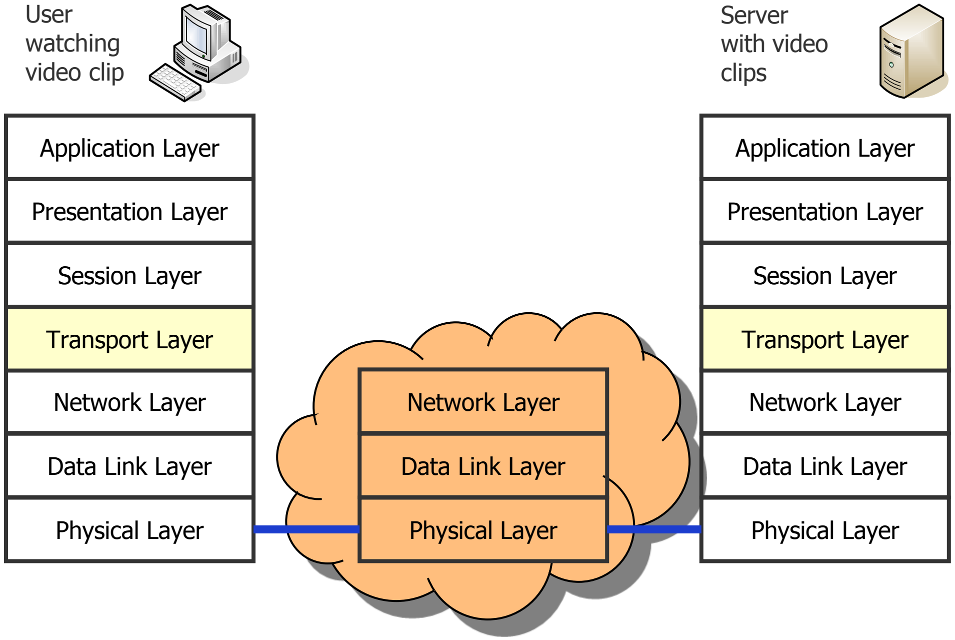
Telematics
Chapter 8: Transport Layer



Univ.-Prof. Dr.-Ing. Jochen H. Schiller

Computer Systems and Telematics (CST)

Institute of Computer Science

Freie Universität Berlin

http://cst.mi.fu-berlin.de

Contents

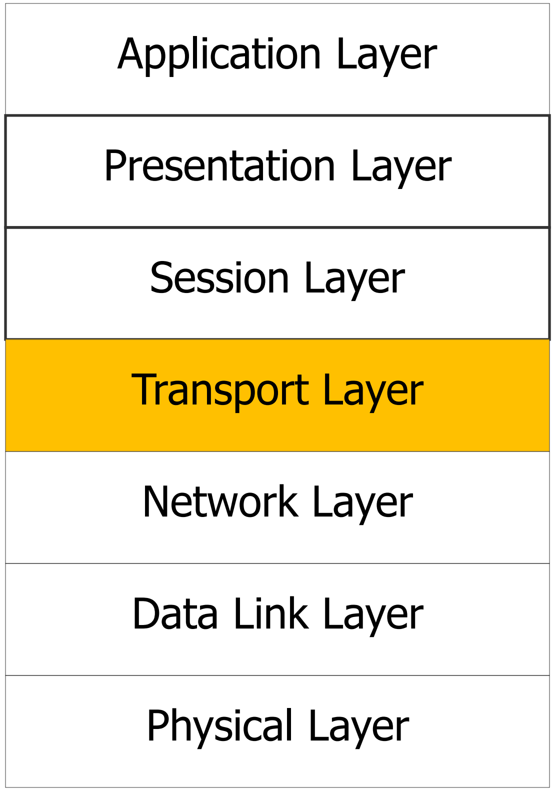
- Design Issues
- User Datagram Protocol (UDP)
- Transmission Control Protocol (TCP)
- An example of socket programming
- New(er) transport protocols
    - SCTP
    - DCCP

Design Issues

Design Issues

- Characteristics of the layers **below** the Transport Layer
  - Available on the hosts and on the routers
  - Operate in a hop-to-hop fashion
  - Typically unreliable
- Characteristics of the Transport Layer
  - Available only on the hosts
  - Operate in a end-to-end fashion
  - It has to operate like a pipe
- Services provided to the upper layers
  - Connection-oriented service
  - Connectionless service
  - Convenient interface for application programmers
    - Berkeley sockets

OSI Reference Model



Application Layer

Presentation Layer

Session Layer

Transport Layer

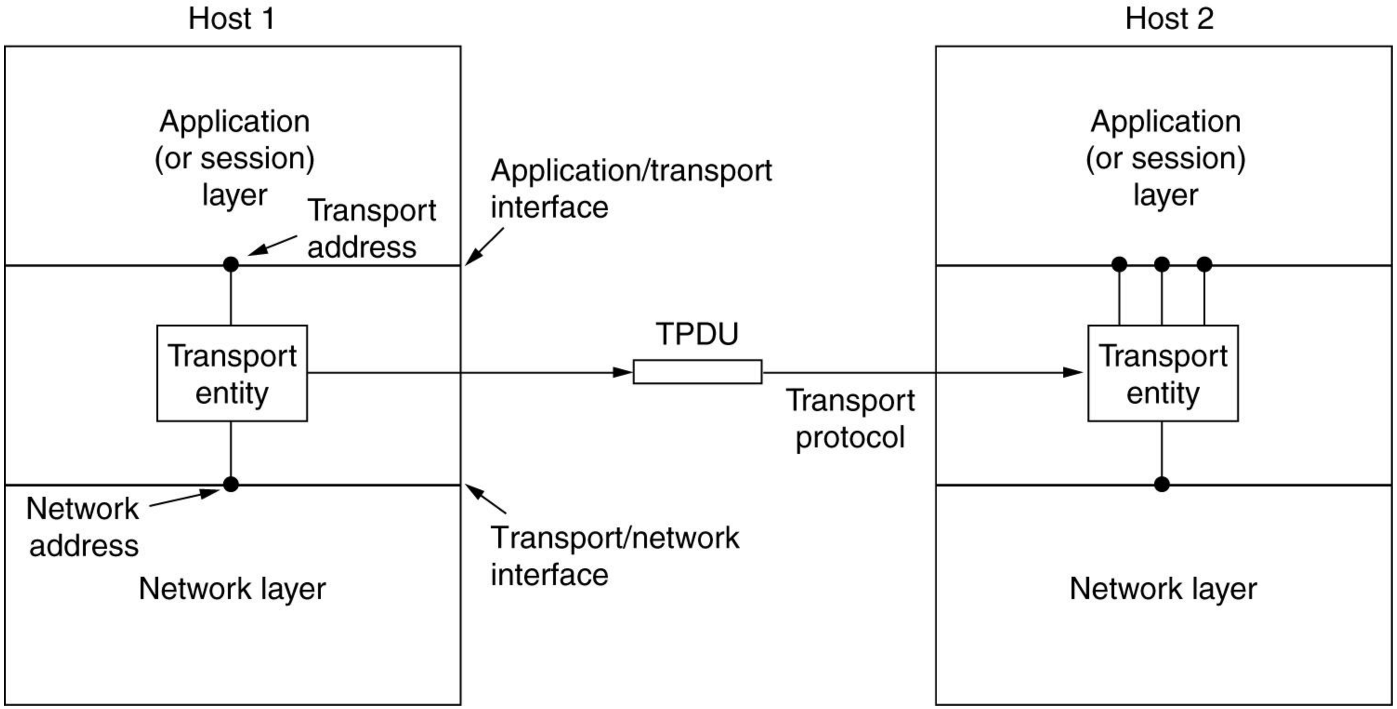
Network Layer

Data Link Layer

Physical Layer

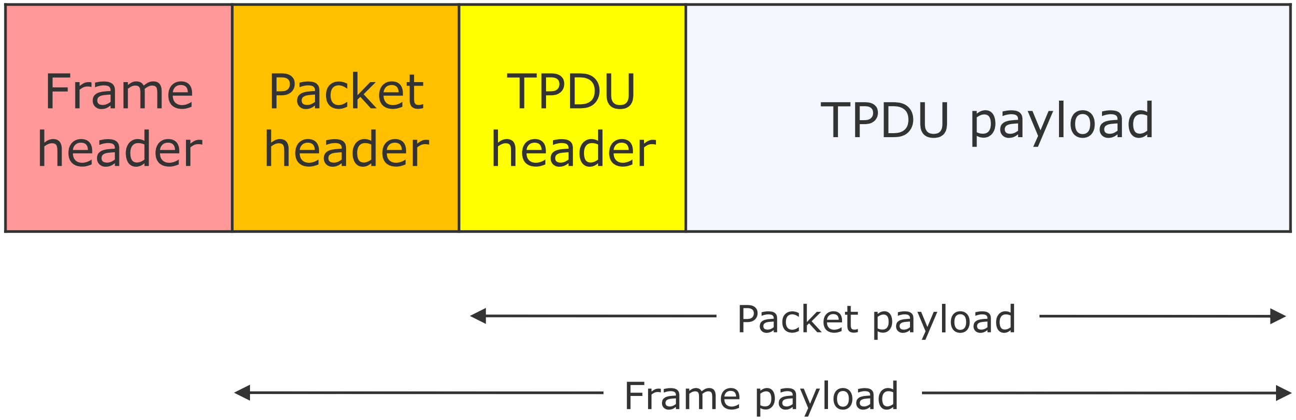
Services Provided to the Upper Layers

- Services provided to the upper layers
  - Goal is to provide an efficient, reliable, and cost-effective service
  - **Transport entity** is responsible to provide that service
    - Maybe located in the operating system kernel, user process, library package, or network interface card



Transport Service Primitives

- Some terminology
  - Messages from transport entities: Transport Protocol Data Unit (TPDU)
  - TPDUs are contained in network layer packets
  - Packets are contained in data link layer frames

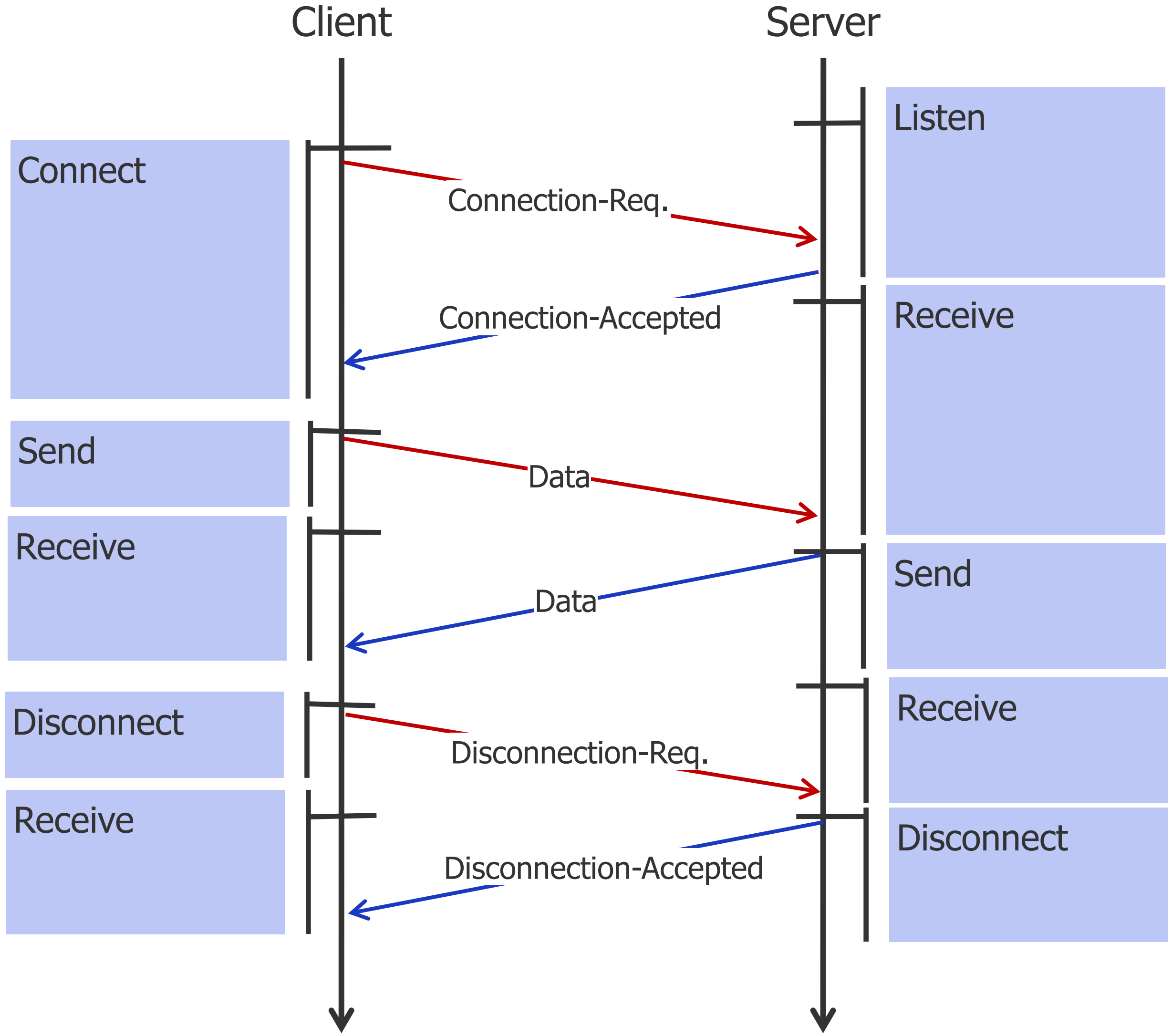


Transport Service Primitives

- Processes on the application layer expect 100% reliable connections
  - They are not interested in acknowledgements, lost packets, congestions, …
- Transport layer provides
  - Unreliable datagram service (Connectionless)
  - Reliable connection-oriented service
    - Three phases: establishment, communication, termination
- The primitives for a simple connection-oriented transport service

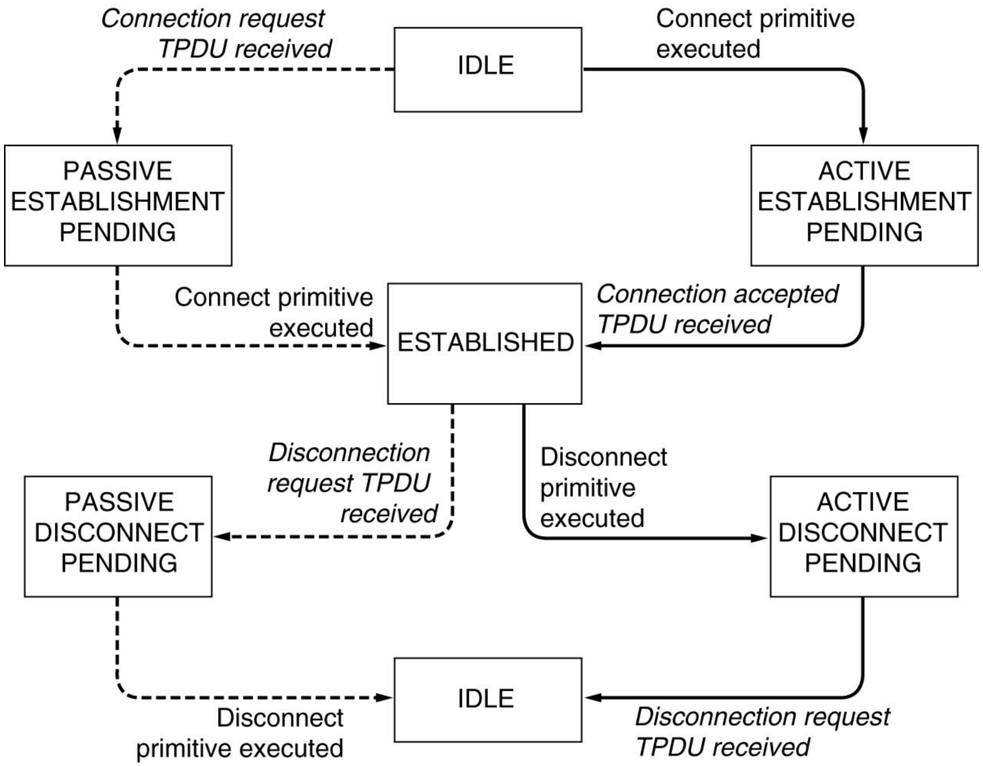
Primitive	Packet sent	Meaning
LISTEN	(none)	Block until some process tries to connect
CONNECT	CONNECTION REQ.	Activeley attempt to establish a connection
SEND	DATA	Send information
RECEIVE	(none)	Block until a DATA packet arrives
DISCONNECT	DISCONNECTION REQ.	This side wants to release the connection

Transport Service Primitives



Transport Service Primitives

- A state diagram for a simple connection management scheme
  - Transitions labeled in italics are caused by packet arrivals
  - The solid lines show the client's state sequence
  - The dashed lines show the server's state sequence

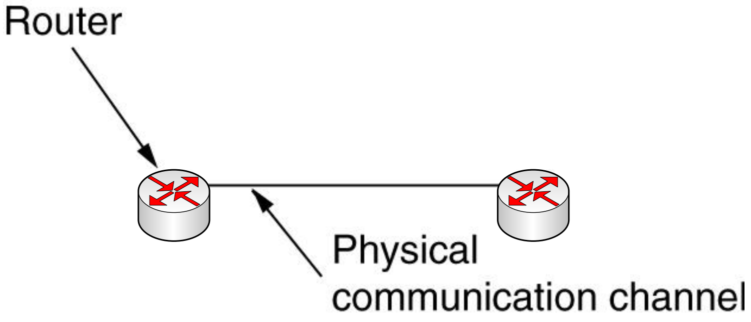

Connection request TPDU received
IDLE
Connect primitive executed
PASSIVE ESTABLISHMENT PENDING
ACTIVE ESTABLISHMENT PENDING
Connect primitive executed
ESTABLISHED
Connection accepted TPDU received
Disconnection request TPDU received
Disconnect primitive executed
PASSIVE DISCONNECT PENDING
ACTIVE DISCONNECT PENDING
IDLE
Disconnect primitive executed
Disconnection request TPDU received

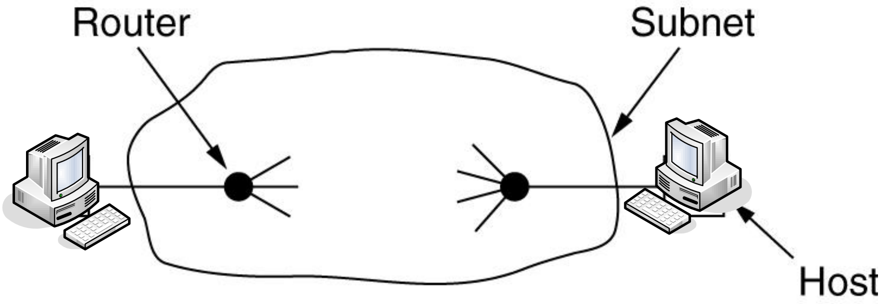
Transport Protocol

Transport protocol

- Transport service is implemented between transport entities by a transport protocol
- Similar to the protocols studied in chapter “Data Link Layer”
    - Have to deal with: error control, sequencing, and flow control

- Environment of the data link layer
    - On DLL two router communicate directly via a physical channel
        - No explicit addressing is required
        - Channel always there
- Environment of the transport layer
    - On TL channel is given by the subnet
        - Explicit addressing of the destination is required
        - Channel is not always there
        - Connection establishment is complicated



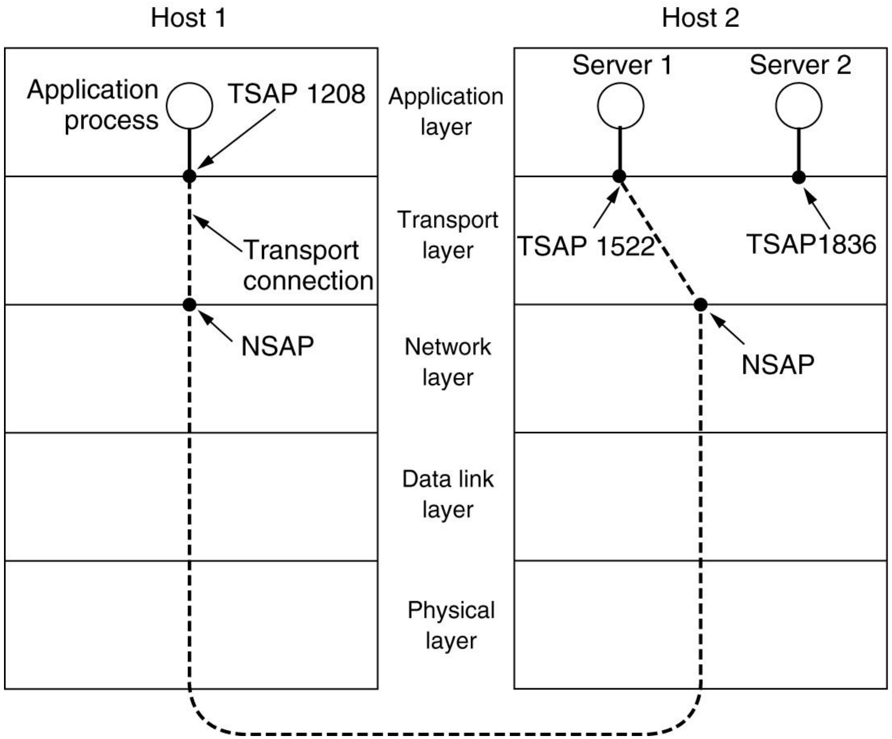


Transport Protocol: Addressing

- Addressing on the transport layer
  - To which process connect to?
  - Transport Service Access Point (TSAP)
  - Network Service Access Point (NSAP)

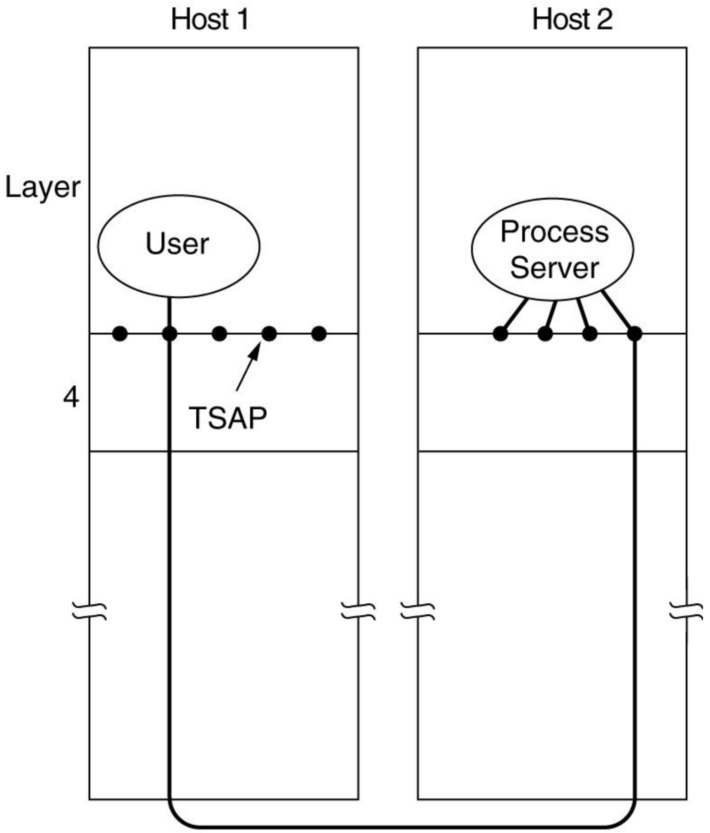
- Questions
  - How does the process on Host 1 know the TSAP of Server 1 on Host 2?

