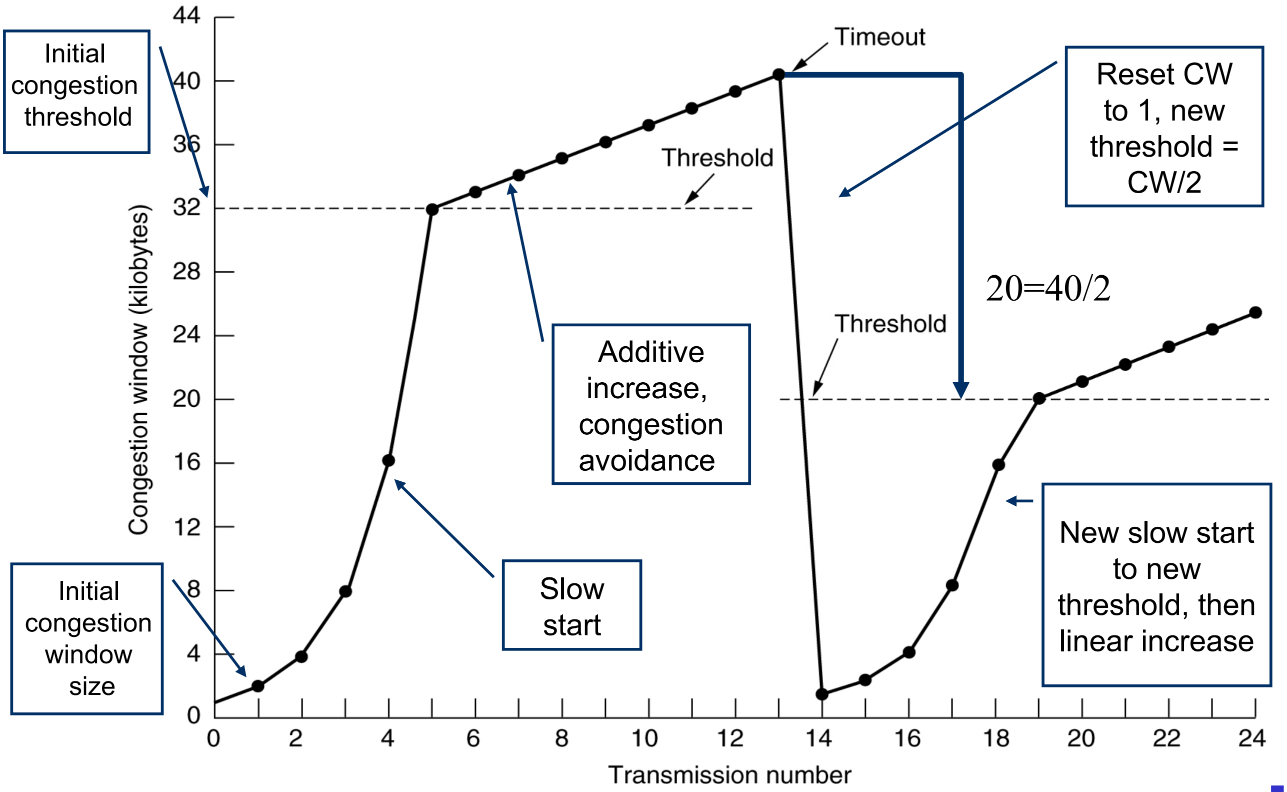
- Congestion control scheme so far: Nice and well, but one problematic case remains
- Suppose
  - A sender transmits its full congestion window
  - Packets arrive, acknowledgements are lost
  - Timeout occurs, CW is halved
  - One packet is retransmitted
  - Cumulative acknowledgement for all outstanding packets arrives
  - ⇒ Sender will then transmit an entire (halved) congestion window worth of data in a single burst! *ACK clocking is missing*!
  - ⇒ Not good! Many packet losses!