- Solution
  - TSAPs are stored in a specific file:
    - Unix: /etc/services
    - Windows: \system32\drivers\etc\services



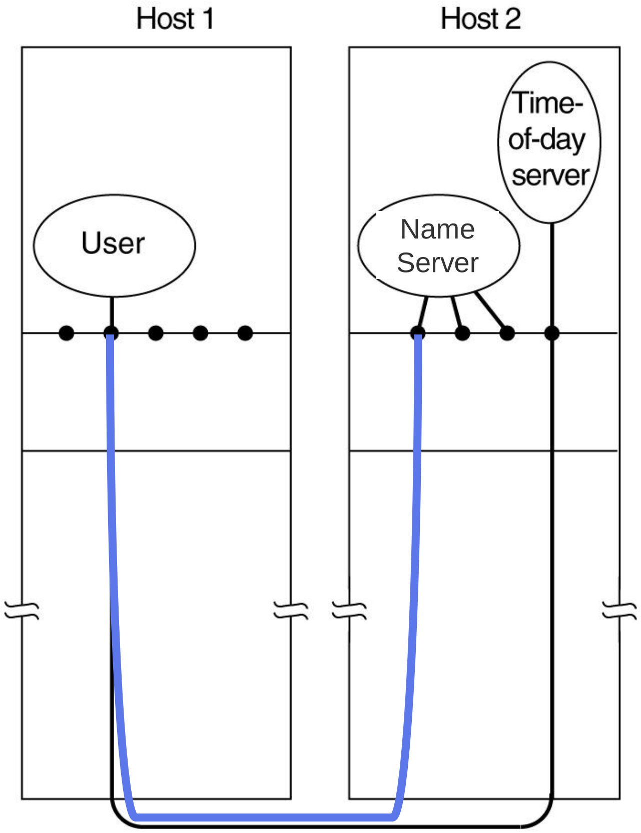
Transport Protocol: Addressing

- Disadvantage of previous solution
  - For small number of servers the solution with specific files works fine.
- Problem
  - When there are many servers, which are rarely used.
- Solution
  - Special process: Process Server
    - Listens to a set of TSAPs
  - When desired server is not active, connection is made with the Process Server
  - Process Servers starts the server for the desired service and passes the connection to it



Transport Protocol: Addressing

- Disadvantage of previous solution
  - What happens if server cannot be started, i.e., service depends on the process?
- Solution
  - Special process: Name Server / Directory Server
  - Client first connects to the Name Server and requests the TSAP of the service
  - Subsequently, connection is established with the desired server
- Requirement
  - Servers have to register with the Name Server
    - Records of (Name, TSAP)

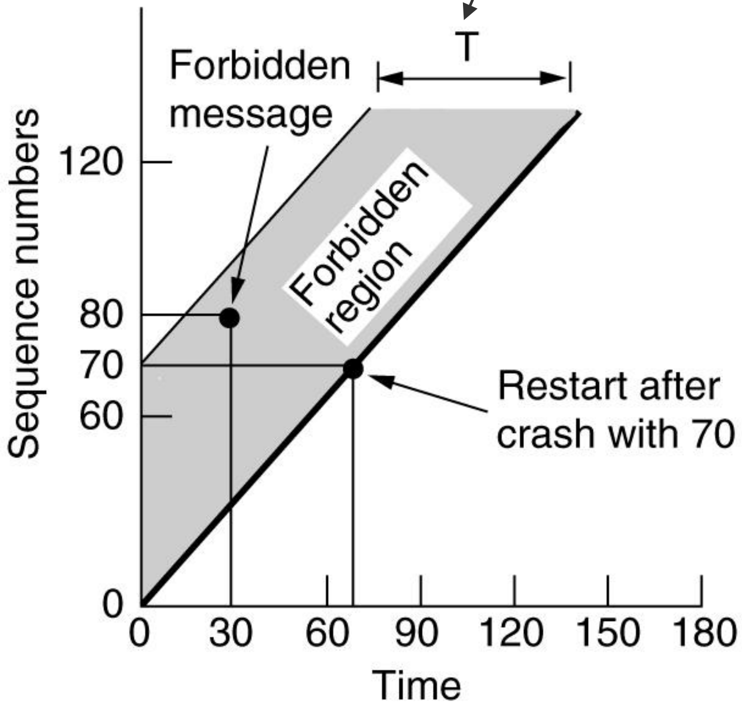


Transport Protocol: Connection Establishment

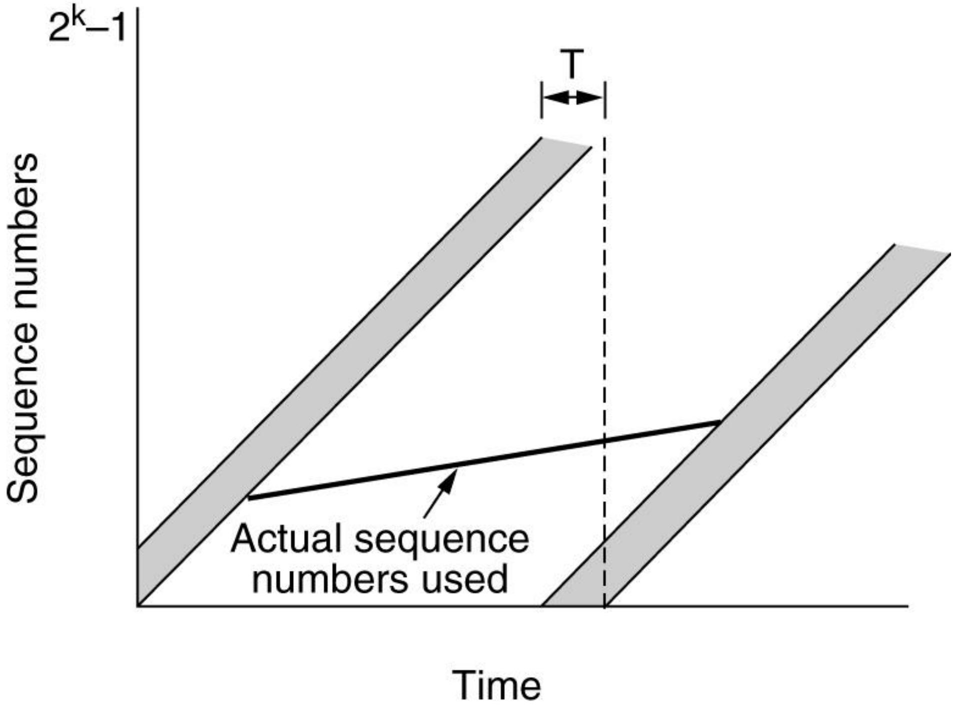
- Problems with connection establishment
  - When network can lose, store, and duplicate packets
  - Delayed duplicates
- Solution approaches
  - Throw-away TSAPs: For each connection a new TSAP is used
    - Management of used TSAPs
  - Restrict packet life time, e.g., by a TTL field, timestamp, …
- Solution of Tomlinson, Sunshine, and Dalal (Three-way Handshake)
  - Each computer has a clock (time of day)
  - Clocks do not need to be synchronized
  - Clock has to run even when the computer crashes or is switched off
  - Idea: Put sequence numbers into TPDUs and two TPDUs with the same sequence number may not outstand at the same time
    - Each connection starts with a different initial sequence number

Transport Protocol: Connection Establishment

Assumption: After T time units TPDU and Ack are dead and have no effect.



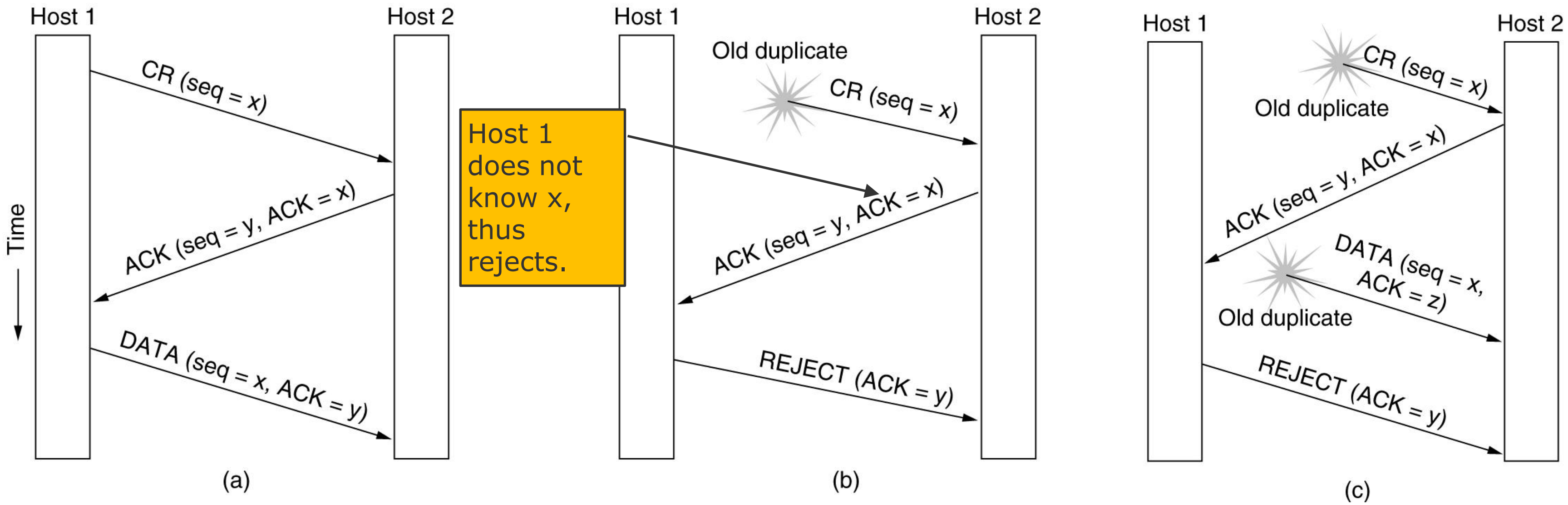
TPDUs may not enter the forbidden region



The resynchronization problem

Transport Protocol: Connection Establishment

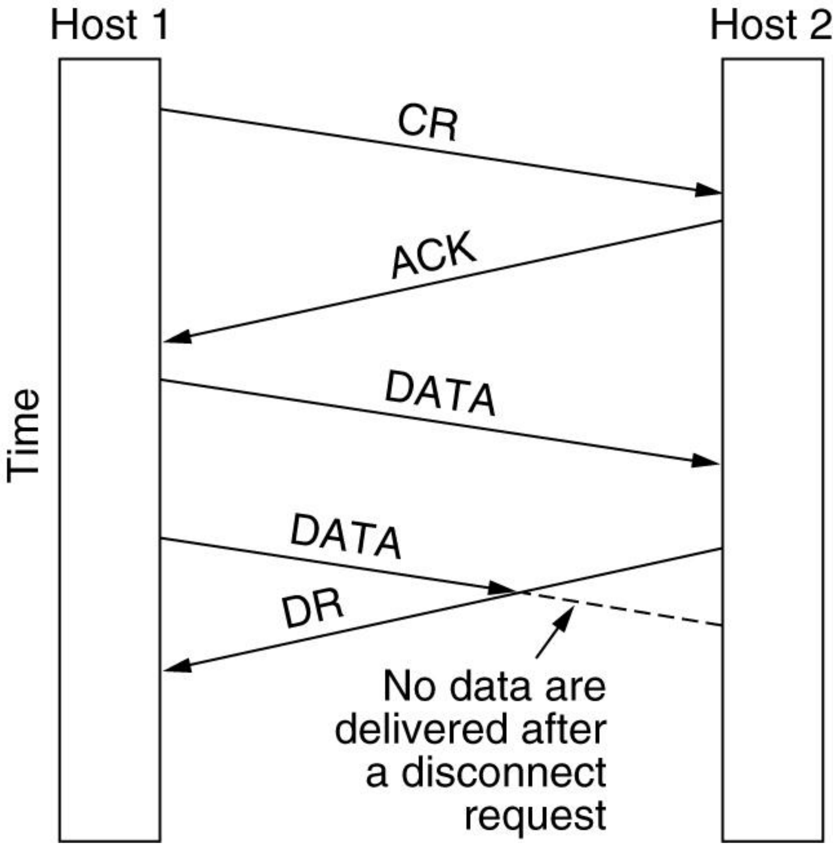
- Connection Establishment with the **Three-way Handshake**
  - Three protocol scenarios for establishing a connection using a three-way handshake. CR denotes CONNECTION REQUEST
    - a) Normal operation
    - b) Old CONNECTION REQUEST appearing out of nowhere
    - c) Duplicate CONNECTION REQUEST and duplicate ACK



Host 1 does not know x, thus rejects.

Transport Protocol: Connection Release

- Terminating a connection
  - Asymmetric release
    - Telephone system model
    - Either one peer can terminate the connection
    - Danger of data loss
  - Symmetric release
    - Model of two independent unicast connections
    - Each peer has to terminate the connection explicitly
    - Data can be received by in the non-terminated direction
- Problem: Data loss can happen on both cases
  - Question: Is there an optimal solution?



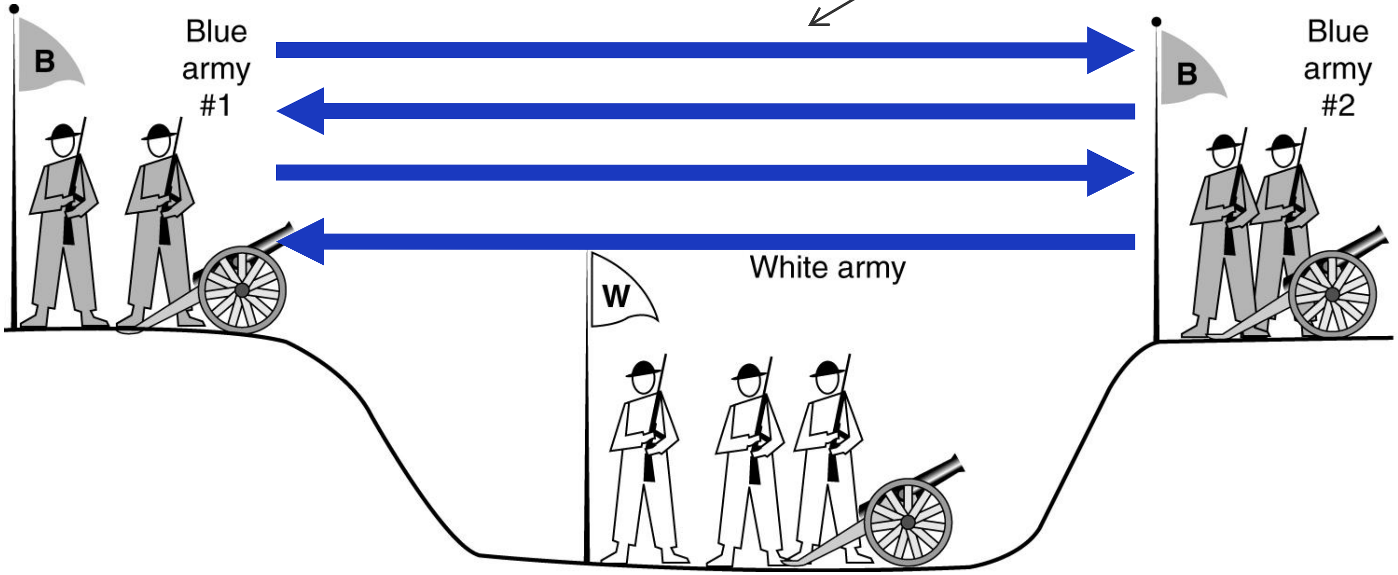
Abrupt disconnection with loss of data.

DR denotes Disconnect Request

Transport Protocol: Connection Release

- Famous example to illustrate the problem of controlled (reliable) connection termination: **The two-army problem**
  - The Blue armies can only communicate with messengers, i.e., soldiers running through the valley
  - Messengers are subject to loss

Messengers of the Blue armies



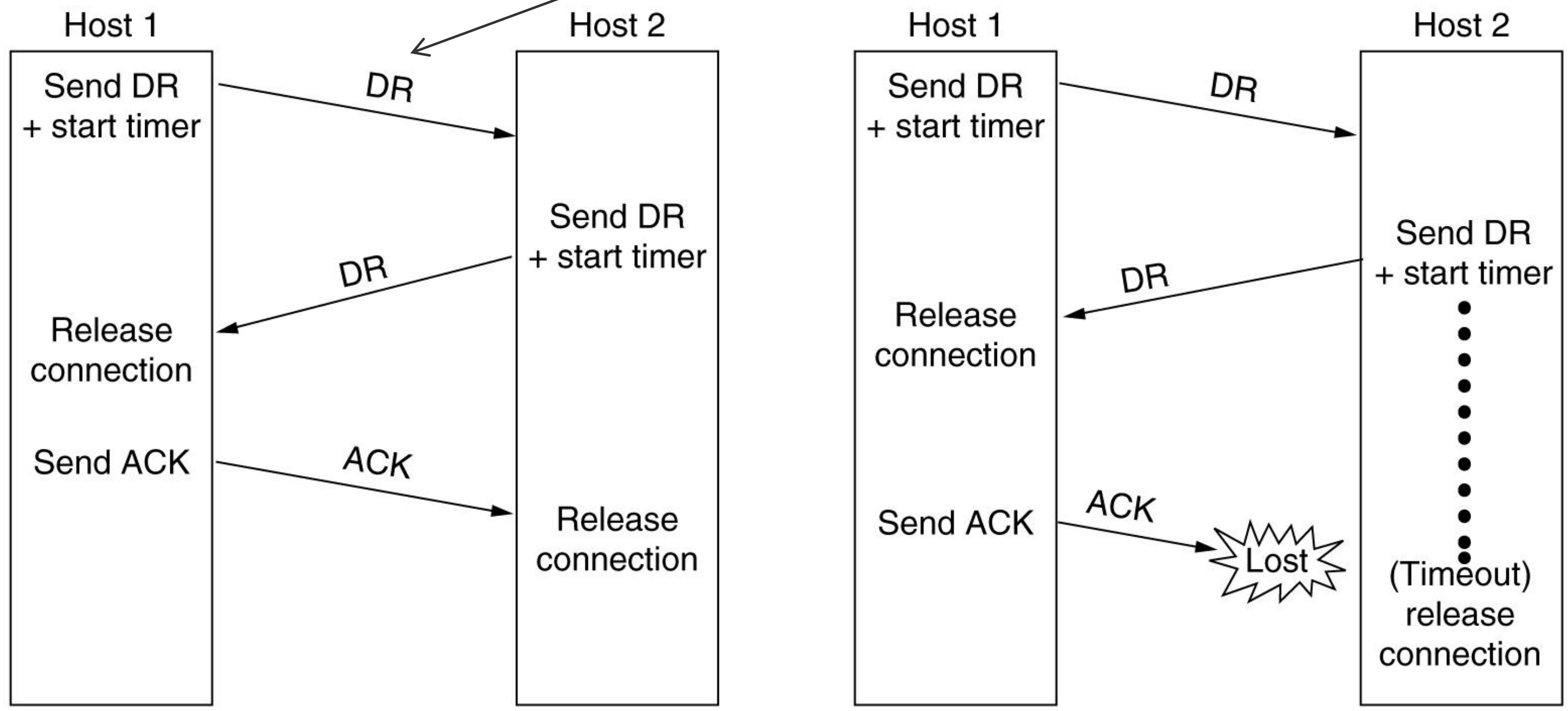
Transport Protocol: Connection Release

Four protocol scenarios for releasing a connection

(a) Normal case of a three-way handshake

(b) final ACK lost

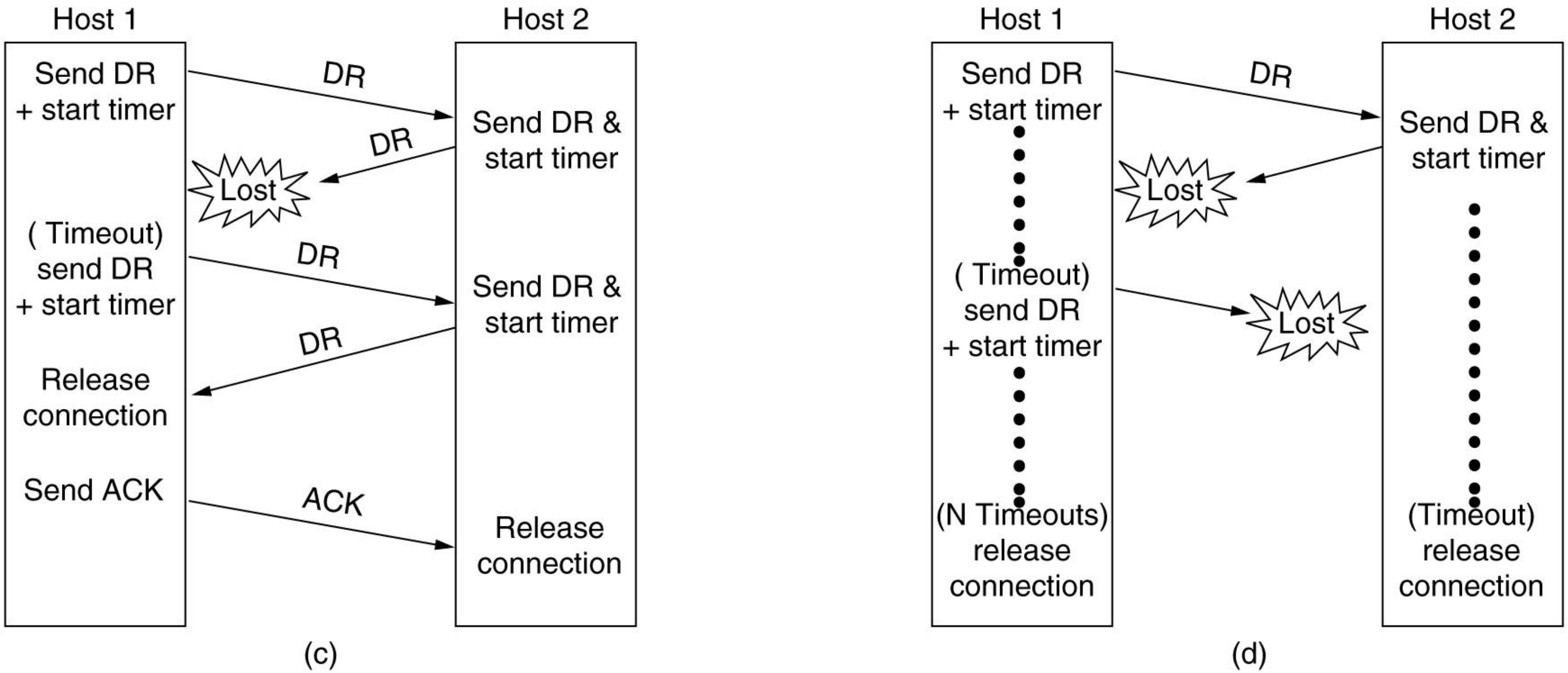
Disconnect Request (DR)



(a) (b)

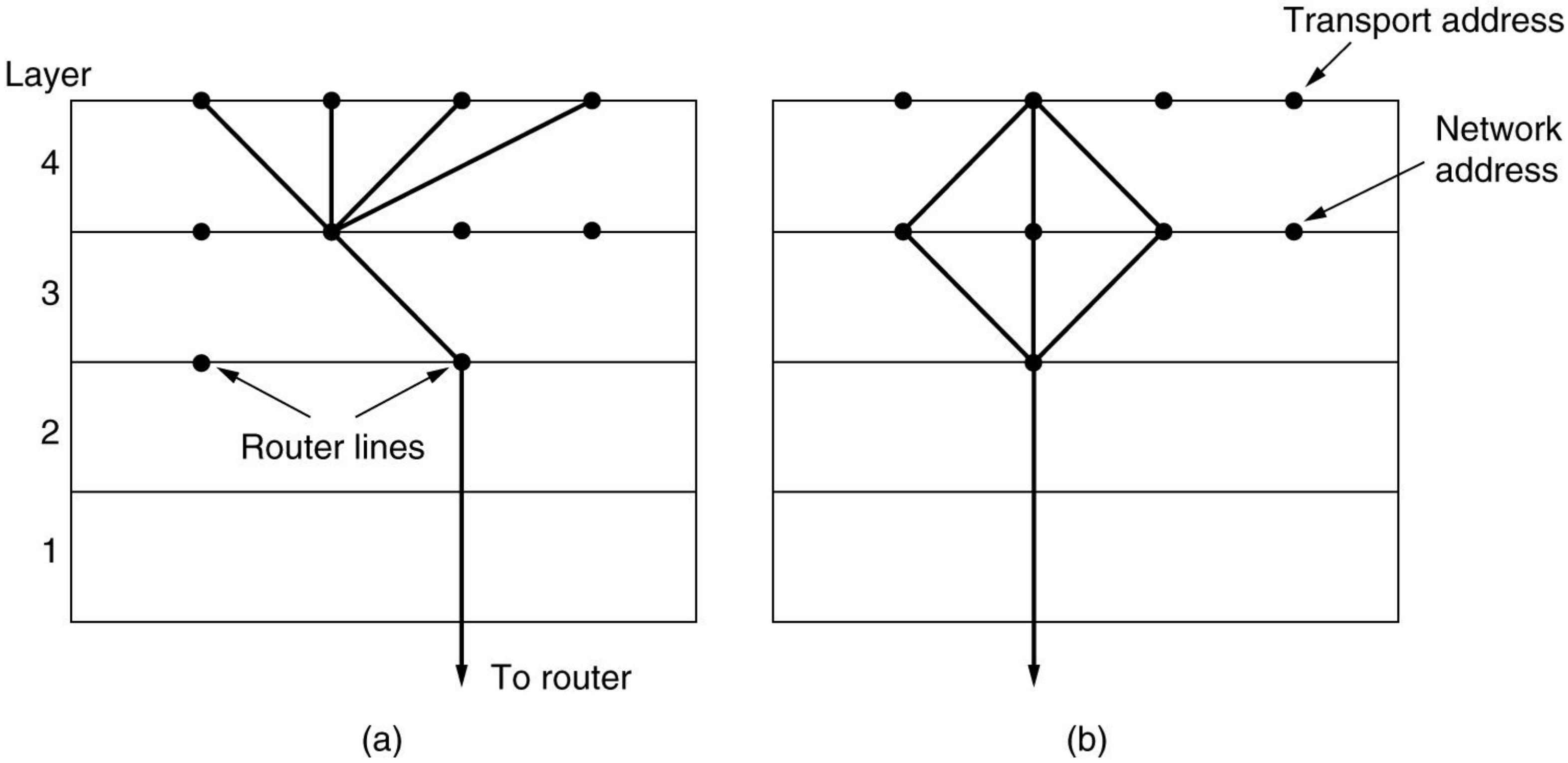
Transport Protocol: Connection Release

- Four protocol scenarios for releasing a connection
  - c) Response lost
  - d) Response lost and subsequent DRs lost



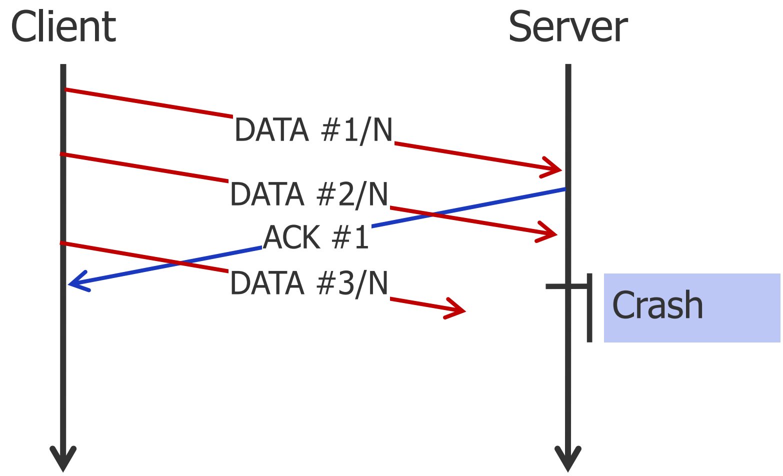
Transport Protocol: Multiplexing

- Multiplexing of several conversations onto connections
  - (a) Upward multiplexing: Many transport connections use the same network address
  - (b) Downward multiplexing: Distribute the traffic of one connection over many network connections



Transport Protocol: Crash Recovery

- If hosts and routers are subject to crashes, recovery becomes an issue
- Scenario
  - A client sends a large file to a server
  - Each chunk of the transmitted file is acked by the server
  - After a crash server does not know the status
- Possible client states
  - S0: No outstanding ack
  - S1: One outstanding ack
- Client strategies
  - Always retransmit last TPDU
  - Never retransmit last TPDU
  - Retransmit last TPDU in S0
  - Retransmit last TPDU in S1



Transport Protocol: Crash Recovery

- Processing strategies of server
  - Strategy 1: First send ack, then write to application
  - Strategy 2: First write to application, then send ack

Crash can occur between the two different operations!