Solution: Use Slow Start Here As Well

- Avoiding such packet bursts by linearly increasing CW too slow
  - We can use the slow start mechanism to get the ACK flow going again
  - ⇒ ***Reset the congestion window to 1, restart slow start***
- In addition: we have some rough idea of what the network's capacity is!
  - When initializing a connection, no idea – have to wait for the first packet loss
  - Here: the previous, halved congestion window is a relatively good guess!
  - We can avoid the next packet loss by using the previous congestion window as a ***congestion threshold***
  - ⇒ Use slow start's exponential growth until congestion threshold is reached, then switch to additive increase

TCP Congestion Window Dynamics



Summary: TCP Congestion Control

- When **CongWin** is below **Threshold**, sender in slow-start phase, window grows exponentially.
- When **CongWin** is above **Threshold**, sender is in congestion-avoidance phase, window grows linearly.
- When a triple duplicate ACK occurs, **Threshold** set to **CongWin/2** and **CongWin** set to **Threshold**.
- When timeout occurs, **Threshold** set to **CongWin/2** and **CongWin** is set to 1 MSS.

Summary: TCP Sender Congestion Control

Event	State	TCP Sender Action	Commentary
ACK receipt for previously unacked data	Slow Start (SS)	CongWin = CongWin + MSS, If (CongWin > Threshold) set state to "Congestion Avoidance"	Resulting in a doubling of CongWin every RTT
ACK receipt for previously unacked data	Congestion Avoidance (CA)	CongWin = CongWin+MSS * (MSS/CongWin)	Additive increase, resulting in increase of CongWin by 1 MSS every RTT
Loss event detected by triple duplicate ACK	SS or CA	Threshold = CongWin/2, CongWin = Threshold, Set state to "Congestion Avoidance"	Fast recovery, implementing multiplicative decrease. CongWin will not drop below 1 MSS.
Timeout	SS or CA	Threshold = CongWin/2, CongWin = 1 MSS, Set state to "Slow Start"	Enter slow start
Duplicate ACK	SS or CA	Increment duplicate ACK count for segment being acked	CongWin and Threshold not changed

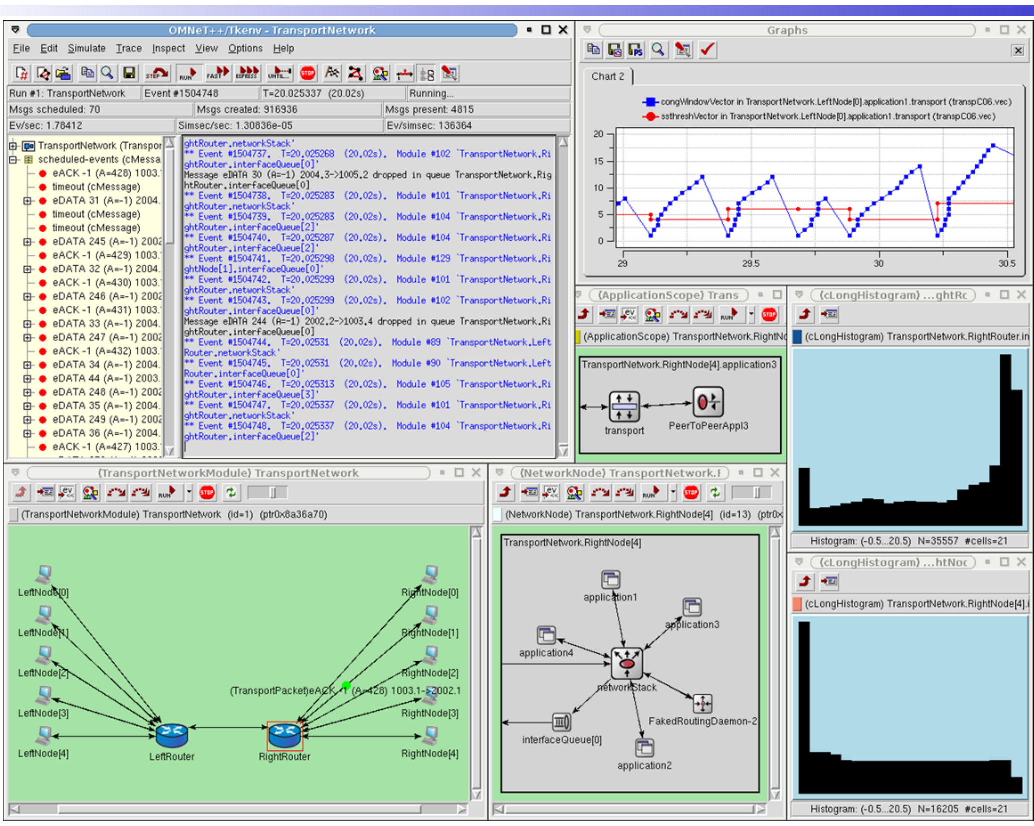
Summary (one more): TCP Congestion Control

- This description still glosses over some (minor) details, but captures the essence
- Extensions to TCP: Fast retransmit, fast recovery
  - Take corrective actions without having to wait for a timeout
  - Necessary for large delay*data rate networks
- Different TCP versions: TCP Tahoe, TCP Reno, TCP Vegas
  - Main difference is the congestion control
  - Correct interoperation is a tricky question (e.g., fairness)
  - Complicated dynamics
- Main source of complications: *Stupidity of the network*

Short Advertisement For Those Who Want More On This...

- There is a specific project seminar *“Simulative Evaluation of Internet Protocol Functions”* on performance evaluation of Internet protocol functions
- It is designed to give you a “hands-on” experience with network protocol functions and simulation studies:
  - Introduces a simulation environment and lets you add protocol functionality
  - Studied protocol functions: forwarding, routing, (interface queues), connection setup, error-, flow- and congestion control
  - Requires good programming skills
  - Knowledge of C++ is an asset (but not a pre-requisite)
  - Allows you to obtain in-depth knowledge of topics covered in Telematics I and the techniques and art of simulation studies – because afterwards “you did it!” :o)

Example: Evaluation of TCP Congestion Control



Chapter Overview

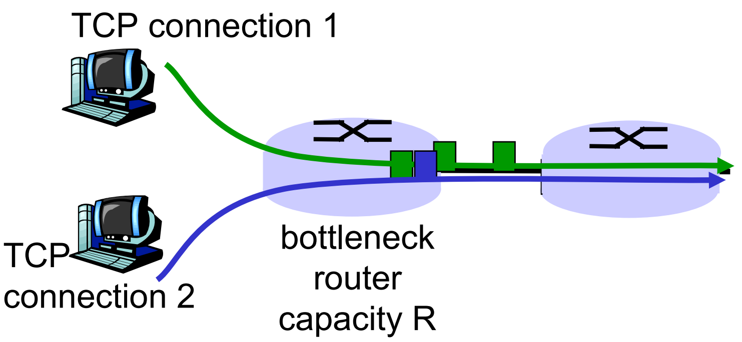
- Transport Layer Services and Protocols
- Addressing and Multiplexing
- Connection Control
- Flow Control
- Congestion Control
- Transport protocols in the Internet:
  - User Datagram Protocol (UDP)
  - Transport Control Protocol (TCP)
    - Connection Management
    - Reliable Data Transfer
    - Flow Control
    - Congestion Control
    - **Performance**

TCP Throughput

- What's the average throughout of TCP as a function of window size and RTT?
  - For the sake of simplicity, let us ignore slow start
- Let W be the window size when loss occurs.
- When window is W, throughput is W/RTT
- Just after loss, window drops to W/2, throughput to W/2RTT.
- Average throughput: .75 W/RTT

TCP Fairness

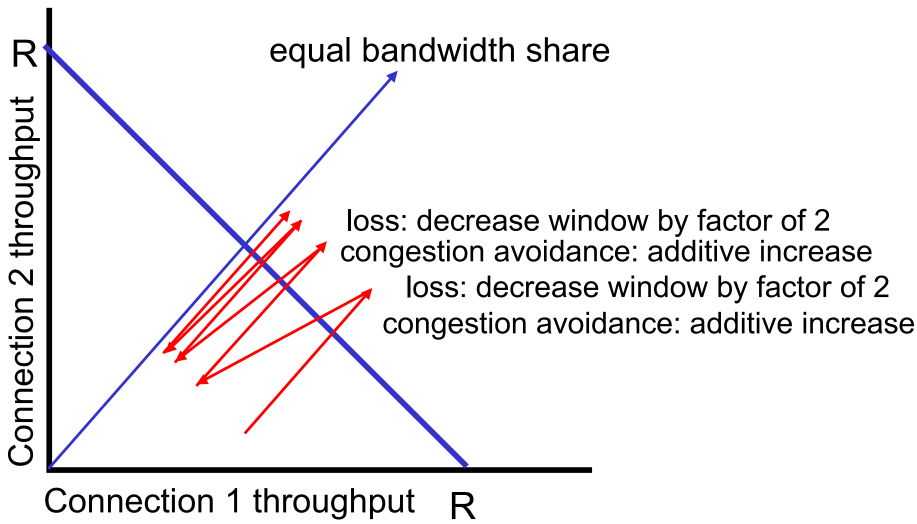
Fairness goal: If K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K



Why is TCP fair?