- Different combinations of client and server strategy
  - Server events are {A=Ack, W=Write, C=Crash}

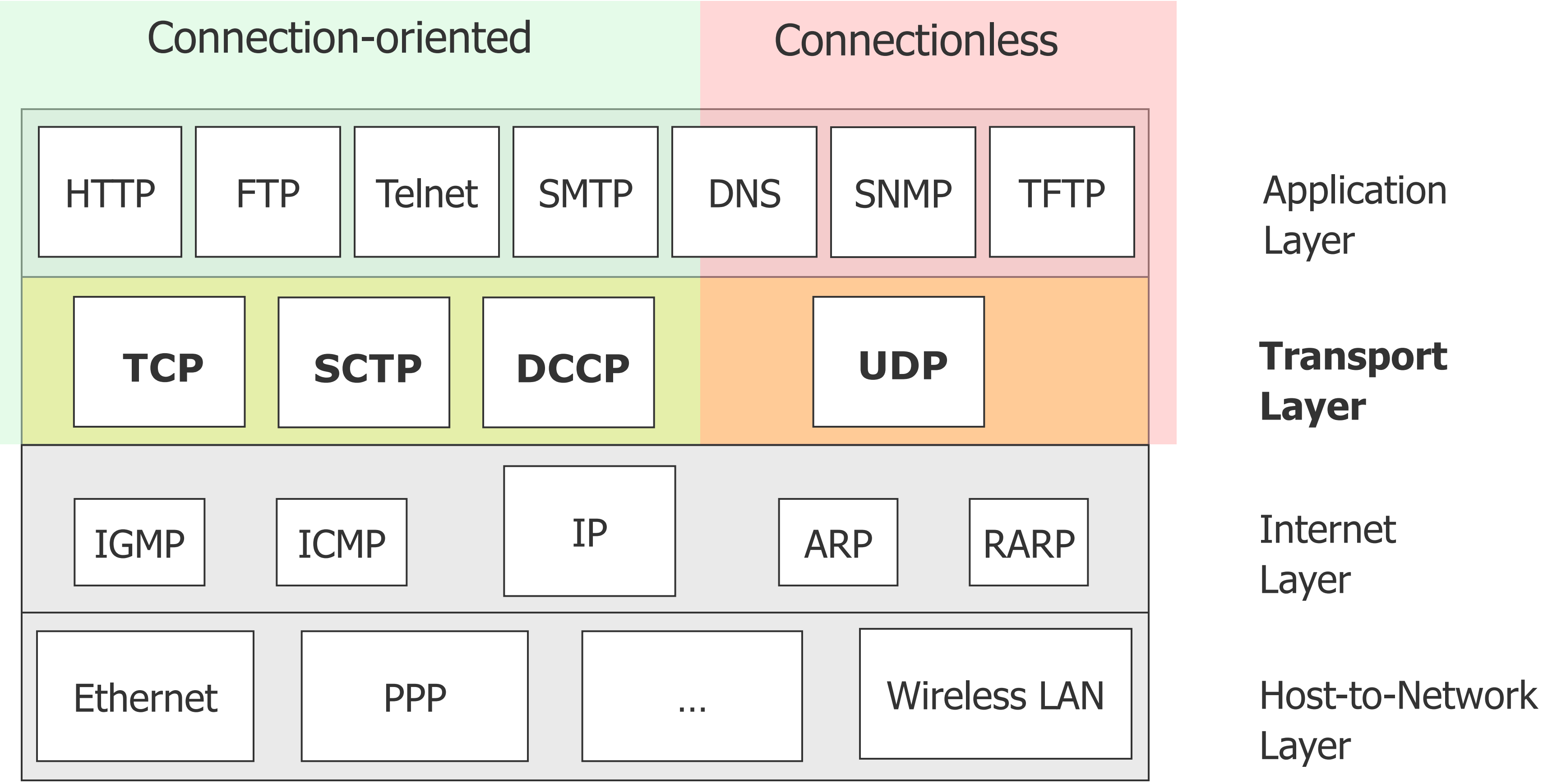
	Strategy used by receiving host					
	First ACK, then write			First write, then ACK		
Strategy used by sending host	AC(W)	AWC	C(AW)	C(WA)	W AC	WC(A)
Always retransmit	OK	DUP	OK	OK	DUP	DUP
Never retransmit	LOST	OK	LOST	LOST	OK	OK
Retransmit in S0	OK	DUP	LOST	LOST	DUP	OK
Retransmit in S1	LOST	OK	OK	OK	OK	DUP

S0: No outstanding ack
S1: One outstanding ack

OK: Protocol functions correctly
DUP: Protocol duplicates message
LOST: Protocol loses a message

Transport Protocols in the TCP/IP Reference Model

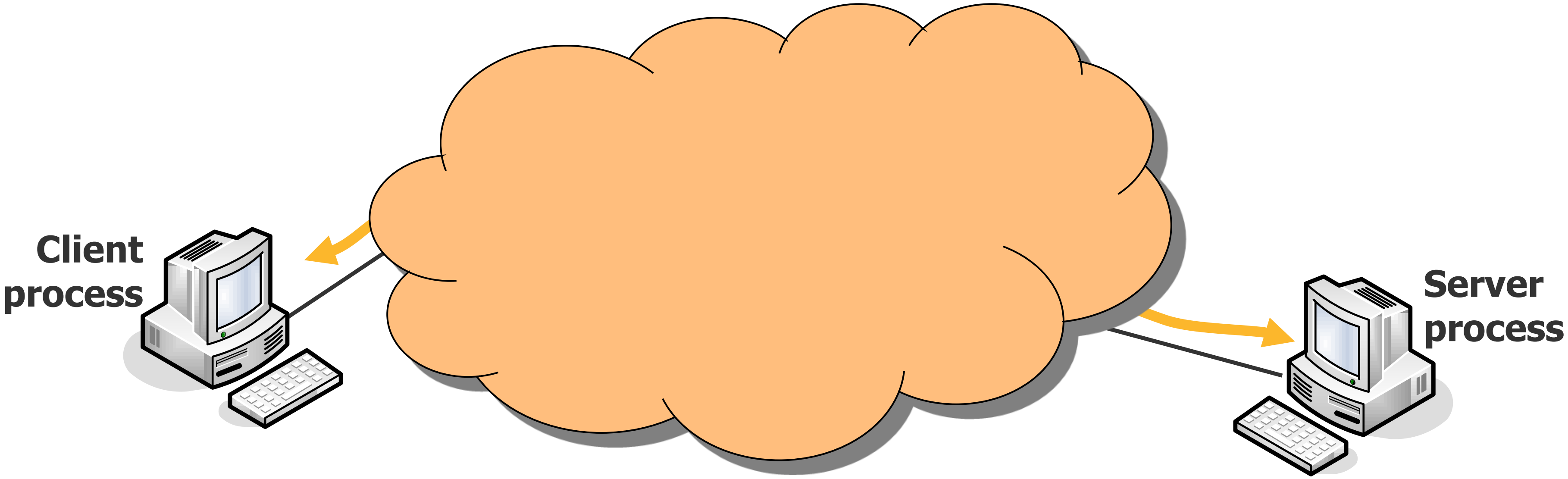
Transport Protocols in the TCP/IP Reference Model



TCP (Transmission Control Protocol): Reliable, connection-oriented
SCTP (Stream Control Transmission Protocol): Reliable, connection-oriented
DCCP (Datagram Congestion Control Protocol): Unrealiable, connection-oriented
UDP (User Datagram Protocol): Unreliable, connectionless

The Classical Transport Layer: TCP and UDP

- Transport protocols are used by the application layer as communication services
  - They allow the communication between application processes
- TCP is a connection-oriented protocol
- UDP is a connectionless



User Datagram Protocol (UDP)

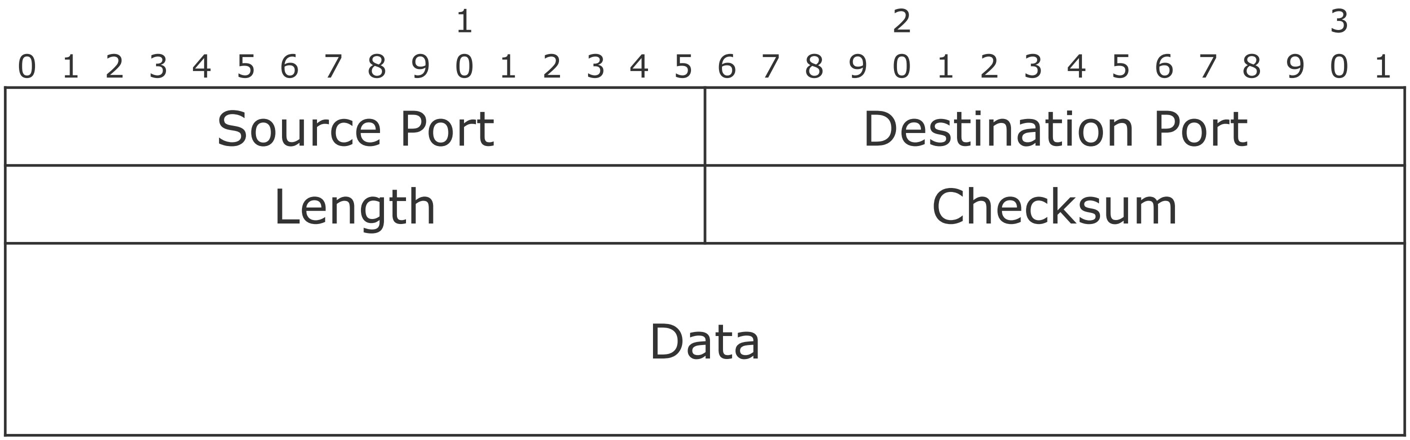
The User Datagram Protocol (UDP)

- Principle: “Keep it simple!”
  - 8 byte header
  - Like IP: connectionless and unreliable
  - Small reliability, but fast exchange of information
  - No acknowledgement between communication peers with UDP
    - Incorrect packets are simply discarded
    - Duplication, sequence order permutation, and packet loss are possible
  - The checksum offers the only possibility of testing the packets on transfer errors
  - Possible: ACKs and retransmissions are controlled by the application
  - Use in multicast (not possible with TCP)
- Why at all UDP?
  - Only the addition of a **port** to a **network address** marks communication unique

(IP Address$_1$, Port$_1$, IP Address$_2$, Port$_2$)

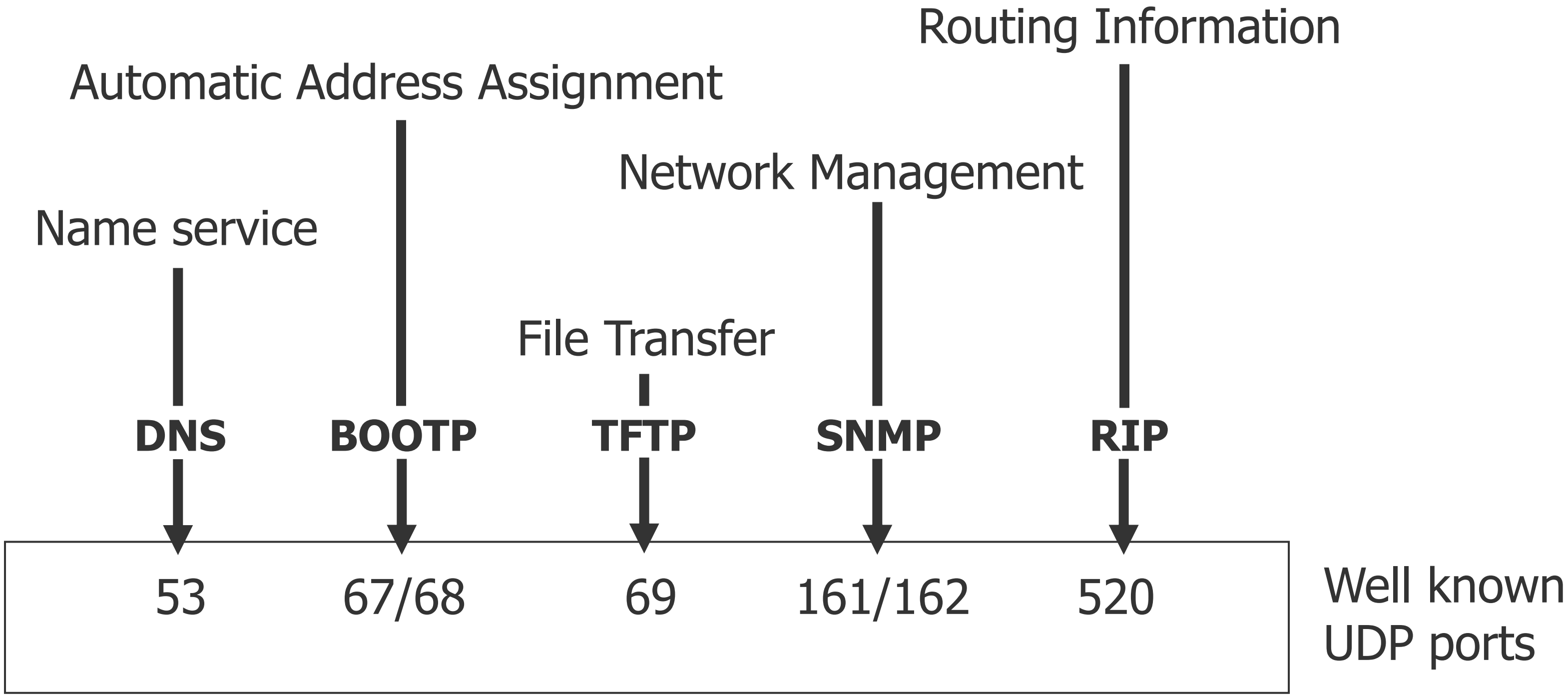
UDP Header

- **Source Port, Destination Port:** Addressing of the applications by port numbers
- **Length:** The total length of the datagram (header + data) in 32-bit words
- **Checksum** (optional)**:** IP does not have a checksum for the data part, therefore it can be a meaningful addition here
  - The same procedure as in TCP
- **Data:** The payload, it is filled up if necessary to an even byte number, since message length counts in 32-bit words



Source Port	Destination Port
Length	Checksum
Data	

UDP-based Applications

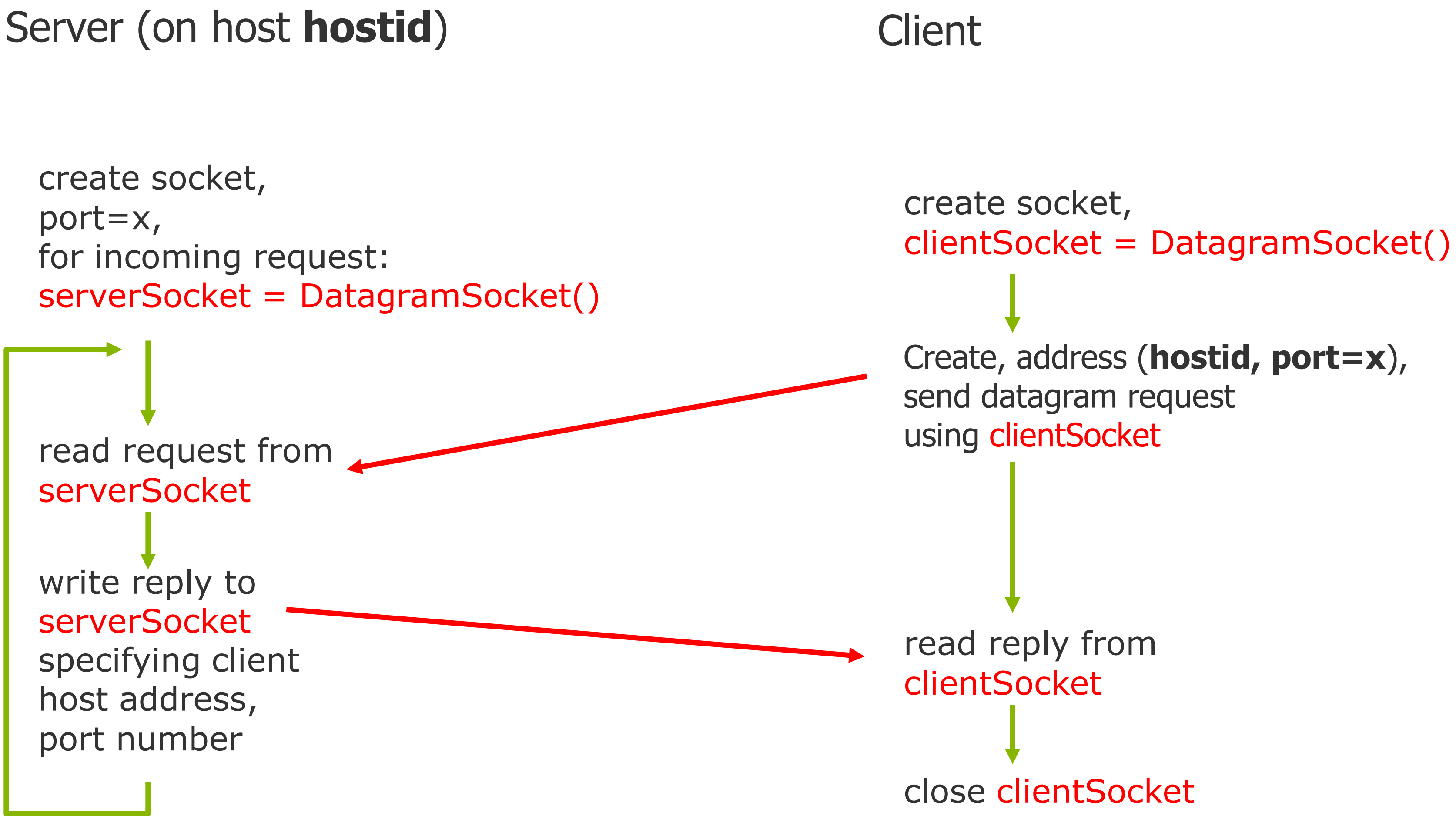


- Port number is 16-bit address, currently three different ranges
  - Ports in the range 0-1023 are **Well-Known** (a.k.a. "system")
  - Ports in the range 1024-49151 are **Registered** (a.k.a. "user")
  - Ports in the range 49152-65535 are **Dynamic/Private**
  - See for more information: http://www.iana.org/assignments/port-numbers

User Datagram Protocol (UDP)

Socket Programming with UDP

Socket Programming with UDP



Server (on host **hostid**)

create socket,
port=x,
for incoming request:
serverSocket = DatagramSocket()

read request from
serverSocket

write reply to
serverSocket
specifying client
host address,
port number

Client

create socket,
clientSocket = DatagramSocket()

Create, address (**hostid, port=x**),
send datagram request
using clientSocket

read reply from
clientSocket

close clientSocket

Example: Java Client (UDP)

```java
import java.io.*;
import java.net.*;

class UDPClient {
    public static void main(String arg []) throws Exception
    {
        BufferedReader inFromUser = new BufferedReader(new InputStreamReader(System.in));
        DatagramSocket clientSocket = new DatagramSocket();
        InetAddress IPAddress = InetAddress.getByName( “hostname” );
        byte[] sendData = new byte[1024];
        byte[] receiveData = new byte[1024];
        String sentence = inFromUser.readLine();
        sendData = sentence.getBytes ();
        DatagramPacket send_pack = new DatagramPacket(sendData, sendData.length,
                                                     IPAddress, 9876);

        clientSocket.send(send_pack);
        DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);
        clientSocket.receive(receivePacket);
        String modifiedSentence = new String(receivePacket.getData());
        System.out.println( “FROM SERVER:”  + modifiedSentence);
        clientSocket.close();
    }
}
```

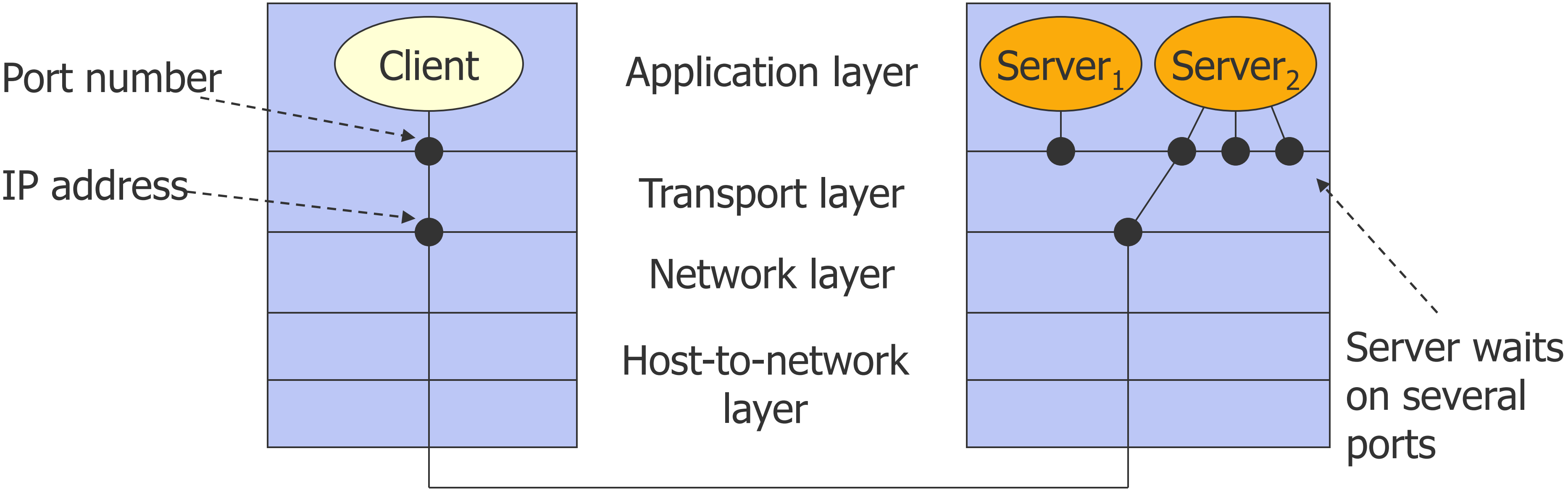
Example: Java Server (UDP)

```
import java.io.*;
import java.net.*;

class UDPServer {
  public static void main(String args[]) throws Exception
  {

    DatagramSocket serverSocket = new DatagramSocket(9876);

    byte[] receiveData = new byte[1024];
    byte[] sendData = new byte[1024];

    while (true)
    {
      DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);
      serverSocket.receive(receivePacket);
      String sentence = new String(receivePacket.getData());
      InetAddress IPAddress = receivePacket.getAddress();
      int port = receivePacket.getPort();
      String capitalizedSentence = sentence.toUpperCase();
      sendData = capitalizedSentence.getBytes();
      DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length,
                                                      IPAddress, port);
      serverSocket.send(sendPacket);
    }
  }
}
```

Transmission Control Protocol (TCP)

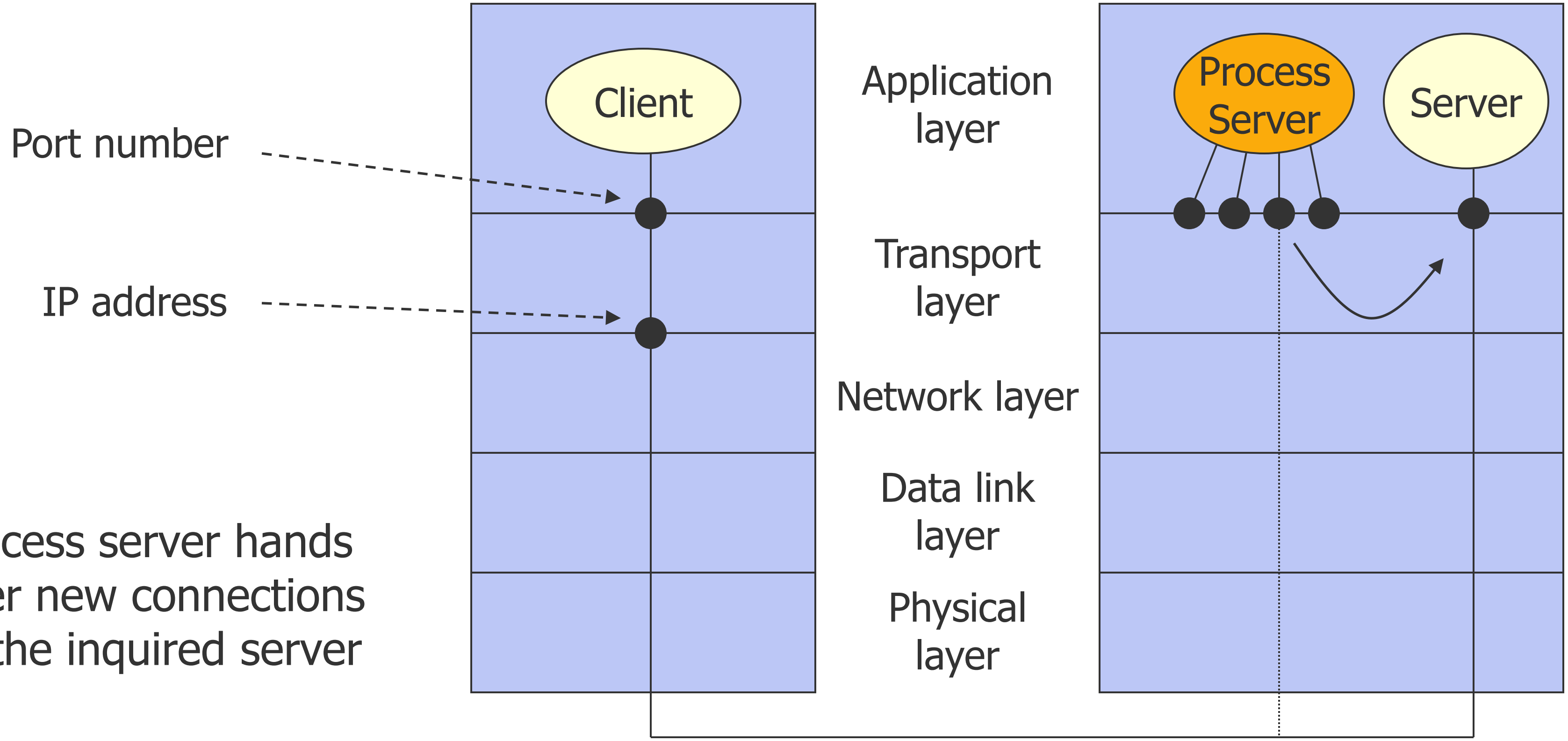
Characteristics of TCP

- Connection-oriented and reliable (error-free, keeps packet order, without duplicates)
- Error handling, acknowledgements, flow control (Sliding Window procedure)
- **Byte stream**, not message stream, i.e., message boundaries are not preserved
- Segmentation (max. segment size of 64 KByte)
- "Urgent"-messages outside of flow control
- Limited QoS
- Addressing of the application by port numbers
  - Port numbers below 1024 are called **well-known ports**, these are reserved for standard services



TCP as a Reliable Connection

If the server port is unknown, the use of a process server (Initial Connection Protocol) is possible:



Process server hands over new connections to the inquired server

Alternatively: Name server (comparable to a phone book) returns the destination port