Two competing sessions:

- Additive increase gives slope of 1, as throughout increases
- Multiplicative decrease decreases throughput proportionally



Fairness (more)

Fairness and UDP

- Multimedia apps often do not use TCP:
  - Do not want rate throttled by congestion control
- Instead use UDP:
  - Pump audio/video at constant rate, tolerate packet loss
- Research area: TCP friendly

Fairness and parallel TCP connections

- Nothing prevents applications from opening parallel connections between 2 hosts.
- Web browsers do this
- Example: link of rate R supporting 9 connections;
  - New application asks for 1 TCP, gets rate R/10
  - New application asks for 9 TCPs, gets R/2 !

Delay Modeling

Question:

- How long does it take to receive an object from a Web server after sending a request?
- Ignoring congestion, delay is influenced by:
  - TCP connection establishment
  - Data transmission delay
  - Slow start

Notations & Assumptions:

- Assume one link between client and server of rate R
- S: MSS (max. segment size, bits)
- O: object size (bits)
- No retransmissions (no loss, no corruption)

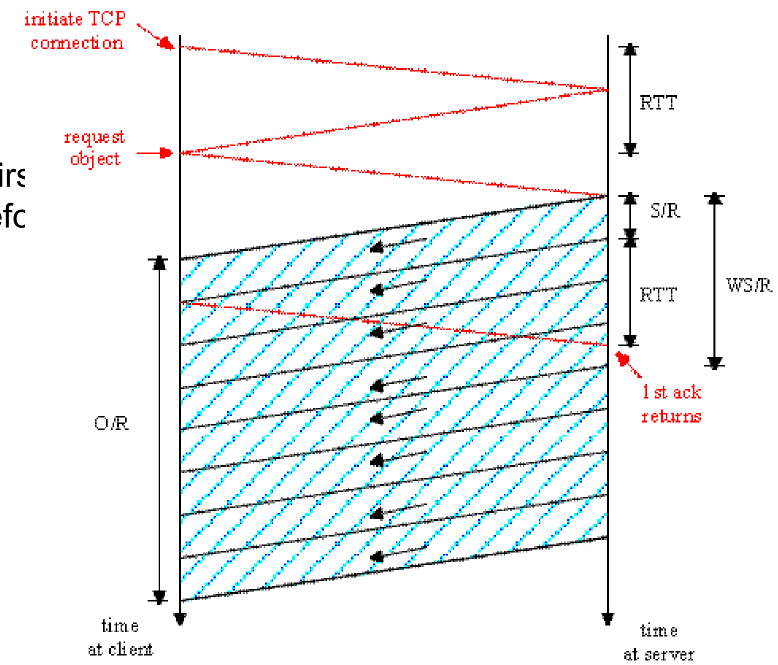
Window size:

- First assume: fixed congestion window, W segments
- Then dynamic window, modeling slow start

Fixed Congestion Window (1)

First case:

WS/R > RTT + S/R: ACK for firs segment in window returns befc window's worth of data sent

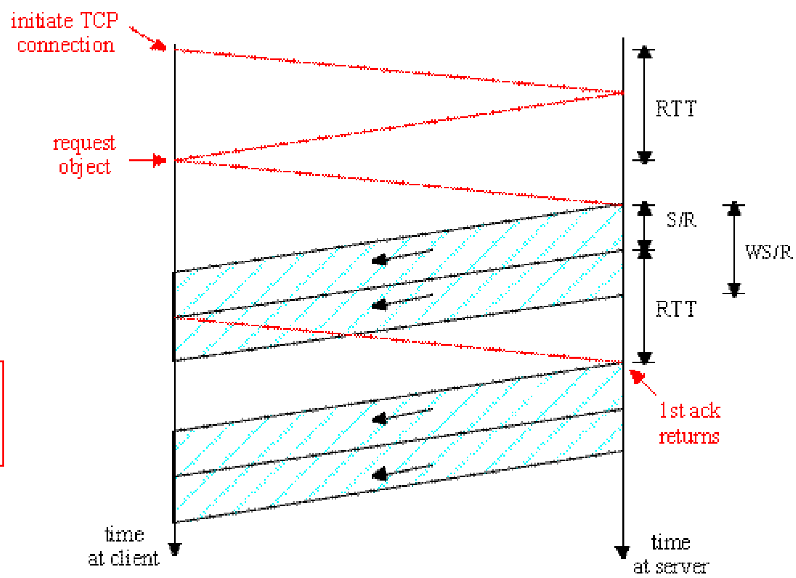


delay = 2RTT + O/R

Fixed Congestion Window (2)

Second case:

- WS/R < RTT + S/R: wait for ACK after sending window's worth of data sent



delay = 2RTT + O/R + (K-1)[S/R + RTT - WS/R]

TCP Delay Modeling: Slow Start (1)

Now suppose window grows according to slow start

We will show that the delay for one object is:

$$\boxed{Latency = 2RTT + \frac{O}{R} + P\left[RTT + \frac{S}{R}\right] - (2^P - 1)\frac{S}{R}}$$

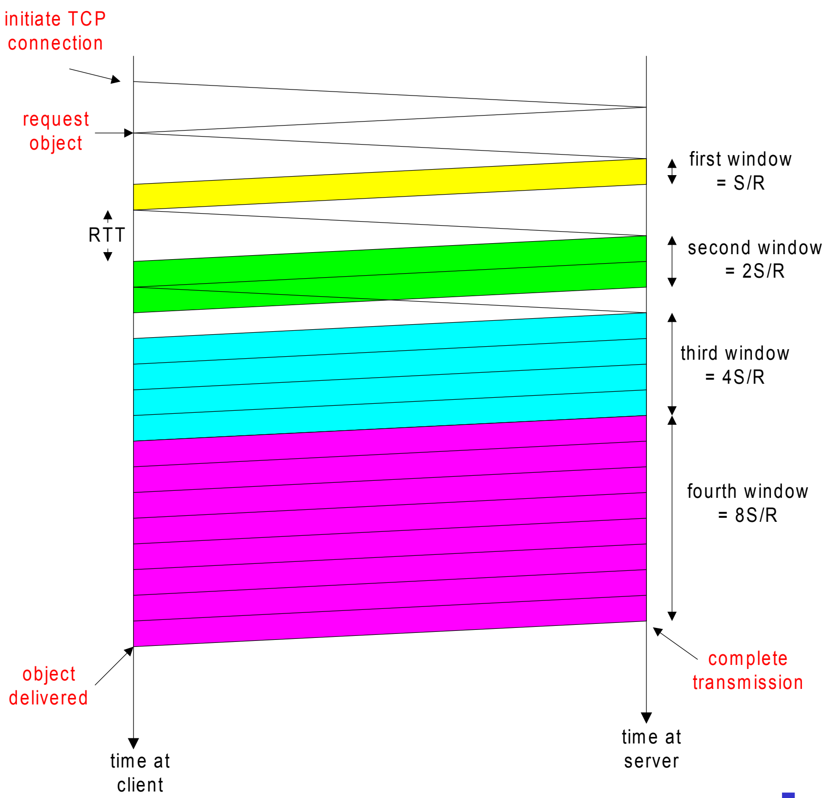
where *P* is the number of times TCP idles at server:

$$P = \min\{Q, K-1\}$$

- where Q is the number of times the server idles if the object were of infinite size.

- and K is the number of windows that cover the object.