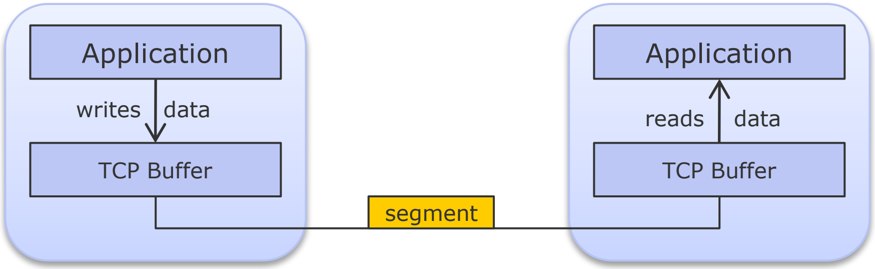
TCP as a Reliable Connection

- Establishes logical connections between two Sockets
  - IP address + 16 bit port number (48 bit address information)

    (IP Address$_1$, Port$_1$, IP Address$_2$, Port$_2$)
  - For an application, sockets are the access points to the network
  - A socket can be used for several connections at the same time
- TCP connections are always **full-duplex** and **point-to-point** connections
- TPDUs exchanged between the two communicating stations are called **segments**
- Segments are being exchanged for realizing
  - Connection establishment
  - Agreement on a window size
  - Data transmission
  - Sending of confirmations
  - Connection termination

TCP: Overview
RFCs: 793, 1122, 1323, 2018, 2581

- Point-to-point
  - One sender, one receiver
- Reliable, in-order byte steam:
  - 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) init’s sender, receiver state before data exchange
- Flow controlled
  - Sender will not overwhelm receiver

Transmission Control Protocol (TCP)

Socket Programming in TCP

Socket Programming in TCP

Server side

- The receiving application process (server) has to run at first
- The server provides a socket over which connection requests are received (i.e. a port is made available)
- In order to be able to receive requests of several clients, the server provides a new socket for a connection request of each client

Client side

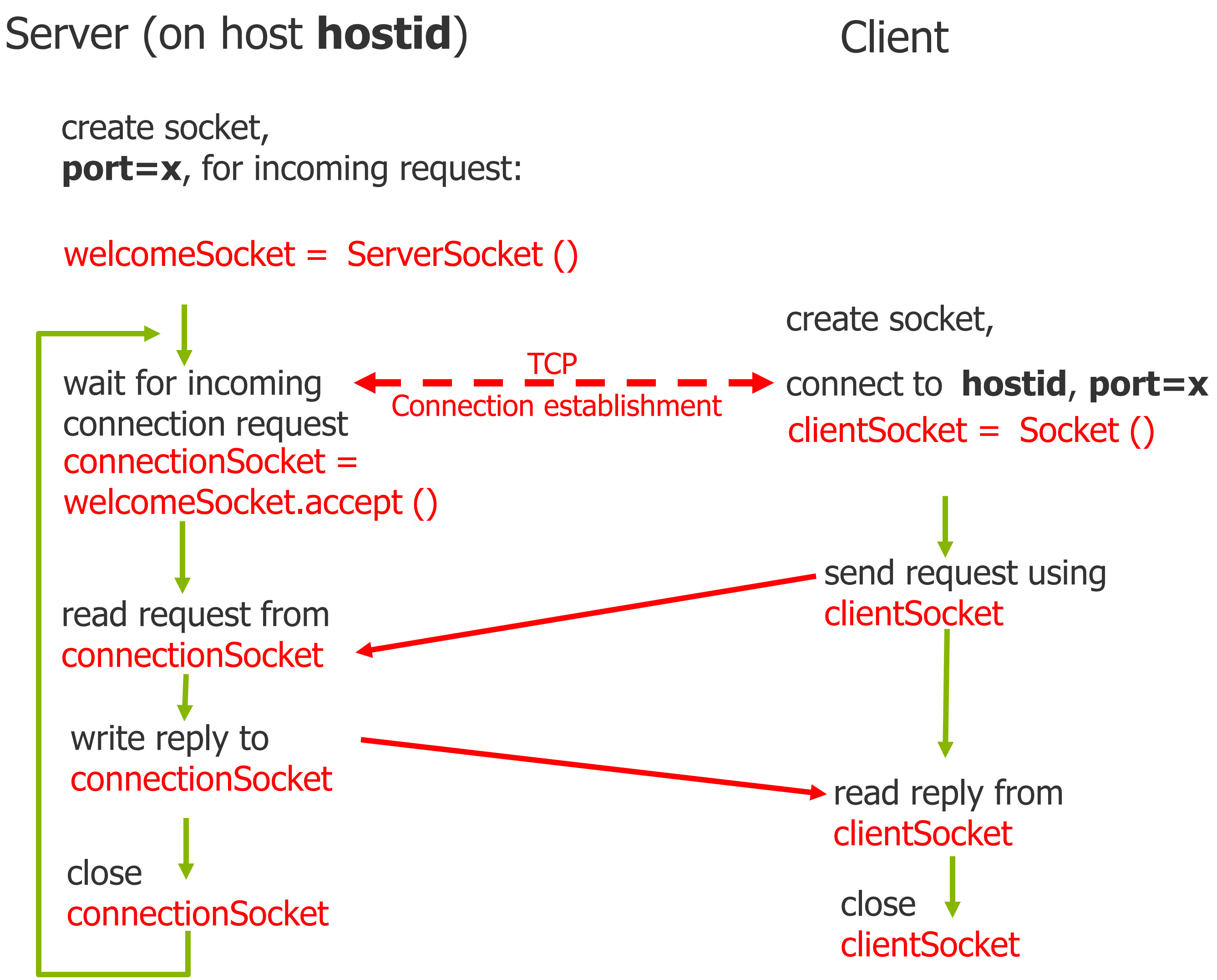
- The client generates a socket
- The client creates a request with IP address and port of the server
- When the client creates its socket, a connection establishment to the server is made

Socket Primitives in TCP

- For communication via TCP, a set of primitives exists which an **application programmer** can use for initializing and carrying out a communication.
- The essential primitives are:

Primitive	Meaning
SOCKET	Creation of a new network access point
BIND	Assign a local address with the socket
LISTEN	Wait for arriving connecting requests
ACCEPT	Accept a connecting request
CONNECT	Attempt of a connection establishment
SEND	Send data over the connection
RECEIVE	Receive data on the connection
CLOSE	Release of the connection

Socket programming in TCP



Server (on host **hostid**)

Client

create socket,
port=x, for incoming request:

welcomeSocket = ServerSocket ()

wait for incoming
connection request
connectionSocket =
welcomeSocket.accept ()

TCP
Connection establishment

create socket,

connect to **hostid**, **port=x**
clientSocket = Socket ()

send request using
clientSocket

read request from
connectionSocket

write reply to
connectionSocket

read reply from
clientSocket

close
connectionSocket

close
clientSocket

Example: Java Client (TCP)

```
import java.io.*;
import java.net.*;

class TCPClient {
    public static void main(string argv[]) throws Exception
    {
        String sentence;
        String modifiedSentence;
        BufferedReader inFromUser = new BufferedReader(new InputStreamReader(System.in));
        Socket clientSocket = new Socket( “hostname” , 6789);
        DataOutputStream outToServer = new
                DataOutputStream(clientSocket.getOutputStream ());

        BufferedReader inFromServer = new BufferedReader(new
                InputStreamReader(clientSocket.getInputStream()));

        sentence = inFromUser.readLine();
        outToServer.writeBytes(sentence +  “₩ n” );
        modifiedSentence = inFromServer.readLine();
        System.out.println( “FROM SERVER: ”  + modifiedSentence);
        clientSocket.close();
    }
}
```

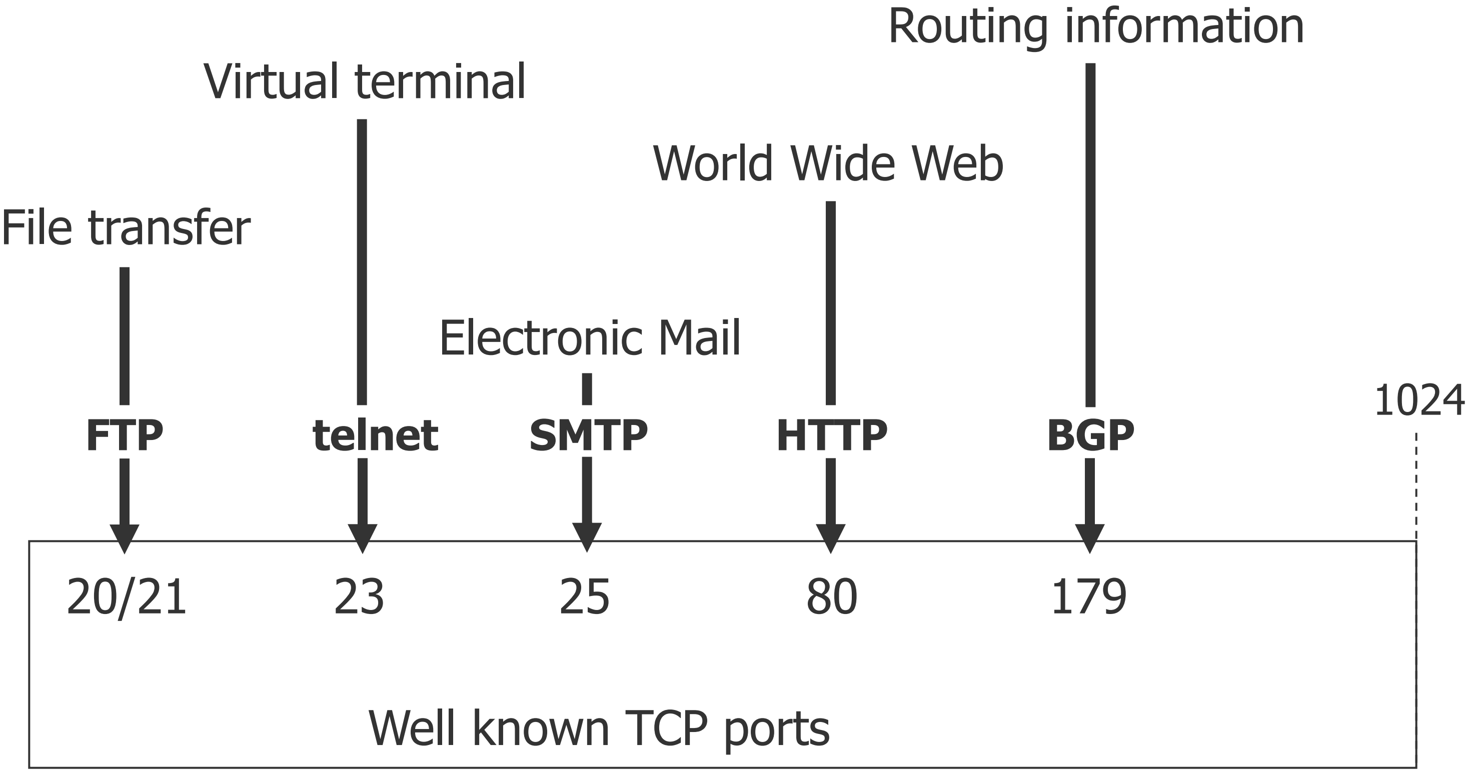
Example: Java Server (TCP)

```
import java.io.*;
import java.net.*;

class TCPServer {
    public static void main(string arg []) throws Exception
    {
        String clientSentence;
        String capitalizedSentence;
        ServerSocket welcomeSocket = new ServerSocket(6789);

        while (true) {
            Socket connectionSocket = welcomeSocket.accept();
            BufferedReader inFromClient =
                new BufferedReader(new
                    InputStreamReader(connectionSocket.getInputStream()));

            DataOutputStream outToClient = new
                DataOutputStream(connectionSocket.getOutputStream());

            clientSentence = inFromClient.readLine();
            capitalizedSentence = clientSentence.toUpperCase() + “₩ n” ;
            outToClient.writeBytes(capitalizedSentence);
        }
        welcomeSocket.close();
    }
}
```

Transmission Control Protocol (TCP)

The TCP Header

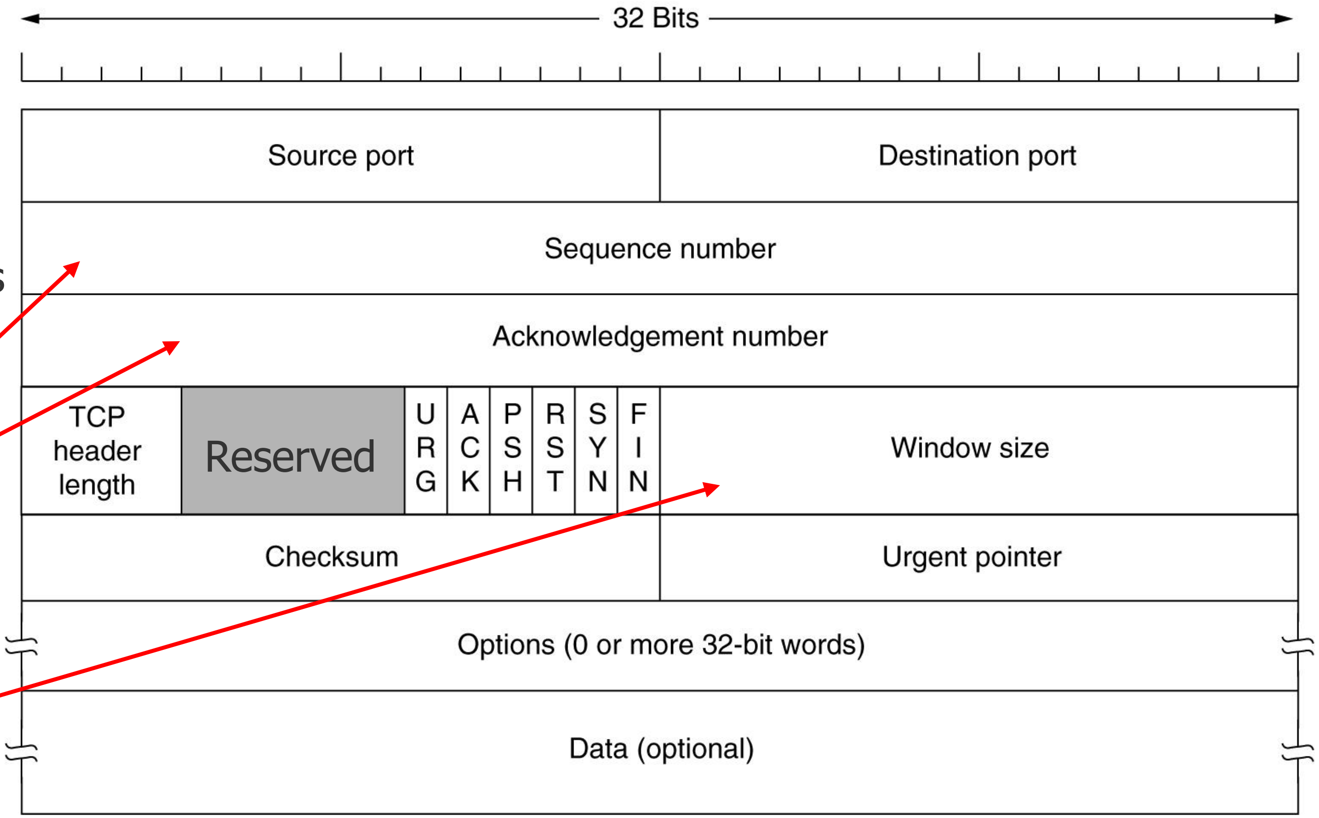
TCP-based Applications



- Port number is 16-bit address, currently three different ranges
- Ports in the range 0-1023 are **Well-Known** (a.k.a. “system”)
- Ports in the range 1024-49151 are **Registered** (a.k.a. “user”)
- Ports in the range 49152-65535 are **Dynamic/Private**
- See for more information: http://www.iana.org/assignments/port-numbers

The TCP Header

- 20 byte header
- Plus options
- Up to 65495 data bytes



32 Bits

Source port									Destination port
Sequence number									
Acknowledgement number									
TCP header length	Reserved	URG	ACK	PSH	RST	SYN	FIN		Window size
Checksum									Urgent pointer
Options (0 or more 32-bit words)									
Data (optional)									

Counting by bytes of data (not segments!)

Number of bytes receiver willing to accept

The TCP Header

- Source and Destination Port: port number of sender resp. receiver
- Sequence Number/Acknowledgment Number: Segments have a 32 bit sequence and acknowledgement number for the window mechanism in flow control (Sliding Window).
  - Sequence and acknowledgement number count **single bytes**!
  - The **acknowledgement number** indicates the **next expected byte**!
  - **Sequence numbers** begin not necessarily with 0! A **random value is chosen** to avoid a possible mix-up of old (late) segments.
  - Piggybacking, i.e., an acknowledgement can be sent in a data segment.
- Header Length: As in case of IP, also the TCP header has an indication of its length. The length is **counted in 32-bit words**.
- Window Size: Size of the receiver's buffer for the connection.
  - Used in flow control: the window of a flow indicates, how many bytes at the same time can be sent.
  - The size of the buffer indicates, the number of bytes the receiver can accept.
  - The window of flow control is adapted to this value.

The TCP Header

- Flags:
  - URG: Signaling of special important data, e.g., abort, Ctrl-C
  - ACK: This bit is set, if an acknowledgement is sent
  - PSH: Immediate transmission of data, no more waiting for further data
  - RST: Reset a connection, e.g., during a host crash or a connecting rejection
    - Generally problems arise when a segment with set RST bit is received
  - SYN: set to 1 for connection establishment
  - FIN: set to 1 for connection termination
- Urgent pointer: indicates, at which position in the data field the urgent data ends (byte offset of the current sequence number).
- Option:
  - Negotiation of a **window scale:** Window size field can be shifted up to 14 bits ➡ allowing windows of up to 2^{30} bytes
  - Use of **Selective Repeat** instead of **Go-Back-N** in the event of an error
  - Indication of the **Maximum Segment Size** (**MSS**) to determine the size of the data field

TCP Pseudo Header

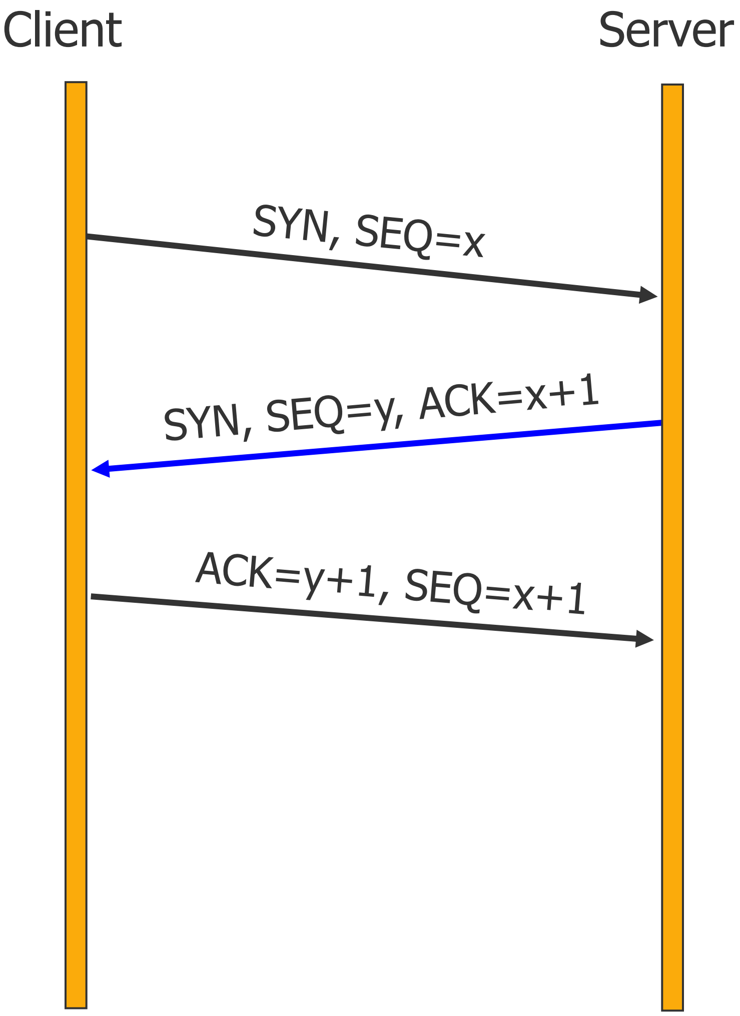
- Checksum: serves among other things for the verification that the packet was delivered to the correct device.
  - The checksum is computed using a **pseudo header**. The pseudo header is placed in front of the TCP header, the checksum is computed based on both headers (the checksum field is here 0).
    - The checksum is computed as the 1-complement of the sum of all 16-bit words of the segment including the pseudo header.
  - The receiver also places the pseudo header in front of the received TCP header and executes the same algorithm (the result must be 0).

<table>
<tr><td colspan="3">Source address (IP)</td></tr>
<tr><td colspan="3">Destination address (IP)</td></tr>
<tr><td>00000000</td><td>Protocol = 6</td><td>Length of the TCP segment</td></tr>
</table>

Transmission Control Protocol (TCP)

Connection Management

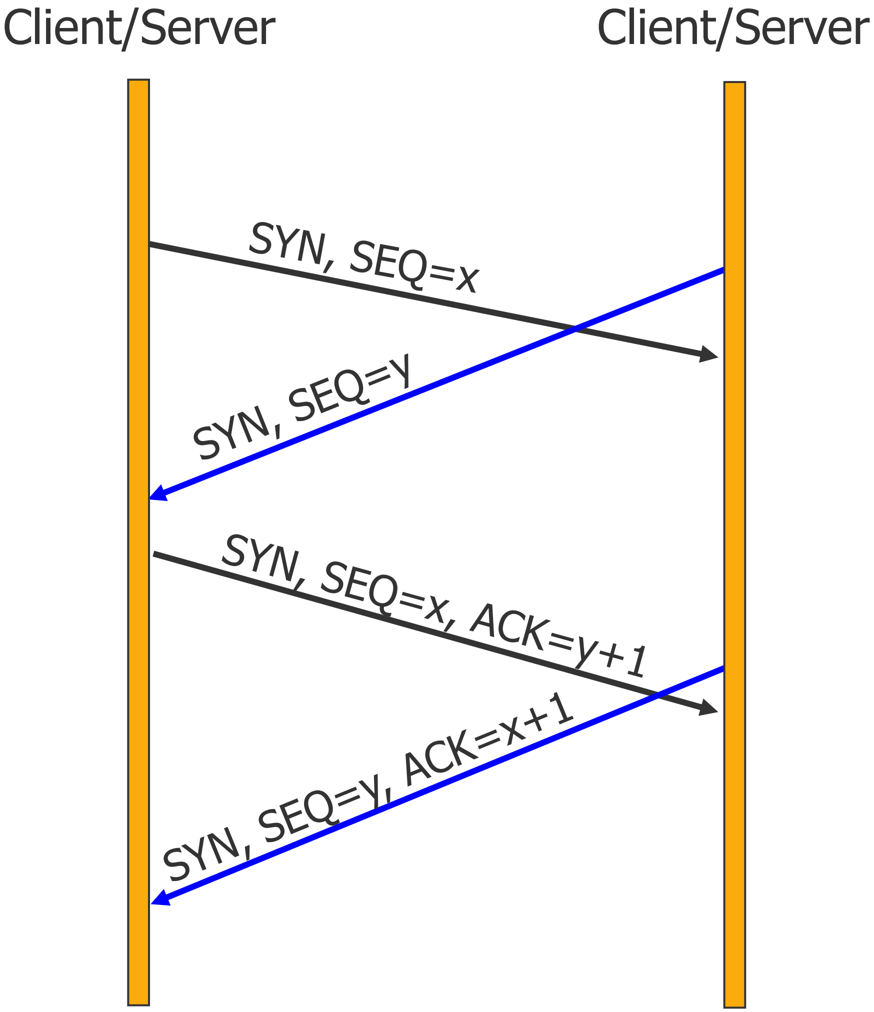
TCP Connection Management:
1. Connection Establishment



Three Way Handshake

- The server waits for connection requests using LISTEN and ACCEPT.
- The client uses the CONNECT operation by indicating IP address, port number, and the acceptable maximum segment size (MSS).
- CONNECT sends a SYN.
- If the destination port of the CONNECT is identical to the port number on which the server waits, the connection is accepted, otherwise it is rejected with RST.
- The server also sends a SYN to the client and acknowledges at the same time the receipt of the client's SYN segment.
- The client sends an acknowledgement for the SYN segment of the server. The connection is established.

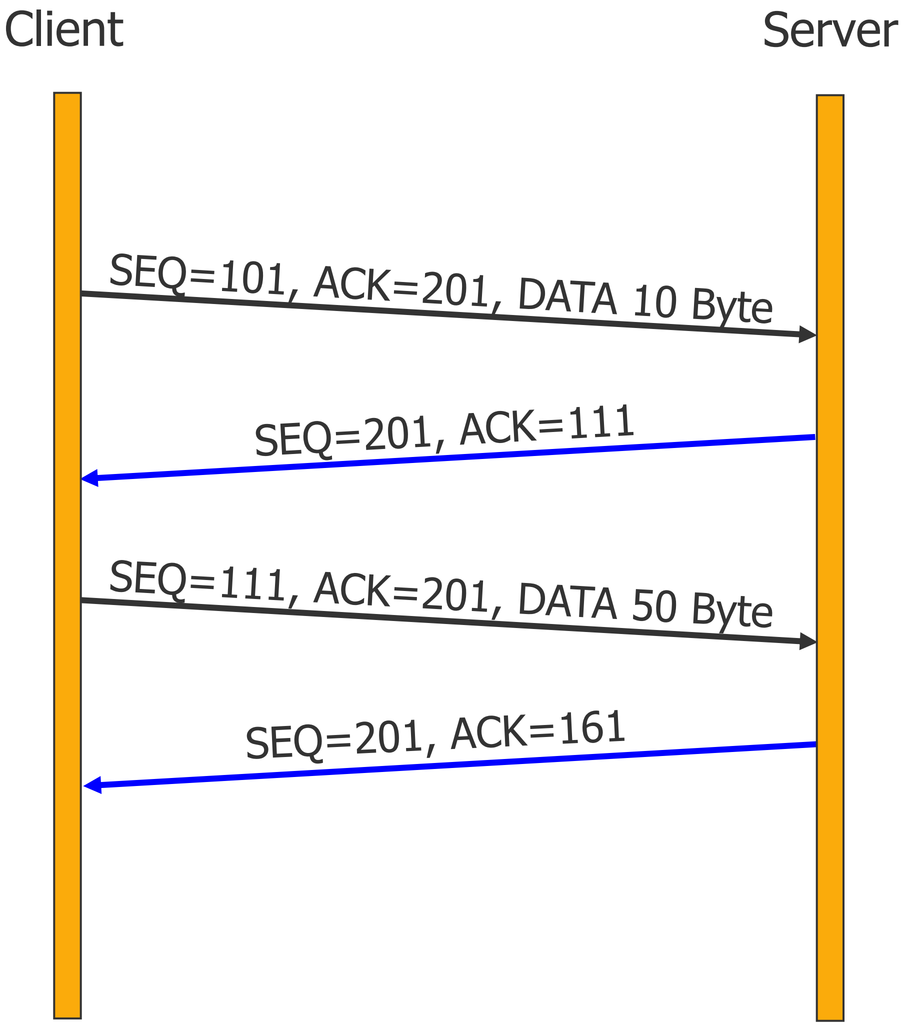
TCP Connection Management: Irregular Connection Establishment



- Two computers at the same time try to establish a connection between the same sockets.
- Connections are characterized by their endpoints; only **one** connection is established between a pair of endpoints.
- The endpoints are uniquely characterized by:

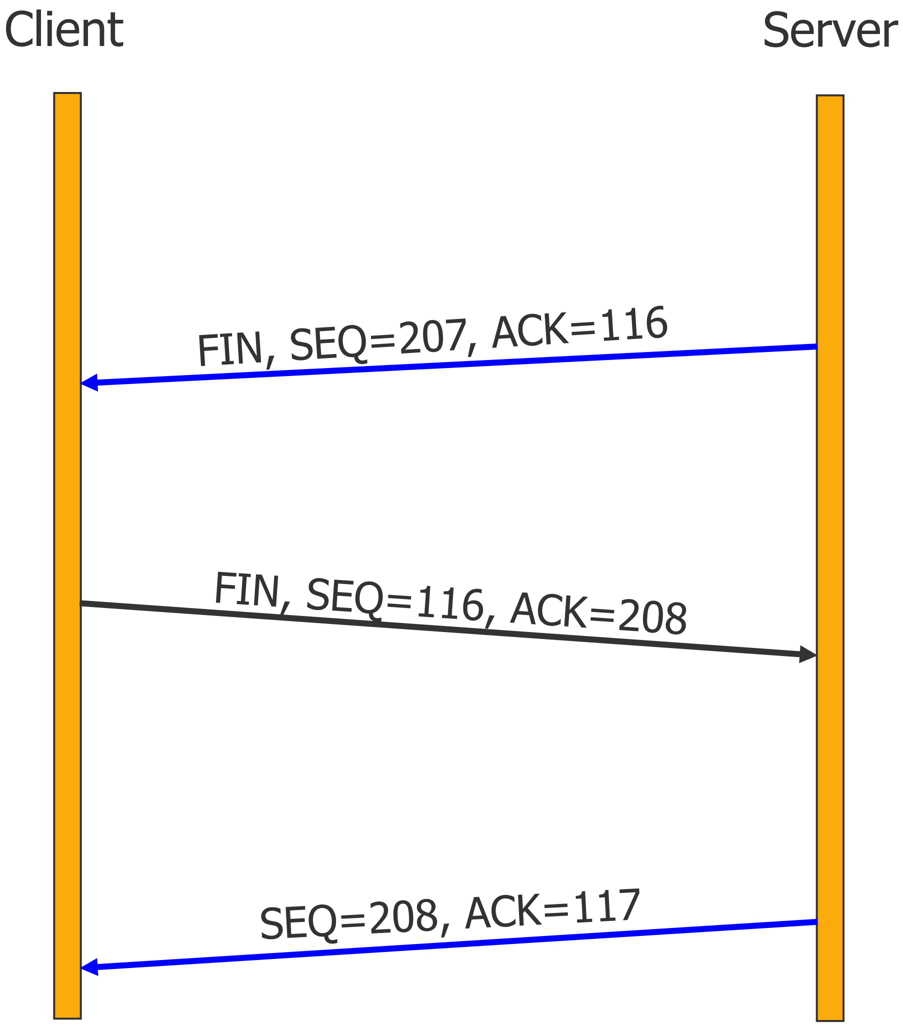
  (IP Address$_1$, Port$_1$, IP Address$_2$, Port$_2$)

TCP Connection Management: 2. Data Transmission


Client
Server
SEQ=101, ACK=201, DATA 10 Byte
SEQ=201, ACK=111
SEQ=111, ACK=201, DATA 50 Byte
SEQ=201, ACK=161

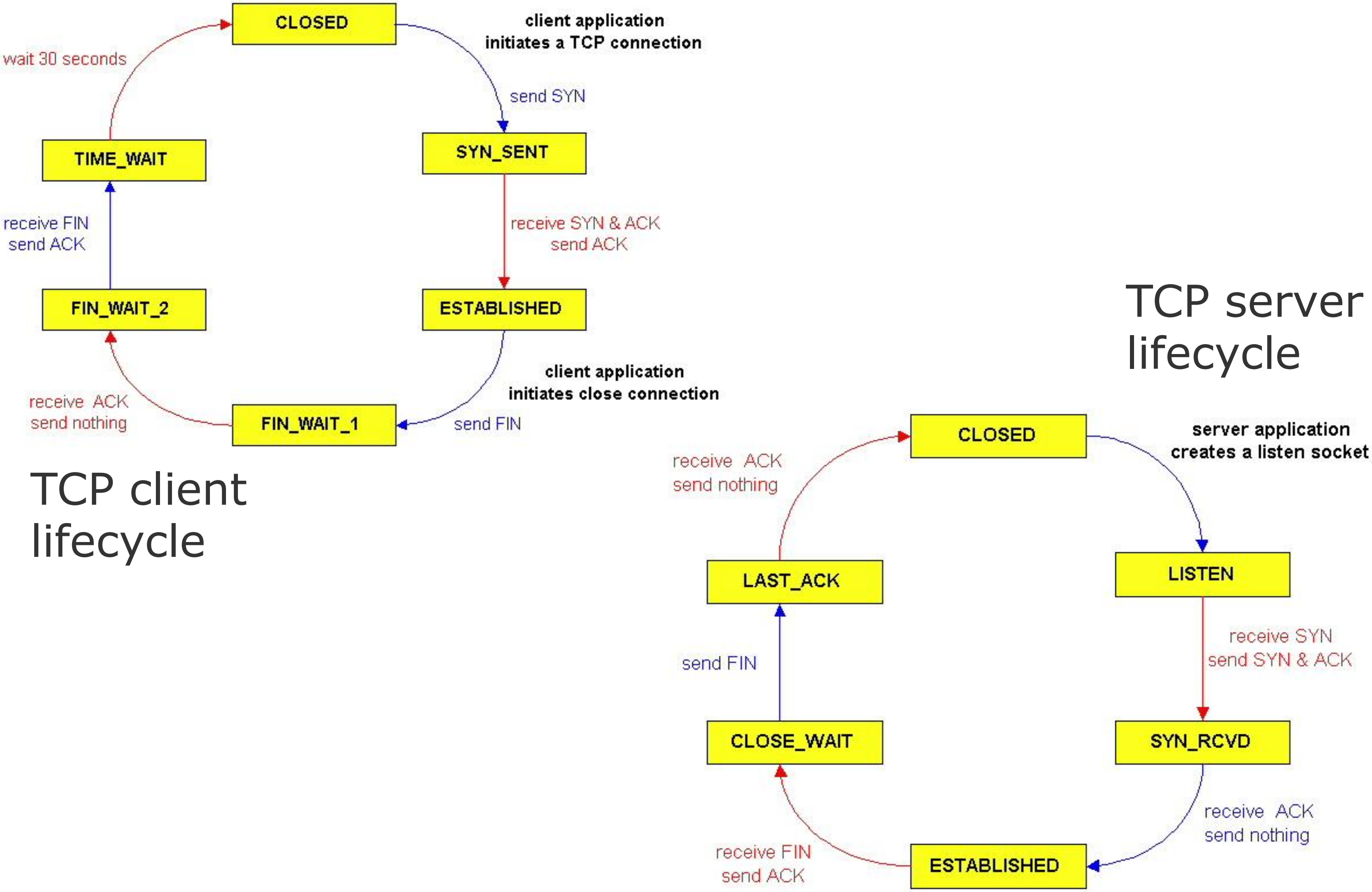
- Full-duplex connection
- Segmentation of a byte stream into segments.
  - Usual sizes are 1500, 536, or 512 byte; thus IP fragmentation is avoided.
  - All hosts have to accept TCP segments of 536 byte +20 byte = 556 byte
- Usual acknowledgement mechanism:
  - All segments up to ACK-1 are confirmed. If the sender has a timeout before an ACK, he repeats the sending.
- Usual procedure for repeating:
  - Go-Back-N or Selective Repeat

TCP Connection Management:
3. Connection Termination



- Termination as two simplex connections
- Send a FIN segment
- If the FIN segment is confirmed, the direction is “switched off”. The opposite direction remains however still open, data can be still further sent.
  - Half-open connections!
- Use of timers to protect against packet loss.

TCP Connection Management



CLOSED

client application initiates a TCP connection

send SYN

SYN_SENT

receive SYN & ACK
send ACK

ESTABLISHED

client application initiates close connection

send FIN

FIN_WAIT_1

receive ACK
send nothing

FIN_WAIT_2

receive FIN
send ACK

TIME_WAIT

wait 30 seconds

TCP client lifecycle

TCP server lifecycle

CLOSED

server application creates a listen socket

LISTEN

receive SYN
send SYN & ACK

SYN_RCVD

receive ACK
send nothing

ESTABLISHED

receive FIN
send ACK

CLOSE_WAIT

send FIN

LAST_ACK

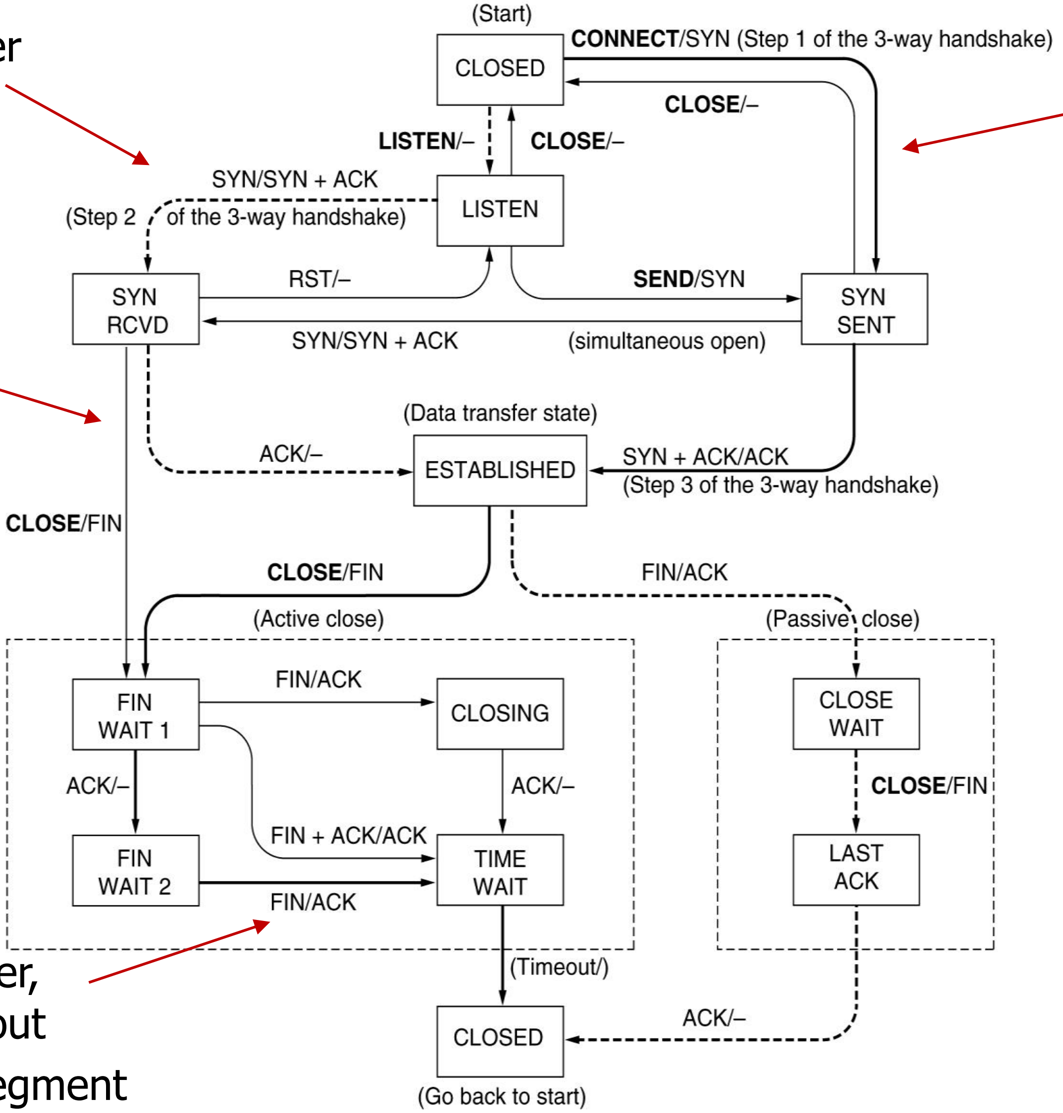
receive ACK
send nothing

The Entire TCP Connection

Normal path of server

Unusual events

Normal path of client



Event/action pair

- Event: System call by user, arrival of segment, timeout
- Action: Send a control segment

States during a TCP Session

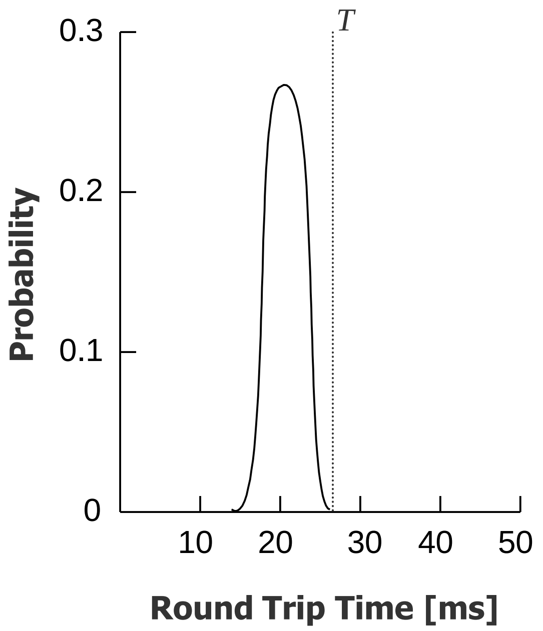
State	Description
CLOSED	No active communications
LISTEN	The server waits for a connection request
SYN RCVD	A connection request was received and processed, wait for the last ACK of the connection establishment
SYN SENT	Application began to open a connection
ESTABLISHED	Connection established, transmit data
FIN WAIT 1	Application starts a connection termination
FIN WAIT 2	The other side confirms the connection termination
TIME WAIT	Wait for late packets
CLOSING	Connection termination
CLOSE WAIT	The other side initiates a connection termination
LAST ACK	Wait for late packets

Transmission Control Protocol (TCP)

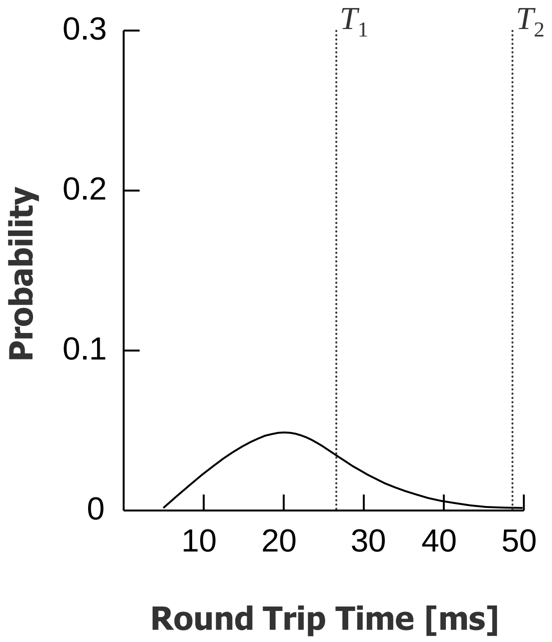
Timer Management

Timer Management with TCP

- TCP uses several timers
- **Retransmission timer** (for repeating a transmission)
  - But: how to select the timer value?
  - Probability density of the time till an acknowledgement arrives:



Data Link Layer

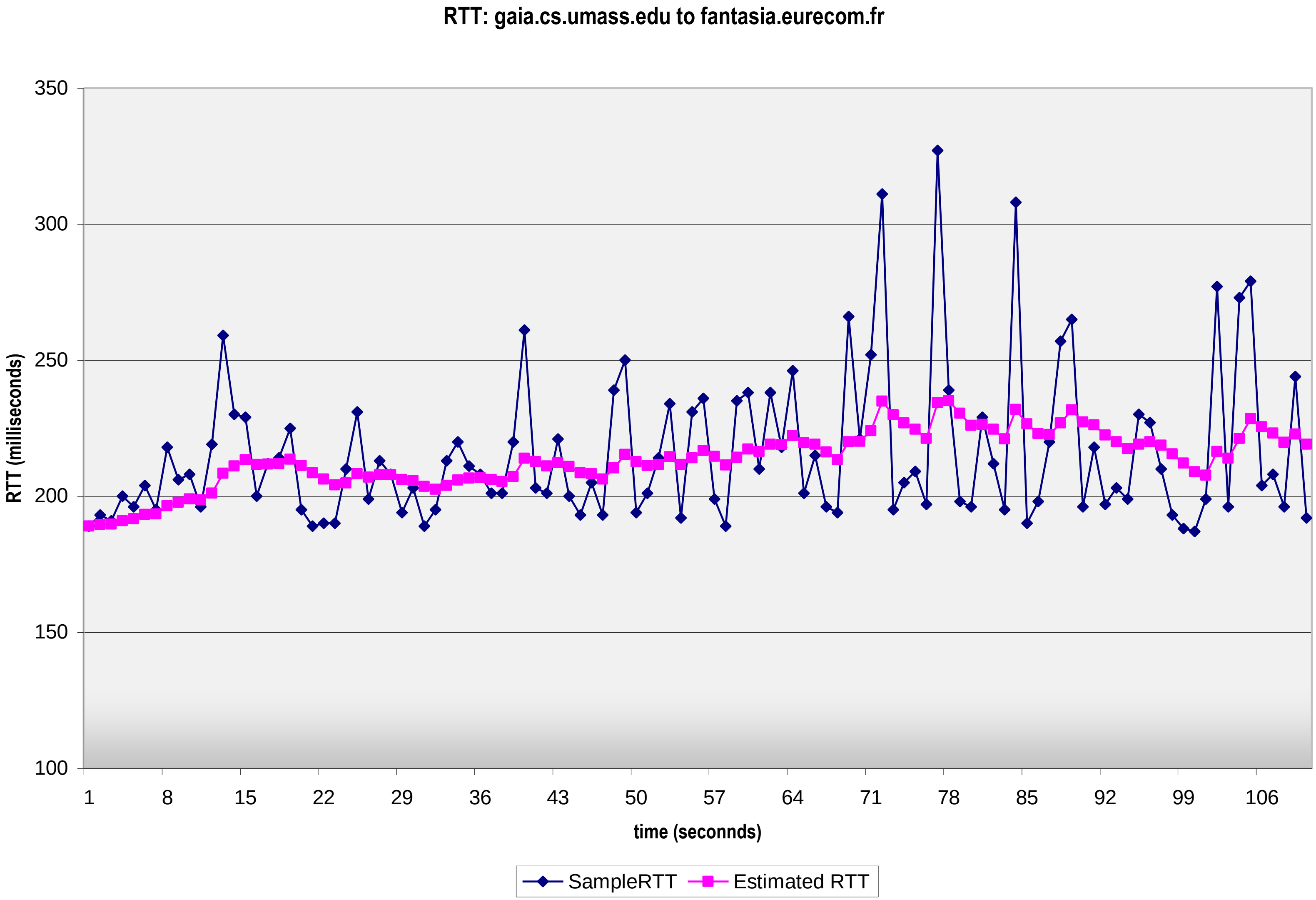


Transport Layer

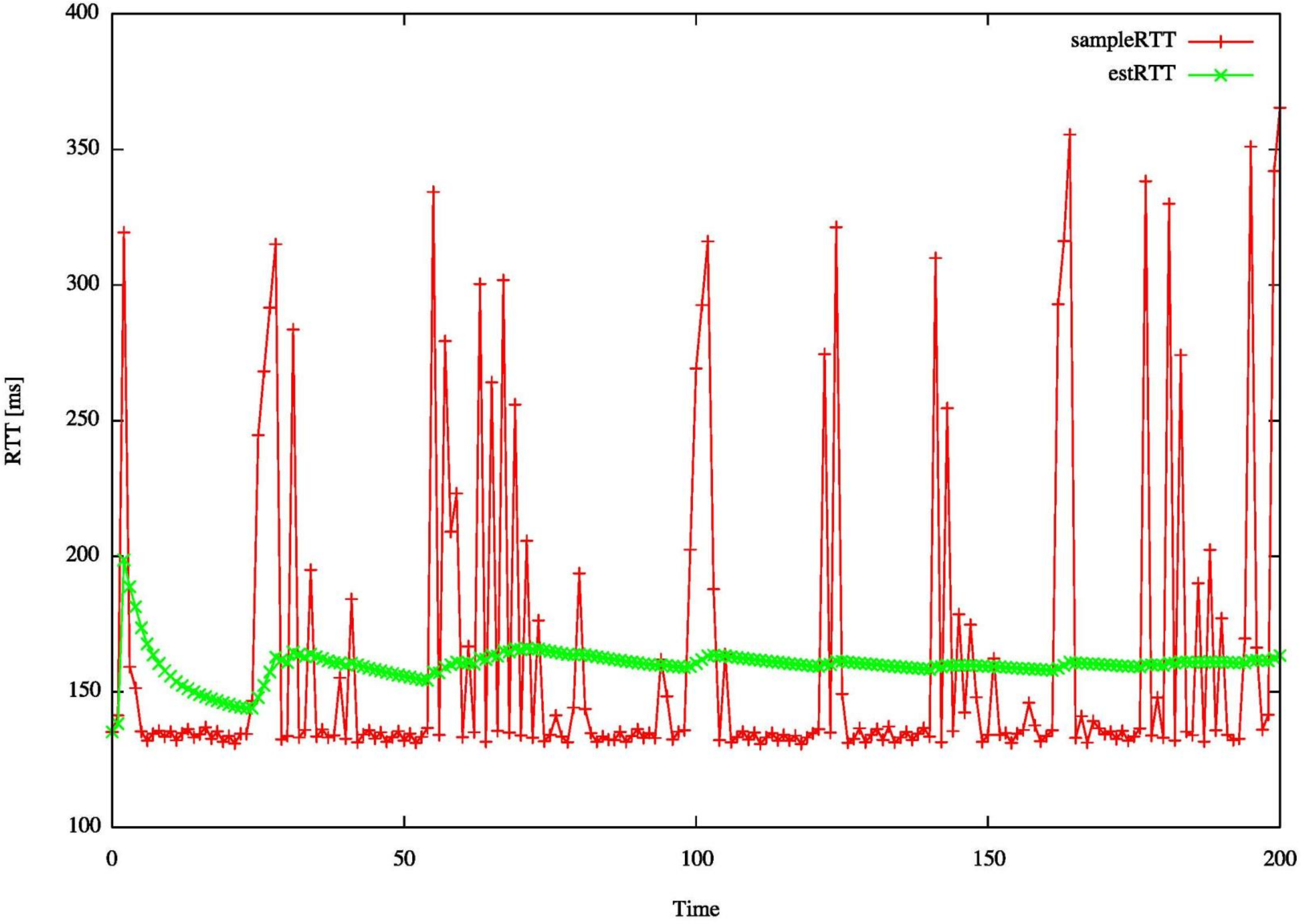
Problem on the transport layer:

- T_1 is too small: too many retransmissions
- T_2 is too large: inefficient for actual packet loss

Timer Management with TCP: Example RTT estimation



Timer Management with TCP: Example RTT estimation



Timer Management with TCP

- 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

Timer Management with TCP: Retransmission Timer

- Solution: dynamic algorithm, which adapts the timer by current measurements of the network performance.
- **Algorithm of Jacobson (1988)**
  - TCP manages a variable RTT (Round Trip Time) for each connection
  - RTT is momentarily the best **estimation** of the round trip time
  - When sending a segment, a timer is started which measures the time the acknowledgement needs and initiates a retransmission if necessary.
  - If the acknowledgement arrives before expiration of the timer (after a time unit $sampleRTT$), RTT is updated:

$$RTT = \alpha \times RTT + (1 - \alpha) \times sampleRTT$$

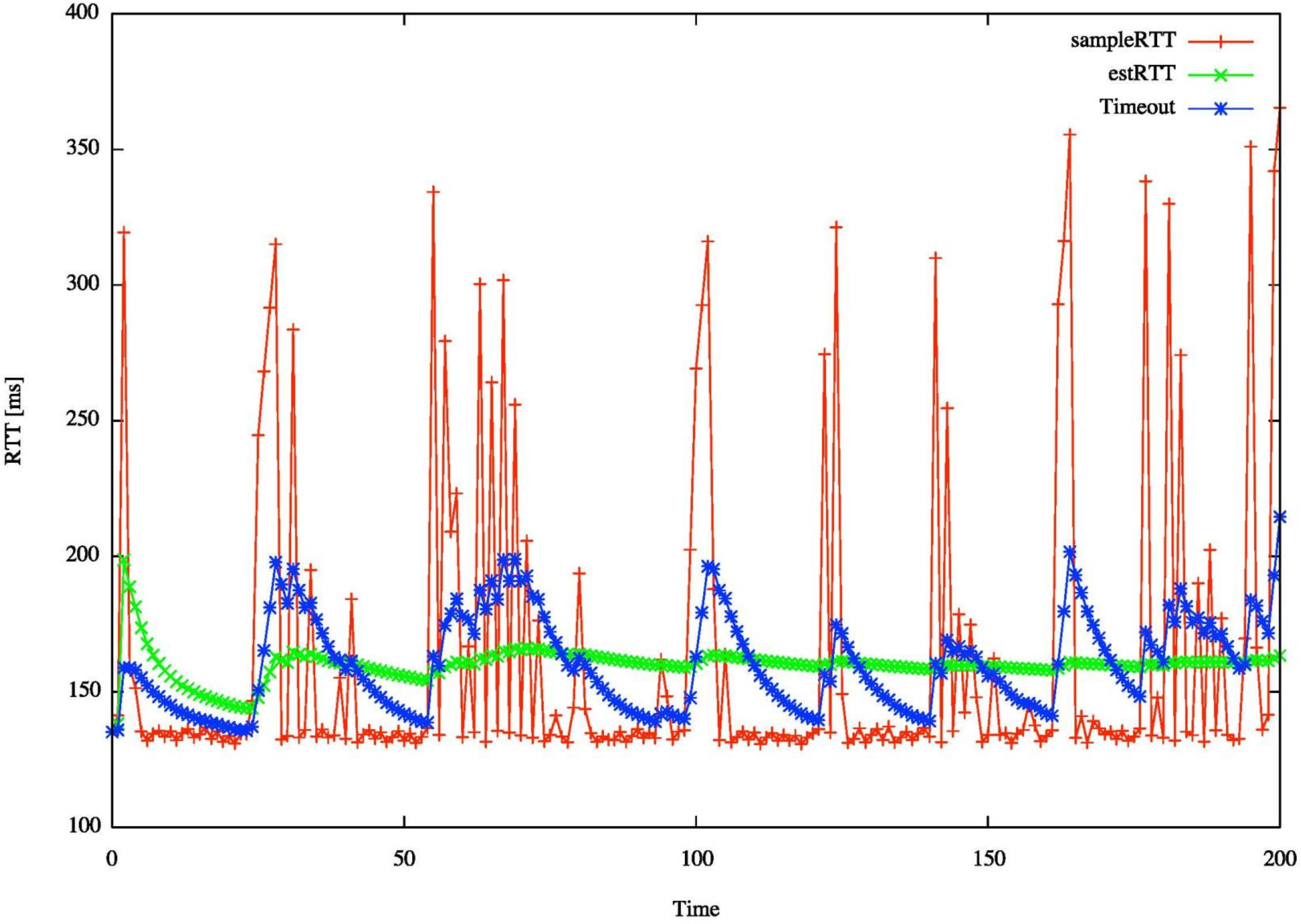
  - α is a smoothing factor, typically 0.875