TCP Delay Modeling: Slow Start (2)



Delay components:

- 2 RTT for connection estab. and request
- O/R to transmit object
- time server idles due to slow start
- Server idles: P = min{K-1,Q} times

Example:

- O/S = 15 segments
- K = 4 windows
- Q = 2
- P = min{K-1,Q} = 2
- Server idles P=2 times

TCP Delay Modeling (3)

$$\frac{S}{R}+RTT=\text{time from when server starts to send segment until server receives acknowledgement}$$

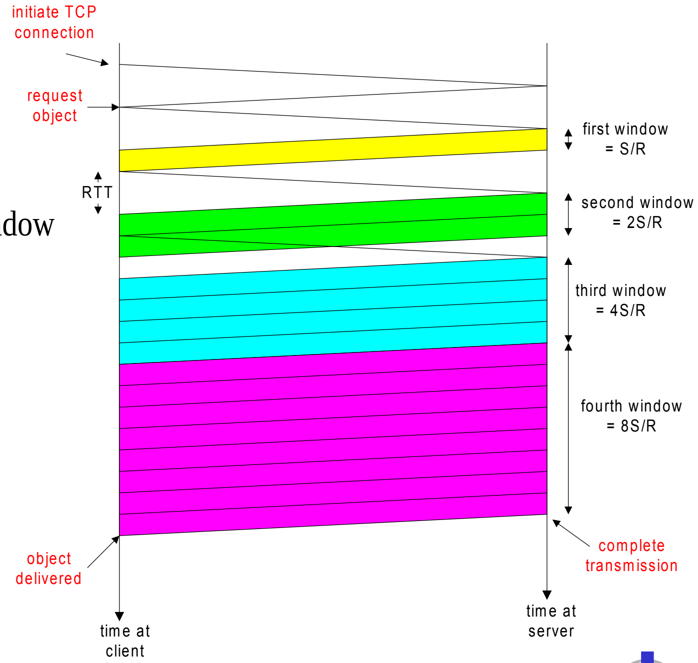
$$2^{k-1}\frac{S}{R}=\text{time to transmit the kth window}$$

$$\left[\frac{S}{R}+RTT-2^{k-1}\frac{S}{R}\right]^{+}=\text{idle time after the } k \text{ th window}$$

$$\text{delay}=\frac{O}{R}+2\,RTT+\sum_{p=1}^{P} idleTime_{p}$$

$$=\frac{O}{R}+2\,RTT+\sum_{k=1}^{P}\left[\frac{S}{R}+RTT-2^{k-1}\frac{S}{R}\right]$$

$$=\frac{O}{R}+2\,RTT+P\left[RTT+\frac{S}{R}\right]-(2^{P}-1)\frac{S}{R}$$



TCP Delay Modeling (4)

Recall K = number of windows that cover object

How do we calculate K ?

$$K=\min\{k:2^{0}S+2^{1}S+\cdots+2^{k-1}S\geq O\}$$

$$=\min\{k:2^{0}+2^{1}+\cdots+2^{k-1}\geq O/S\}$$

$$=\min\{k:2^{k}-1\geq\frac{O}{S}\}$$

$$=\min\{k:k\geq\log_{2}(\frac{O}{S}+1)\}$$

$$=\lceil\log_{2}(\frac{O}{S}+1)\rceil$$

Calculation of Q, number of idles for infinite-size object, is similar

Chapter Summary

- Principles behind transport layer services:
  - Addressing, multiplexing, demultiplexing
  - Connection control
  - Flow control
  - Congestion control
- Instantiation and implementation in the Internet:
  - UDP
  - TCP
- As we have seen, in TCP three important protocol functions are implemented “altogether” in one sliding window protocol:
  - Error control: by sequence numbers, ACKs & retransmissions
  - Flow control: by looking at acknowledgements and permits (& seqnums)
  - Congestion control: by further slowing down the sender if packets or ACKs get lost (assumption: packets mainly get lost because of congestion!)

Additional References

[KR04] J. F. Kurose & K. W. Ross, Computer Networking: A Top-Down Approach Featuring the Internet, 2004, 3rd edition, Addison Wesley

[Kar04] H. Karl. *Communication Networks Chapter 8: Congestion Control.* course slides, University of Paderborn, Germany, 2004. http://wwwcs.upb.de/cs/ag-karl/teaching/ws0405/vl-rnetze.html

[Sch04] J. Schiller. *Telematik.* Vorlesungsfolien, Freie Universität Berlin, 2004. http://www.inf.fu-berlin.de/inst/ag-tech/teaching/LehreFUSeiten/WS03/19545-V/index.html

[Tan02] A. S. Tanenbaum. *Computer Networks.* 4th edition, Prentice Hall, 2002.