  - The choice of the timeout is based on RTT

$$Timeout = \beta \times RTT$$

  - At the beginning, β was chosen as 2, but this was too inflexible

Timer Management with TCP



Timer Management with TCP: Retransmission Timer

- **Improvement (Jacobson):** set timer proportionally to the standard deviation of the arrival time of acknowledgements
  - Computation of standard deviation:

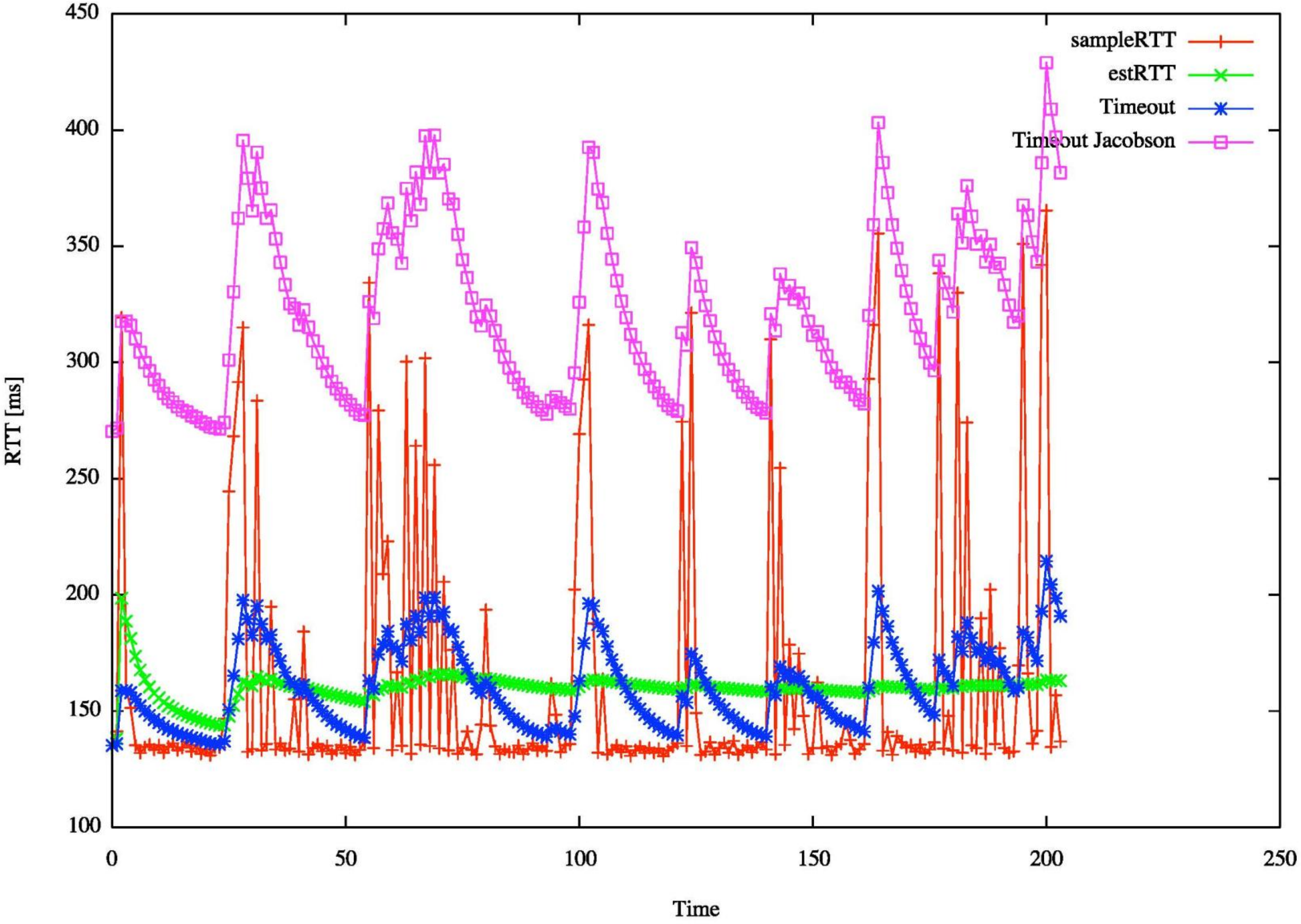
$$devRTT = \gamma \times devRTT + (1 - \gamma) \times |RTT - sampleRTT|$$

  - Typically $\gamma = 0.75$
  - Standard timeout interval:

$$Timeout = RTT + 4 \times devRTT$$

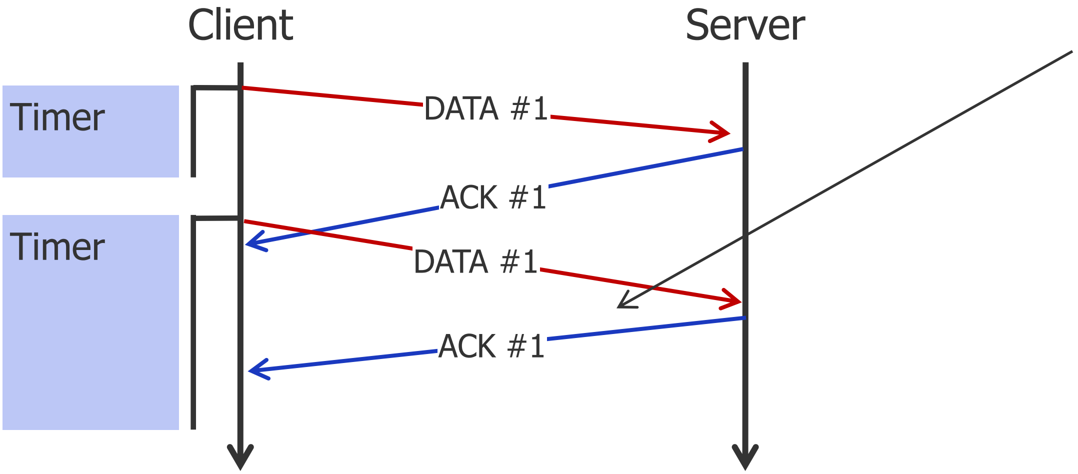
  - The factor 4 was determined on the one hand by trying out, on the other hand because it is fast and simple to use in computations.

Timer Management with TCP



Timer Management with TCP: Retransmission Timer

- Karn’s Algorithm
  - Very simple proposal, which is used in most TCP implementations (optional)
  - Do not update RTT on any segments that have been **retransmitted**.
    - The timeout is doubled on each failure until the segments get through.



Is the ack for the first transmission or the second one?

Timer Management with TCP: Other Timers

Persistence timer

- Prevents a deadlock with a loss of the buffer release message of a receiver
- With expiration of the timer, the sender transfers a test segment. The response to this transmission contains the current buffer size of the receiver. If it is still zero, the timer is started again.

Keep-alive timer

- If a connection is inactive for a longer time, at expiration of the timer it is examined whether the other side is still living.
- If no response is given, the connection is terminated.
- Disputed function, not often implemented.

Time Wait timer

- During the termination of a connection, the timer runs for the double packet life time to be sure that no more late packets arrive.

Transmission Control Protocol (TCP)

Reliable Data Transfer

TCP Reliable Data Transfer

- Reliable transfer with 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

TCP Sender Events

- Data received from app:
    - Create segment with sequence number
    - Sequence number 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 acknowledges previously unacked segments
        - Update what is known to be acked
        - Start timer if there are outstanding segments

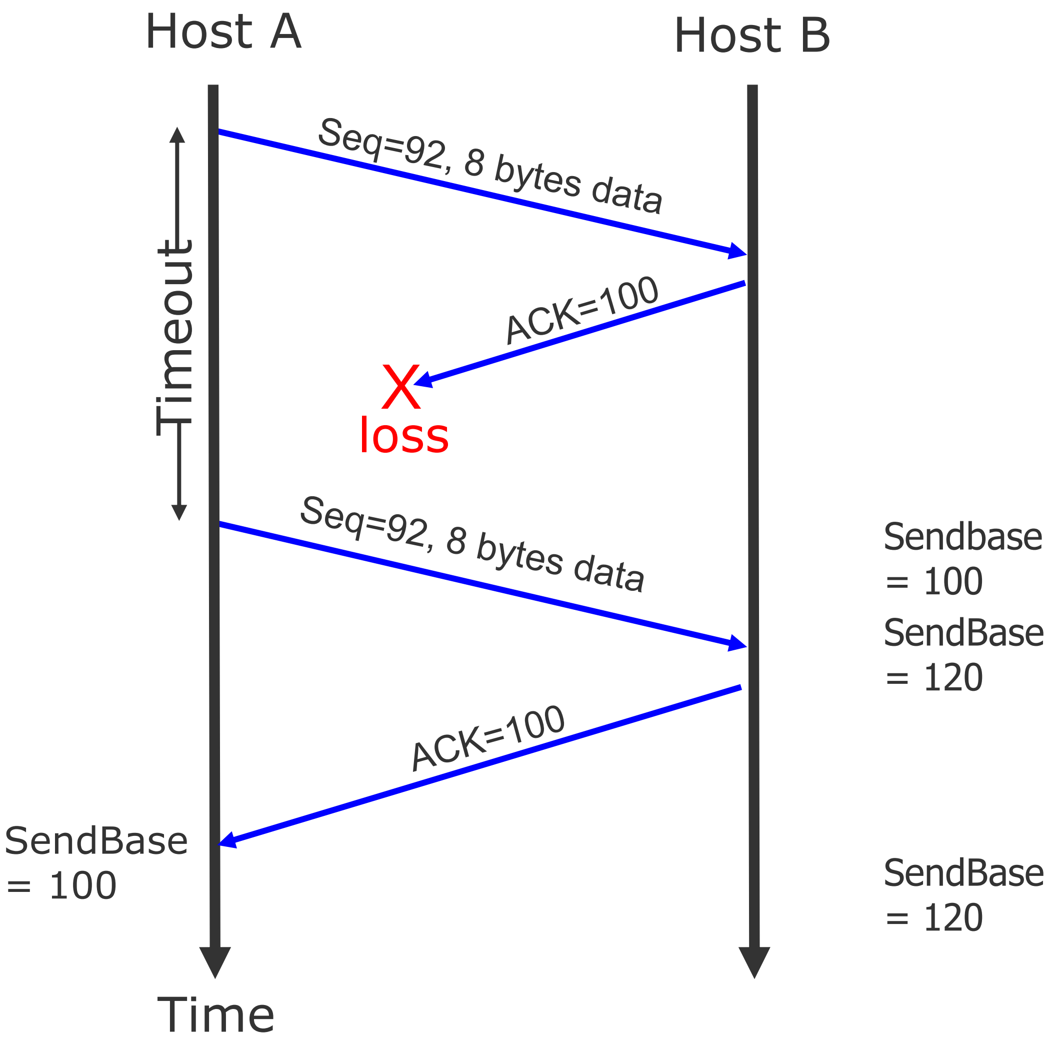
TCP Sender (simplified)

```
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 */
```

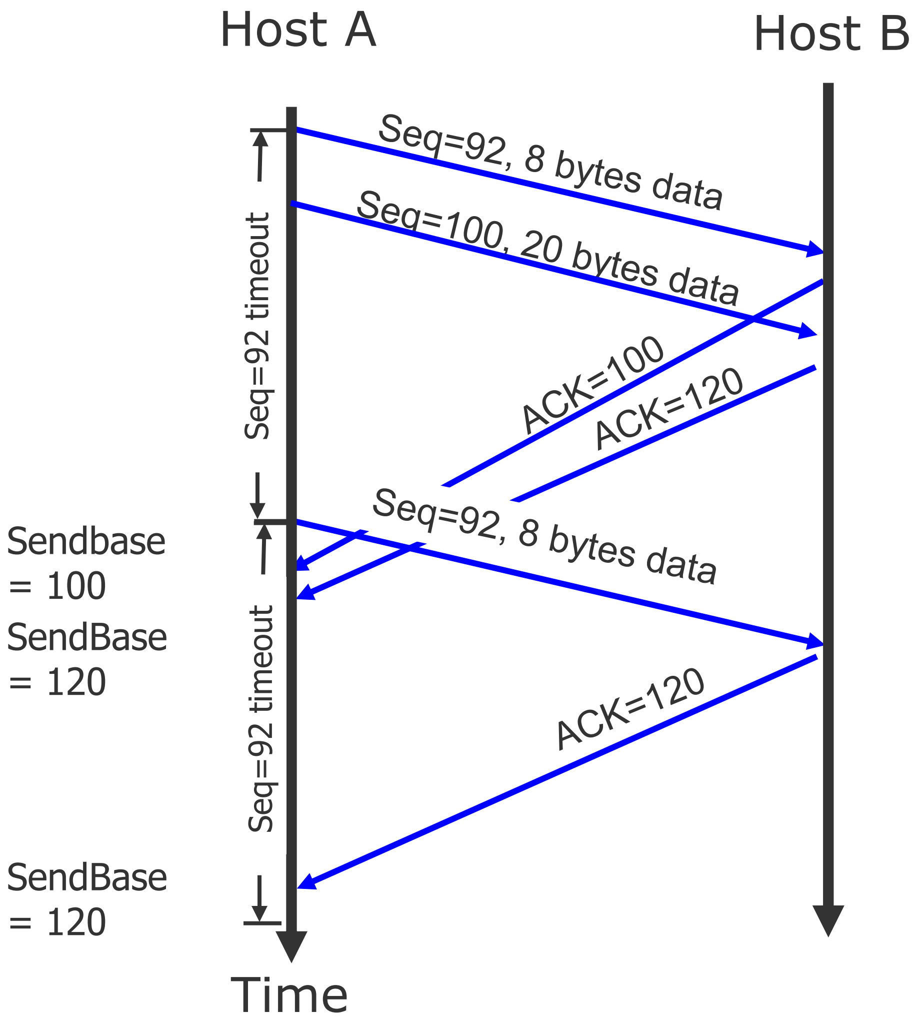
Comment:
- SendBase-1: last cumulatively ack'ed byte

Example:
- SendBase-1 = 71; y= 73, so the rcvr wants 73+; y > SendBase, so that new data is acked

TCP Retransmission Scenarios

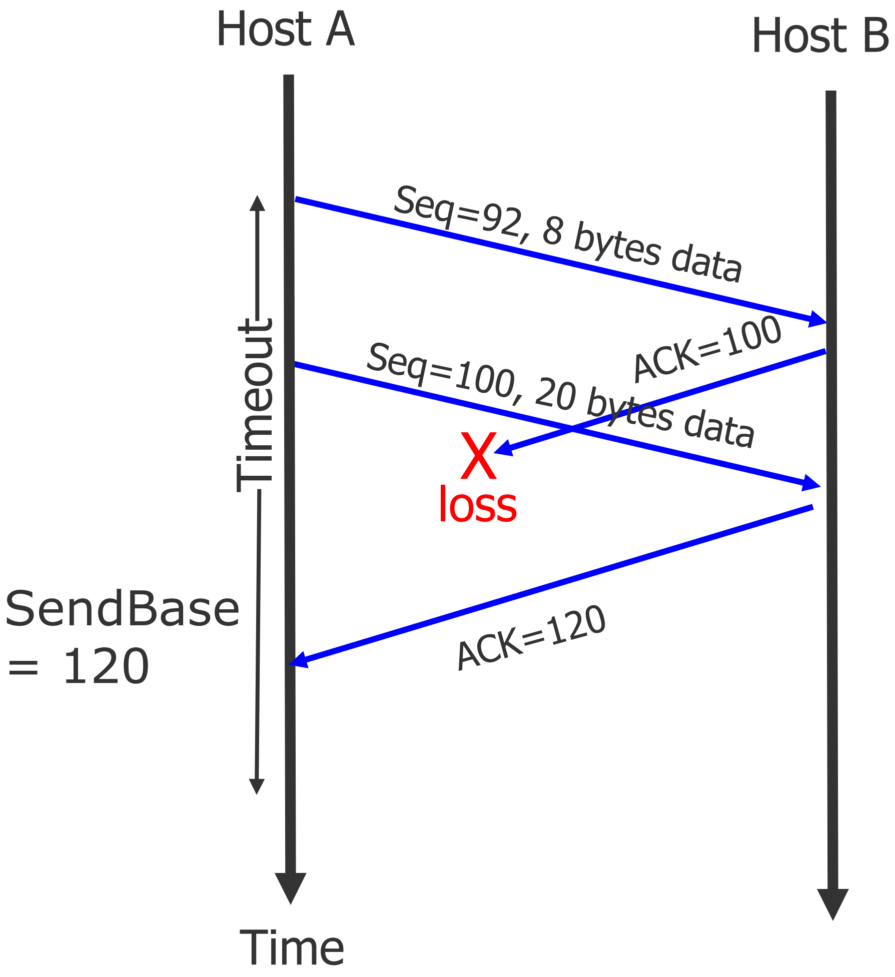


Lost ACK scenario



Premature timeout

TCP Retransmission Scenarios



Cumulative ACK scenario

TCP ACK generation [RFC 1122, RFC 2581]

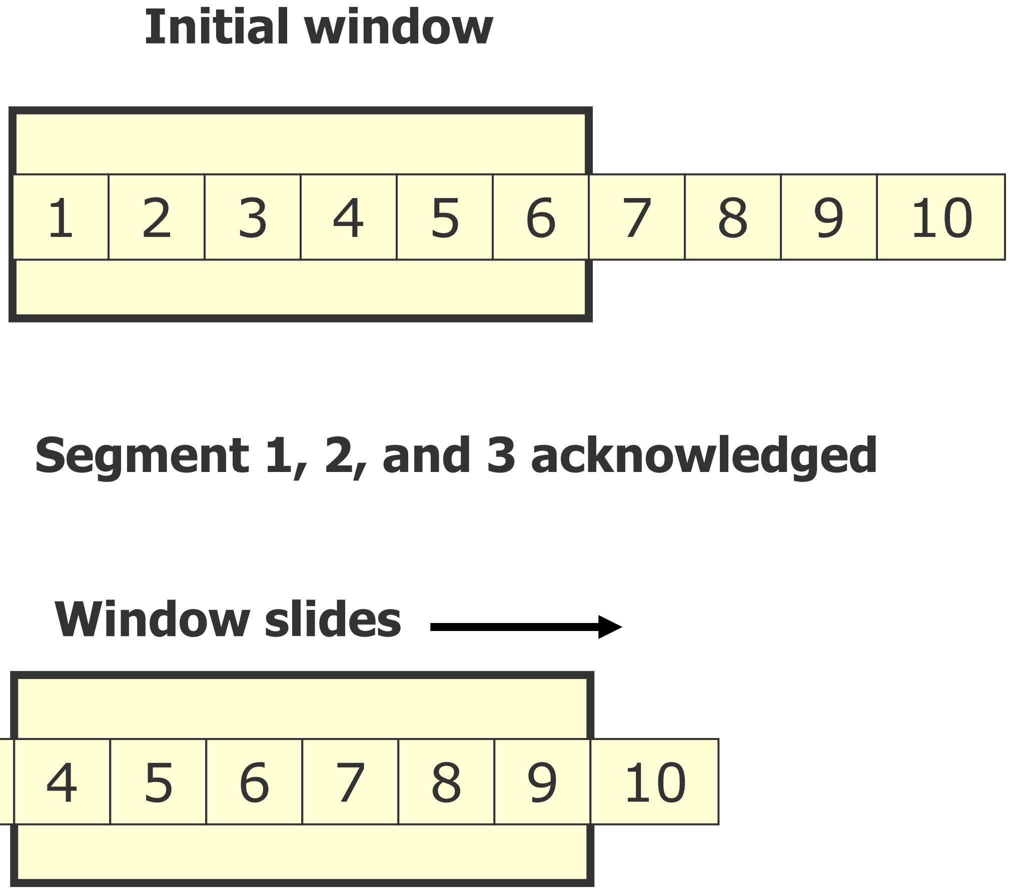
Event at Receiver	TCP Receiver Action
Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
Arrival of in-order segment with expected seq #. One other segment has ACK pending	Immediately 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	Immediately send ACK, provided that segment starts at lower end of gap

Transmission Control Protocol (TCP)

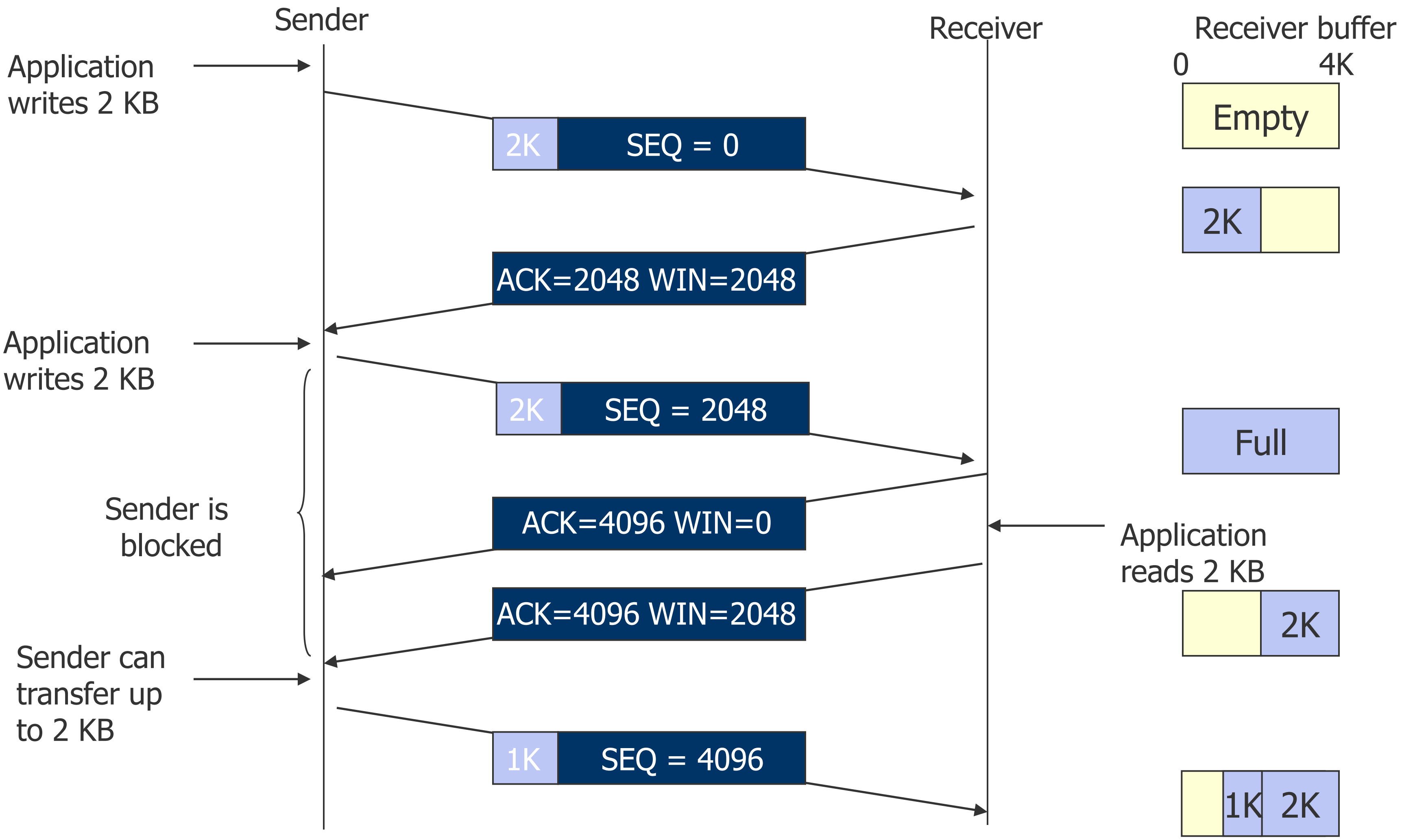
Flow Control

TCP Flow Control: Sliding Window

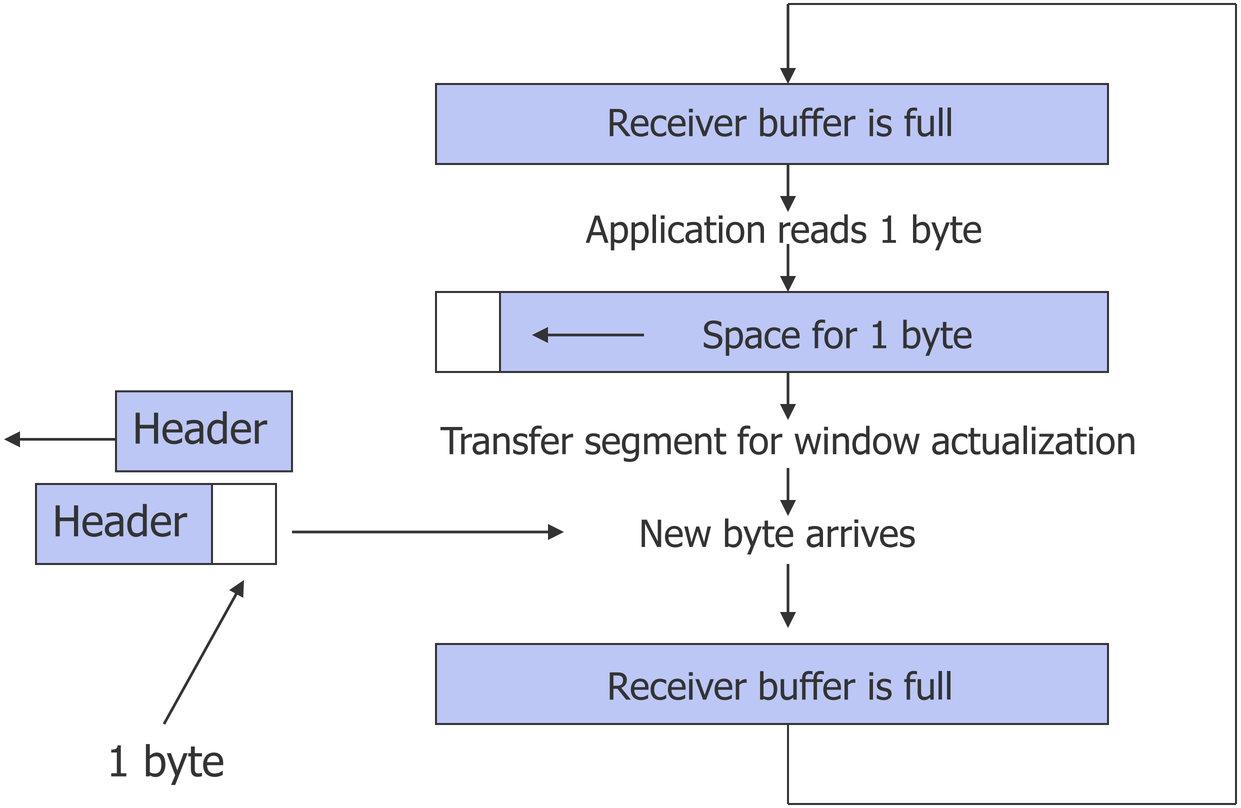
- To provide reliable data transfer, as on Layer 2, a sliding window mechanism is used.
- Differences:
  - Sender sends **bytes** according to the window size
  - Window is shifted by **n** byte as soon as an ACK for **n** bytes arrives
  - Exception: Urgent data (URGENT flag is set) are sent immediately
  - Characteristic: the window size can be changed during the transmission phase



TCP Flow Control: Sliding Window, Example

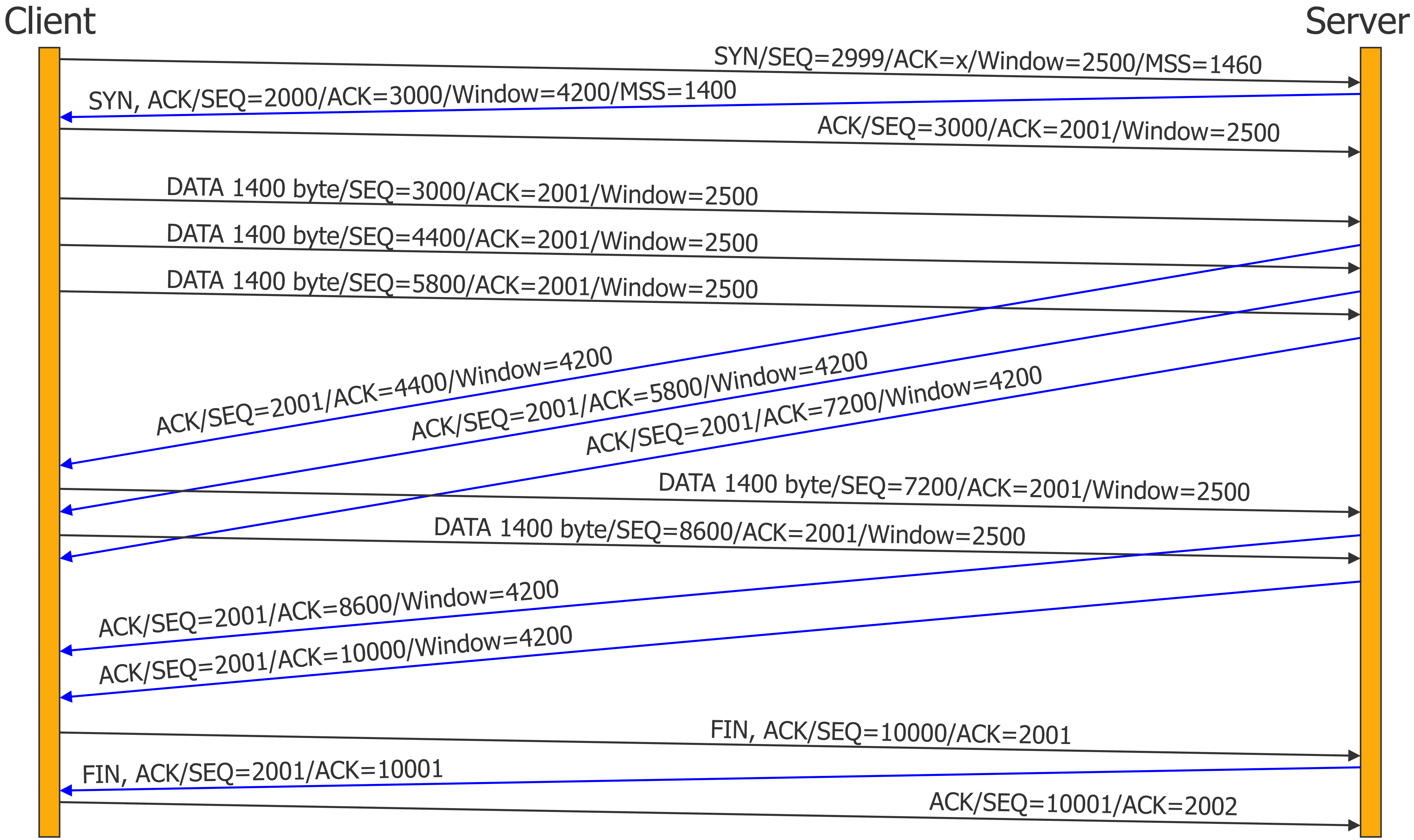


"Silly Window" Syndrome

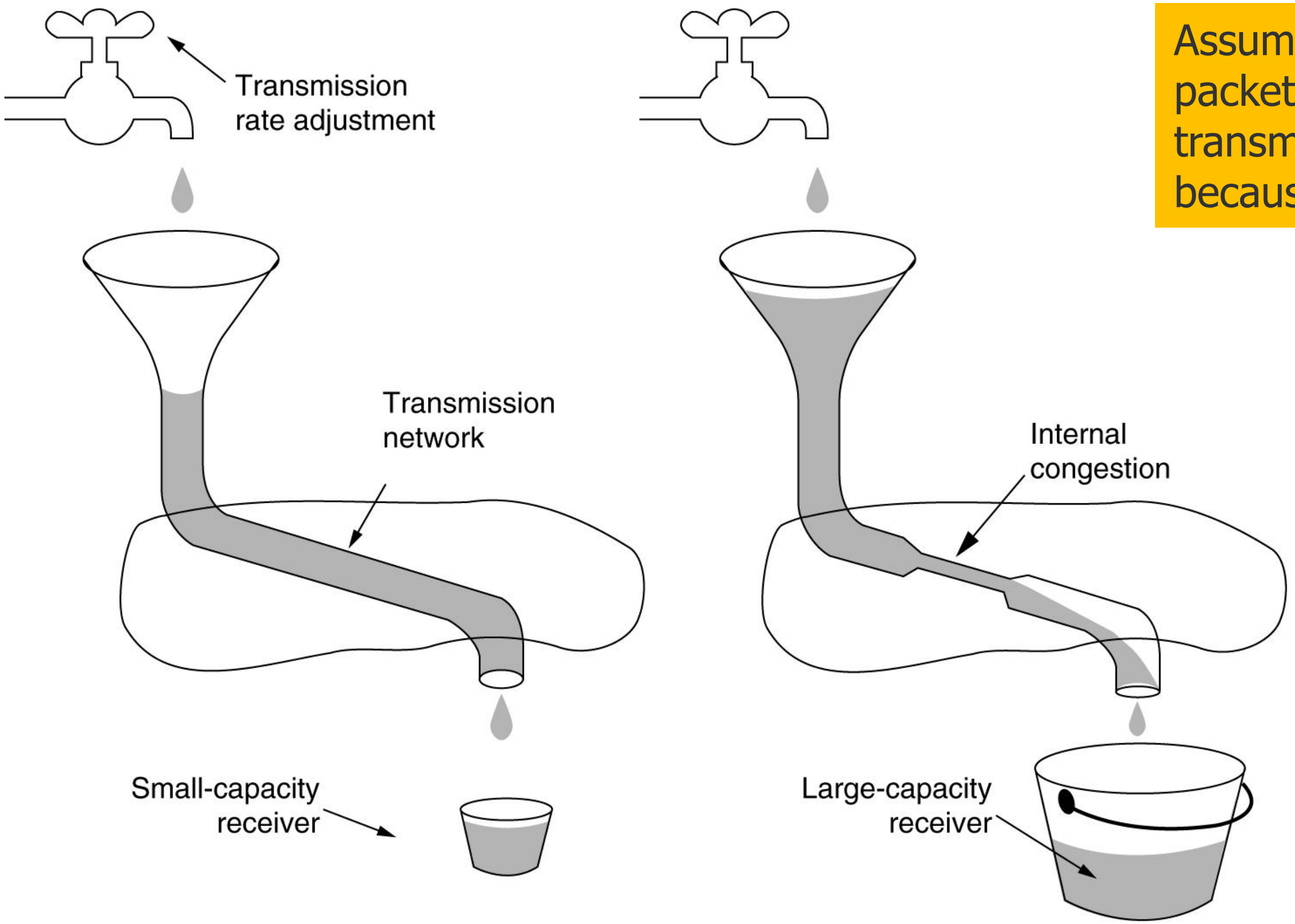


Solution of Clark:
The receiver must wait with the next window actualization until the receiver buffer again is reasonably empty.

The whole TCP Session


Client
Server
SYN/SEQ=2999/ACK=x/Window=2500/MSS=1460
SYN, ACK/SEQ=2000/ACK=3000/Window=4200/MSS=1400
ACK/SEQ=3000/ACK=2001/Window=2500
DATA 1400 byte/SEQ=3000/ACK=2001/Window=2500
DATA 1400 byte/SEQ=4400/ACK=2001/Window=2500
DATA 1400 byte/SEQ=5800/ACK=2001/Window=2500
ACK/SEQ=2001/ACK=4400/Window=4200
ACK/SEQ=2001/ACK=5800/Window=4200
ACK/SEQ=2001/ACK=7200/Window=4200
DATA 1400 byte/SEQ=7200/ACK=2001/Window=2500
DATA 1400 byte/SEQ=8600/ACK=2001/Window=2500
ACK/SEQ=2001/ACK=8600/Window=4200
ACK/SEQ=2001/ACK=10000/Window=4200
FIN, ACK/SEQ=10000/ACK=2001
FIN, ACK/SEQ=2001/ACK=10001
ACK/SEQ=10001/ACK=2002

Flow Control: Network Bottlenecks



Assumption:
packet loss is rarely because of transmission errors, rather because of overload situations.

Necessary:

Congestion Control

Capacity of the receiver:
Flow Control Window

Capacity of the network:
Congestion Window

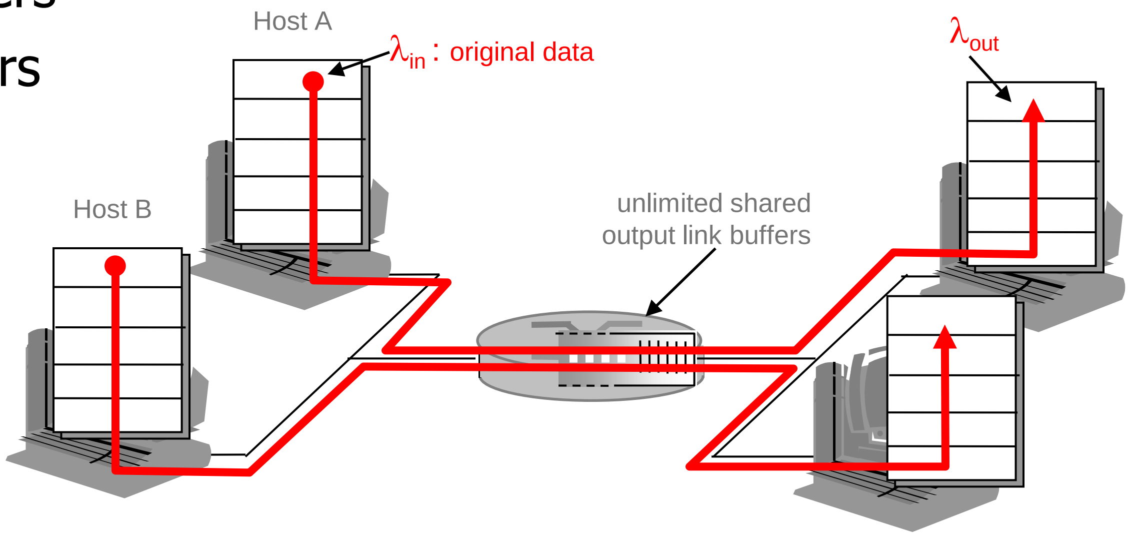
Principles of Congestion Control

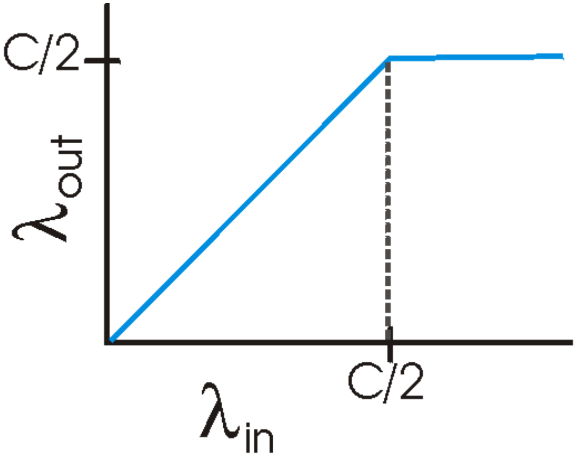
Principles of Congestion Control

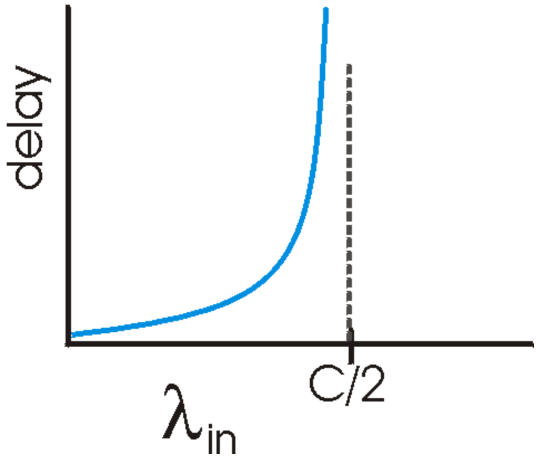
- Congestion:
  - Informally: “too many sources sending too much data too fast for network to handle”
  - Different from flow control!
  - Manifestations:
    - Lost packets (buffer overflow at routers)
    - Long delays (queueing in router buffers)
  - A top-10 problem!

Causes/costs of congestion: Scenario 1

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


Host A
λ_{in} : original data
λ_{out}
Host B
unlimited shared output link buffers

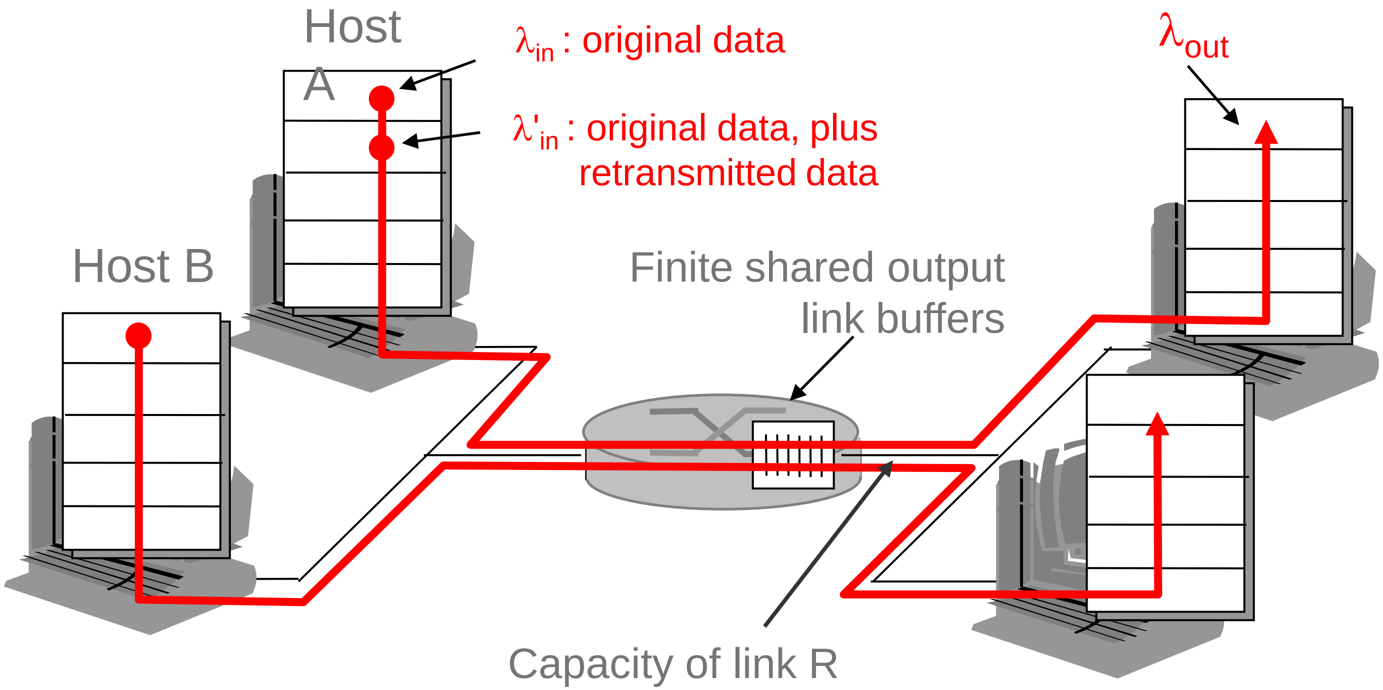




- Large delays when congested
- Maximum achievable throughput

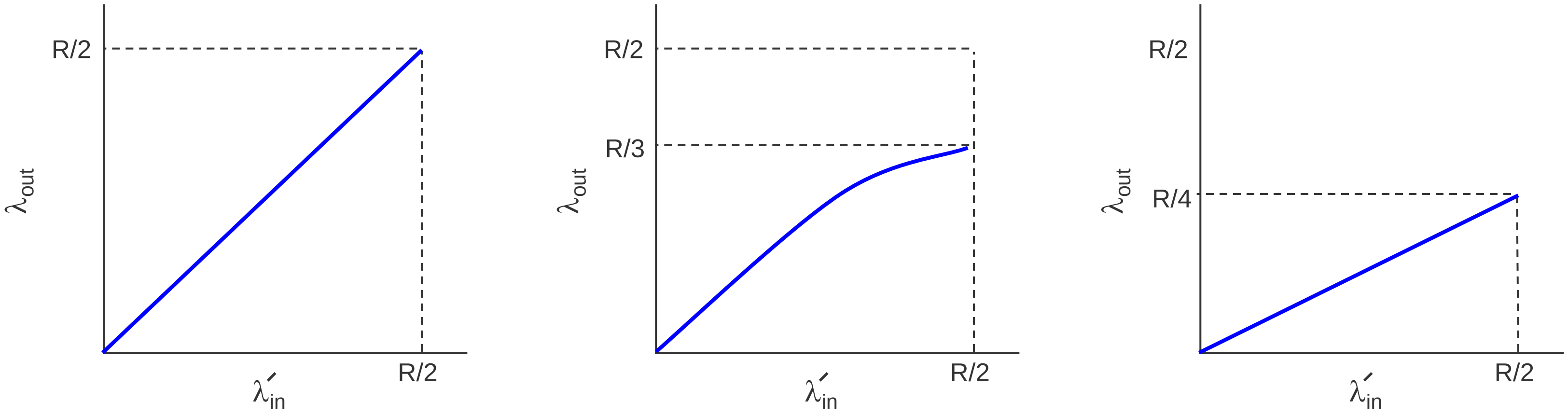
Causes/costs of congestion: Scenario 2

- One router, finite buffers
- Sender retransmission of lost packet
- Offered load λ'_{in}: Original data + retransmitted data


Host A
λ_{in} : original data
λ'_{in} : original data, plus retransmitted data
λ_{out}
Host B
Finite shared output link buffers
Capacity of link R

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}

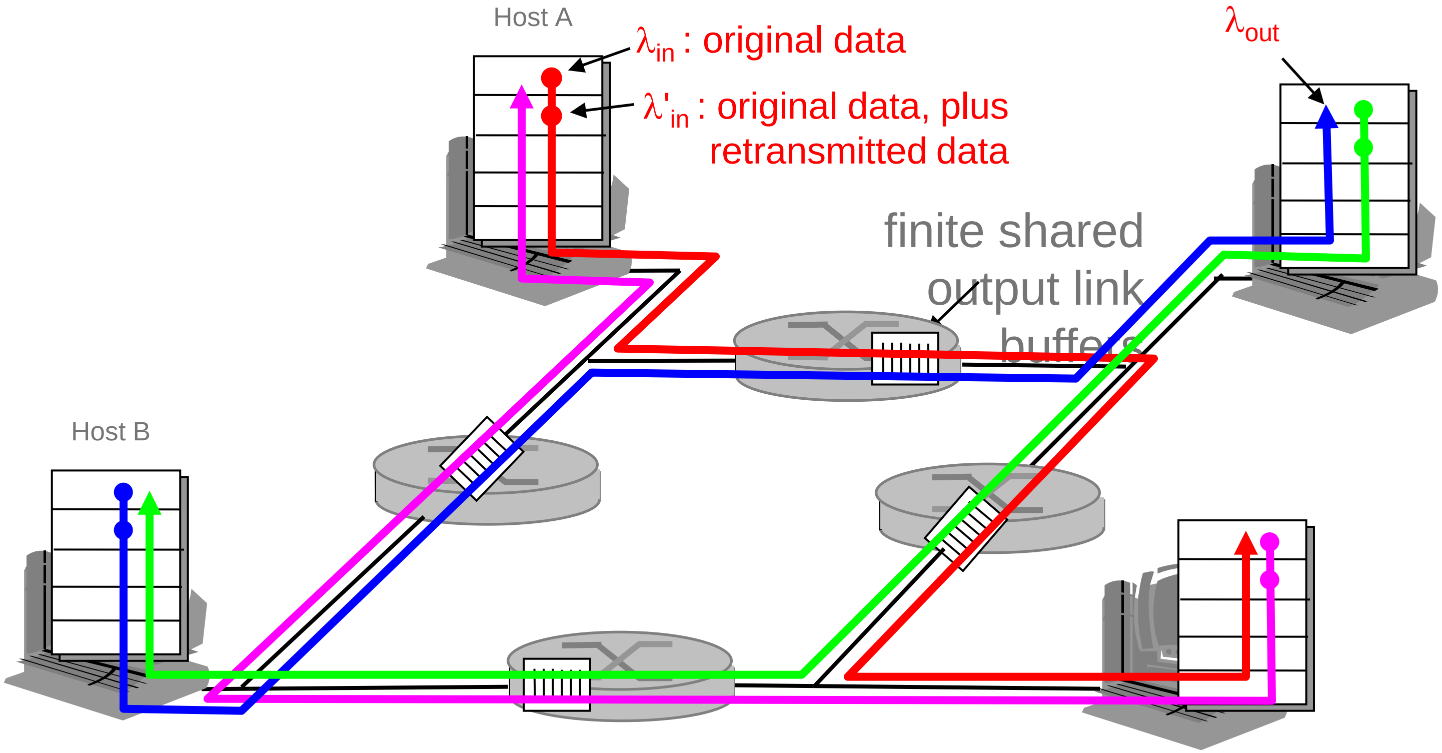


- "Costs" of congestion:
  - More work (retrans) for given "goodput"
  - Unneeded retransmissions: link carries multiple copies of packet

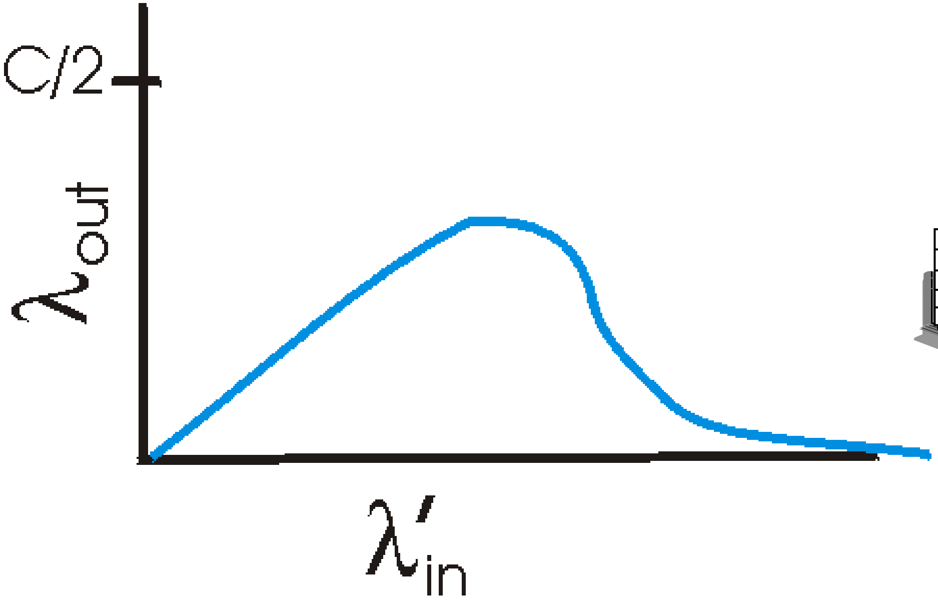
Causes/costs of congestion: Scenario 3

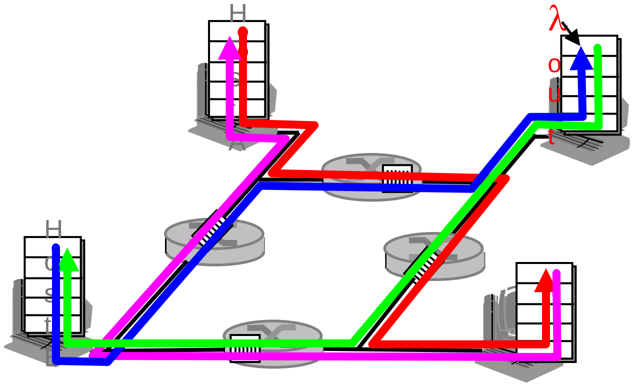
- Four senders
- Multi-hop paths
- Timeout/retransmit

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


Host A
λ_{in} : original data
λ'_{in} : original data, plus retransmitted data
λ_{out}
finite shared output link buffers
Host B

Causes/costs of congestion: Scenario 3





- Another “cost” of congestion:
  - When packet dropped, any “upstream transmission capacity used for that packet was wasted!

Approaches towards congestion control

Two broad approaches towards congestion control:

- End-end congestion control:
  - No explicit feedback from network
  - Congestion inferred from end-system observed loss, delay
  - Approach taken by TCP

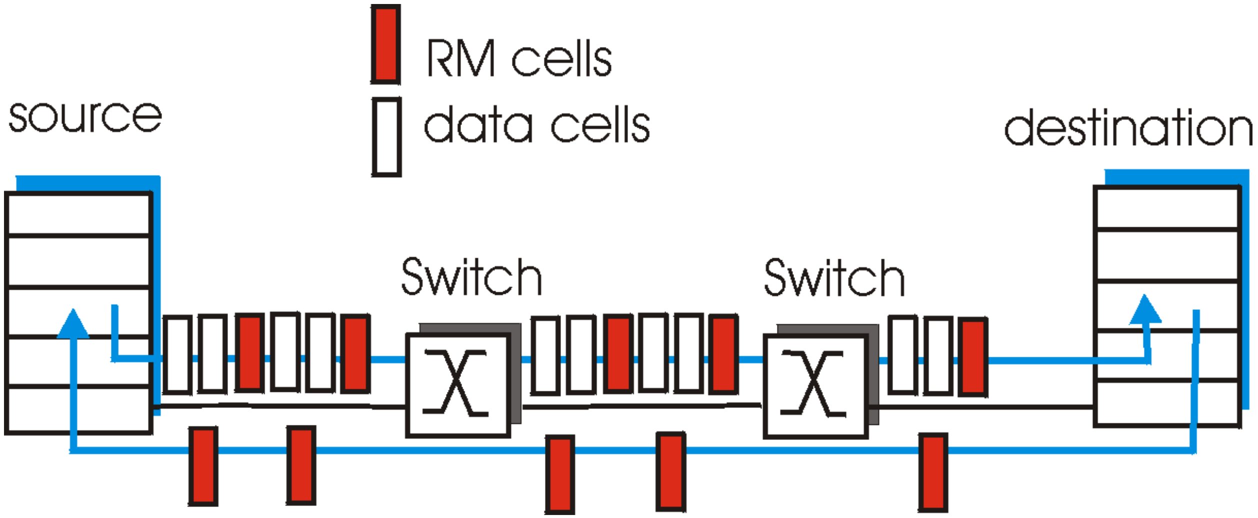
- Network-assisted congestion control:
  - Routers provide feedback to end systems
    - Single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
    - Explicit rate sender should send at

Case study: ATM ABR congestion control

- ABR: available bit rate:
  - "elastic service"
  - If sender's path "underloaded":
    - Sender should use available bandwidth
  - If sender's path congested:
    - Sender throttled to minimum guaranteed rate

- RM (resource management) cells:
  - Sent by sender, interspersed with data cells
  - Bits in RM cell set by switches ("network-assisted")
    - NI bit: no increase in rate (mild congestion)
    - CI bit: congestion indication
  - RM cells returned to sender by receiver, with bits intact

Case study: ATM ABR congestion control



- Two-byte ER (explicit rate) field in RM cell
  - Congested switch may lower ER value in cell
  - Sender’s send rate thus maximum supportable rate on path
- EFCI bit in data cells: set to 1 in congested switch
  - If data cell preceding RM cell has EFCI set, sender sets CI bit in returned RM cell

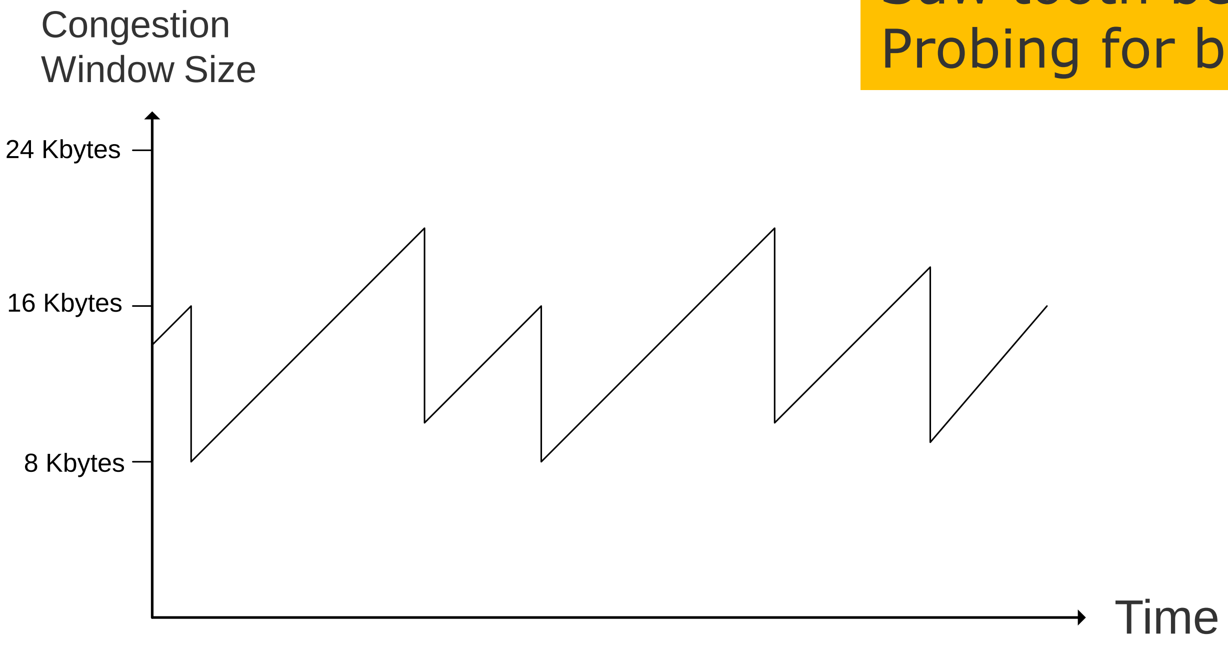
Transmission Control Protocol (TCP)

Congestion control

TCP congestion control: additive increase, multiplicative decrease

- Approach: increase transmission rate (window size), probing for usable bandwidth, until loss occurs
  - Additive Increase: increase **CongWin** by 1 MSS every RTT until loss detected
  - Multiplicative Decrease: cut **CongWin** in half after loss
  - ➧ Additive Increase Multiplicative Decrease (AIMD)

Saw tooth behavior: Probing for bandwidth



TCP Congestion Control: Details

- Sender limits transmission:

LastByteSent - LastByteAcked ≤ CongWin

- Roughly:

$$\boxed{\text{rate} = \frac{\text{CongWin}}{\text{RTT}} \text{ Bytes/sec}}$$

- CongWin is dynamic, function of perceived network congestion

- How does sender perceive congestion?

  Loss event = Timeout **or**
  3 duplicate acks

- TCP sender reduces rate (CongWin) after loss event

- Three mechanisms:
  - AIMD
  - Slow start
  - Conservative after timeout events

TCP Slow Start

- When connection begins, CongWin = 1 MSS
  - Example: MSS = 500 bytes & RTT = 200 msec
  - Initial rate = 20 kbit/s
- Available bandwidth may be >> MSS/RTT
  - Desirable to quickly ramp up to respectable rate
- When connection begins, increase rate exponentially fast until first loss event

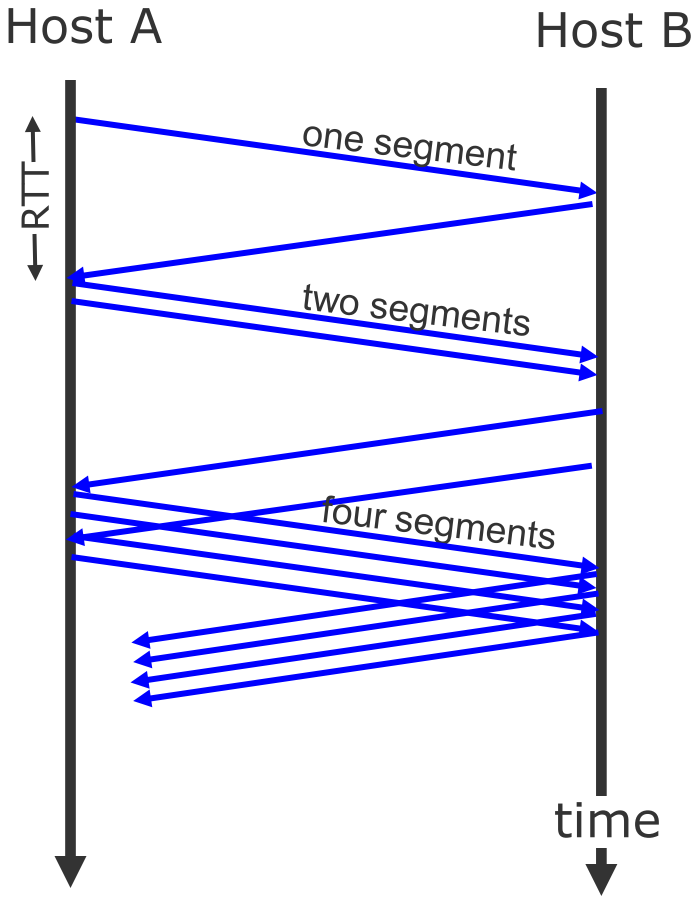
TCP Slow Start

- Each sender maintains two windows for the number of bytes which can be sent:
    - Flow Control Window: granted receiver buffer
    - Congestion Window: “network granted” capacity (**cwnd**)
- Minimum of both windows is the maximum number of bytes that can be sent
- With connection establishment, the sender initializes the congestion window to one **maximum segment size (MSS)**
    - MSS is the maximum number of application data that can be send in one segment!!!
- A segment with MSS bytes of application data is sent
- **Slow Start Algorithm**
    - If an acknowledgement arrives before timeout, double the congestion window, otherwise reset it to the initial value. Thus a “grope” takes place up to the transmission capacity.
    - Enlargement stops with reaching the flow control window
    - Refinement by introduction of a threshold value **ssthresh**
        - At the beginning 64 Kbyte
        - Only linear enlargement by one maximum segment size (**Congestion Avoidance**)
        - With a timeout the threshold value is put back **to half of the maximum window size** reached before

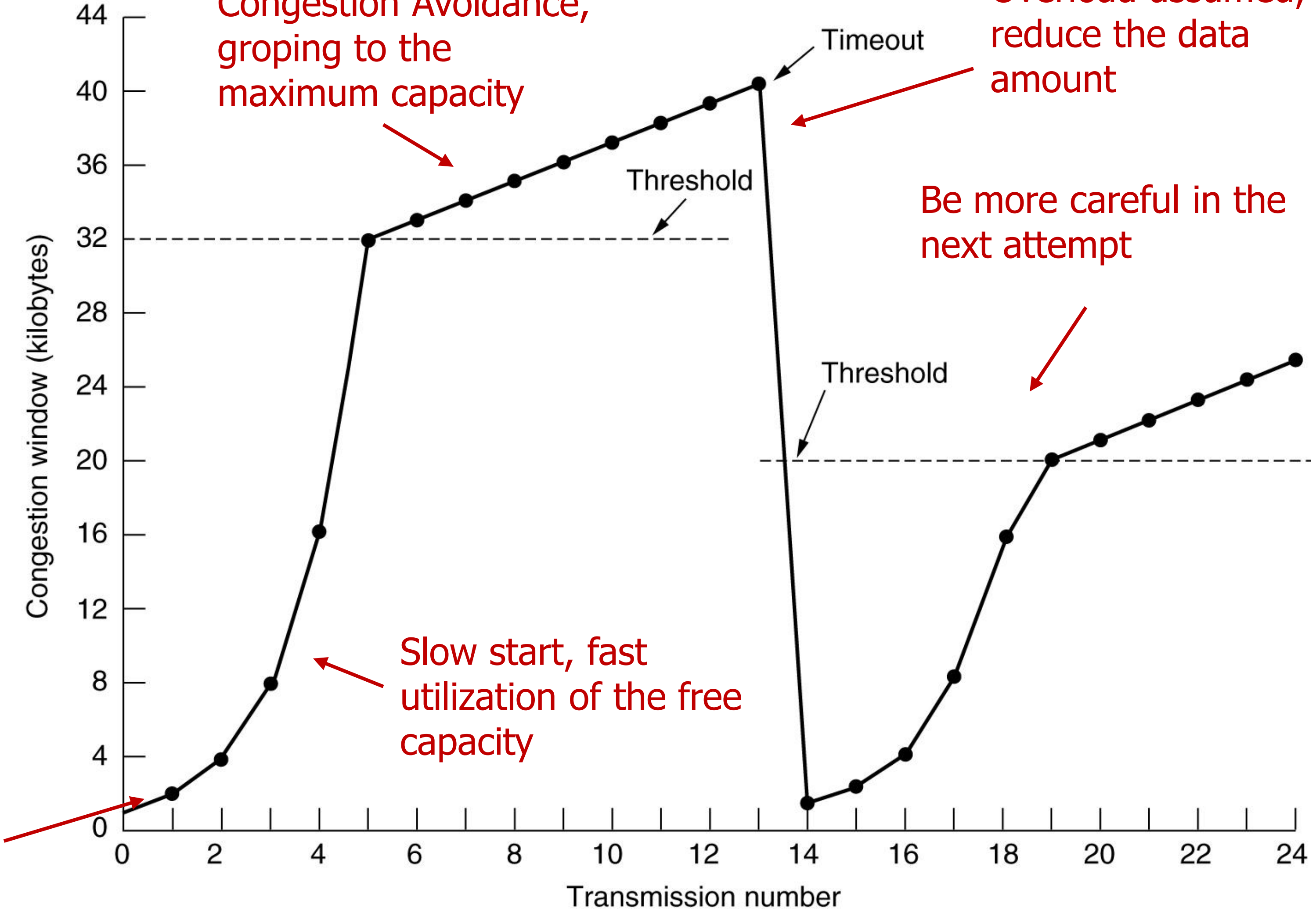
TCP Slow Start

- When connection begins, increase rate exponentially until first loss event:
  - Double `CongWin` every RTT
  - Done by incrementing `CongWin` for every ACK received

- Summary:
  - Initial rate is slow but ramps up exponentially fast



TCP Slow Start



Congestion Avoidance, groping to the maximum capacity

Overload assumed, reduce the data amount

Be more careful in the next attempt

Slow start, fast utilization of the free capacity

Start with one segment of MSS size

Refinement: Inferring loss

- After 3 dup ACKs:
  - CongWin is cut in half
  - Window then grows linearly
- But after timeout event:
  - CongWin instead set to 1 MSS;
  - Window then grows exponentially
  - To a threshold, then grows linearly

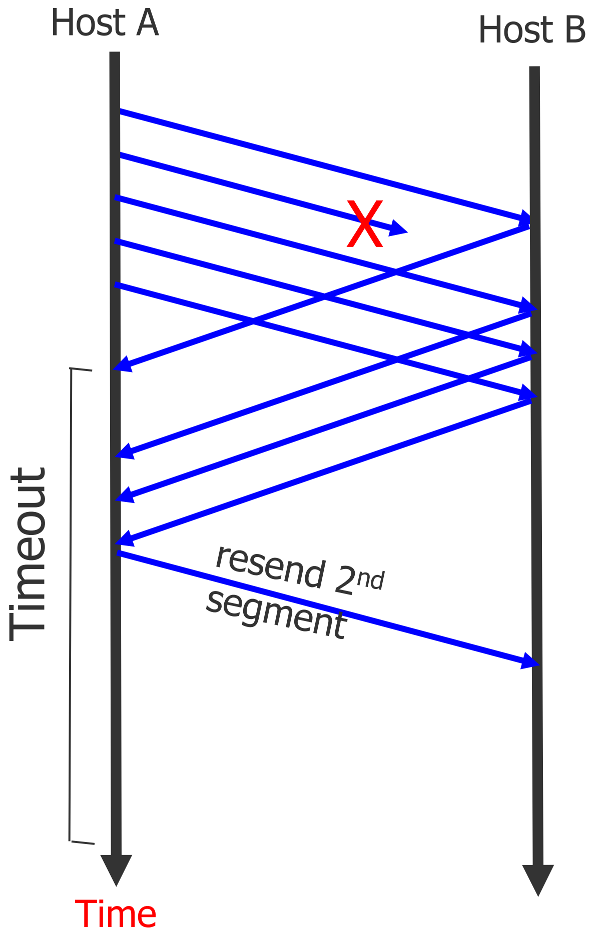
- Philosophy
  - 3 dup ACKs indicates network capable of delivering some segments
  - Timeout indicates a “more alarming” congestion scenario

Fast Retransmit and Fast Recovery

- Slow Start is not well suited when only a single packet is lost…
  - Time-out period often relatively long ➡ Long delay before resending lost packet
  - E.g. short interference on a wireless link

- **Fast Retransmit**
  - The **receiver** sends a duplicate ACK immediately when an out-of-order segment arrives
  - When the **sender** has received 3 duplicate ACKs, it retransmits the missing segment.
    - Hopefully, the acknowledgement for the repeated segment arrives before a timeout

Fast Retransmit

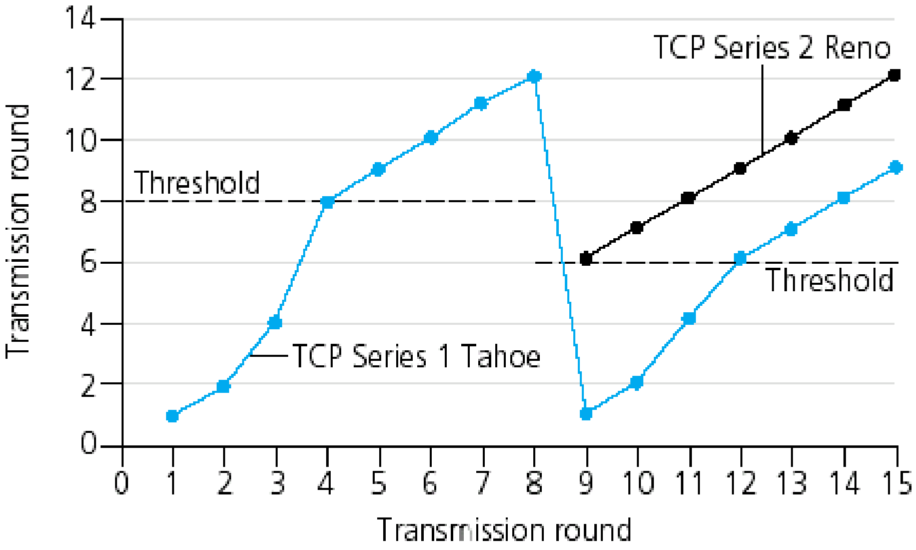


Resending a segment after triple duplicate ACK

Refinement

- Questions
  - Q: When should the exponential increase switch to linear?
  - A: When CongWin gets to 1/2 of its value before timeout.

- Implementation:
  - Variable Threshold
  - At loss event, Threshold is set to 1/2 of CongWin just before loss event



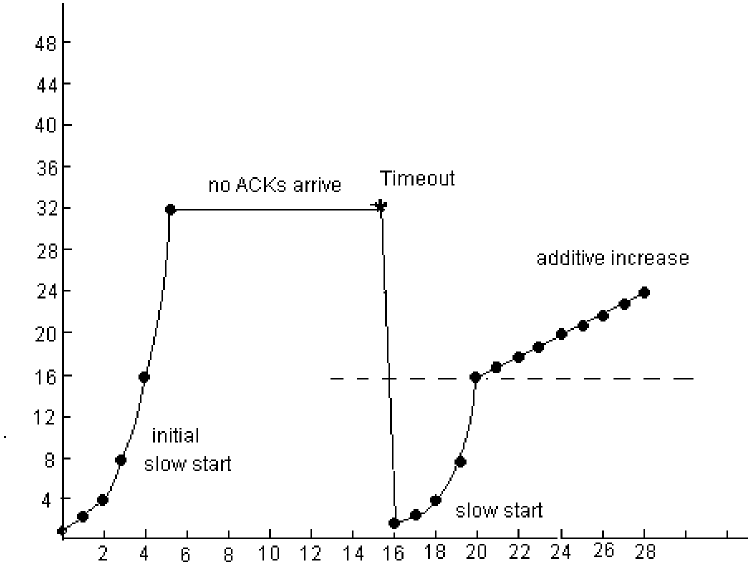
Fast Retransmit and Fast Recovery

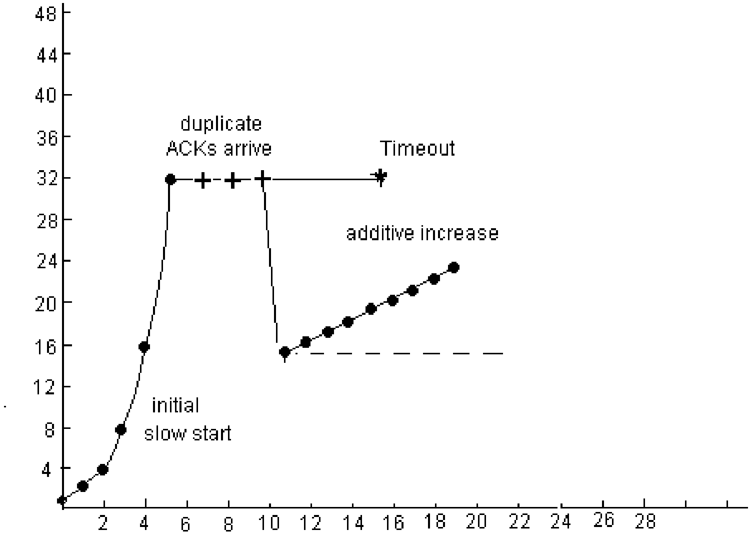
- Fast Retransmit has to be enhanced to be useful in overload situations…
- **Fast Recovery**
  - When the third ACK is received, reduce

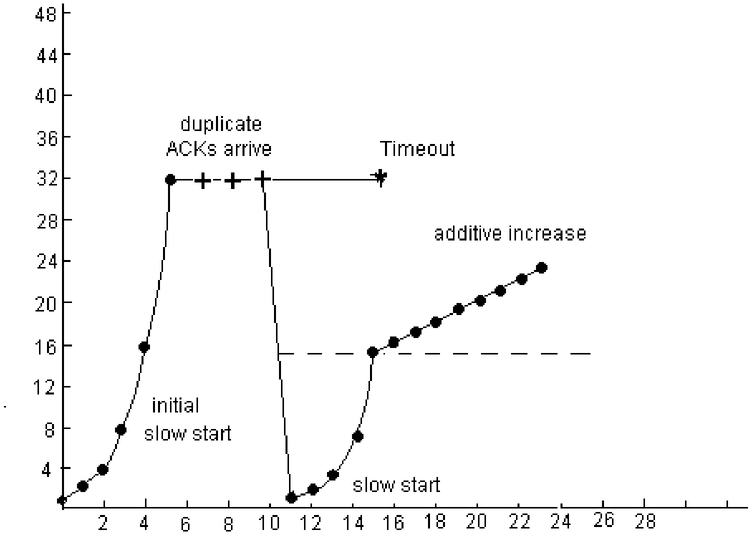
    ssthresh = max(ssthresh/2, 2×MSS)
  - Retransmit the missing segment, set

    cwnd = ssthresh + 3*MSS
  - For each more duplicated ACK, increment **cwnd** by MSS
  - This reduces **cwnd** by the amount of lost segments to adapt to the network situation
  - If the new **cwnd** (and the receiver's buffer) allows, send a segment
  - When the next (normal) ACK arrives, set **cwnd** to **ssthresh** to go on normally

Fast Retransmit and Fast Recovery







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.

TCP sender congestion control

State	Event	TCP Sender Action	Commentary
Slow Start (SS)	ACK receipt for previously unacked data	CongWin = CongWin + MSS, If (CongWin > Threshold) set state to “Congestion Avoidance”	Resulting in a doubling of CongWin every RTT
Congestion Avoidance (CA)	ACK receipt for previously unacked data	CongWin = CongWin+MSS * (MSS/CongWin)	Additive increase, resulting in increase of CongWin by 1 MSS every RTT

SS or CA	Loss event detected by triple duplicate ACK	Threshold = CongWin/2,
		CongWin = Threshold,
		Set state to “Congestion Avoidance”
			Fast recovery, implementing multiplicative decrease. CongWin will not drop below 1 MSS.

SS or CA	Timeout	Threshold = CongWin/2,
		CongWin = 1 MSS,
		Set state to “Slow Start”
			Enter slow start

SS or CA	Duplicate ACK	Increment duplicate ACK count for segment being acked	CongWin and Threshold not changed

Transmission Control Protocol (TCP)

TCP Throughput

TCP throughput

- What’s the average throughout of TCP as a function of window size and RTT?
  - 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 throughout: $(W/RTT+W/2RTT)/2$ ➡ $0.75W/RTT$

TCP Futures: TCP over “long, fat pipes”

- Example: 1500 byte segments, 100ms RTT, want 10 Gbit/s throughput
- Requires window size W = 83,333 in-flight segments
- Throughput in terms of loss rate:

$$\frac{1.22 \cdot MSS}{RTT\sqrt{L}}$$

➡L = $2 \cdot 10^{-10}$

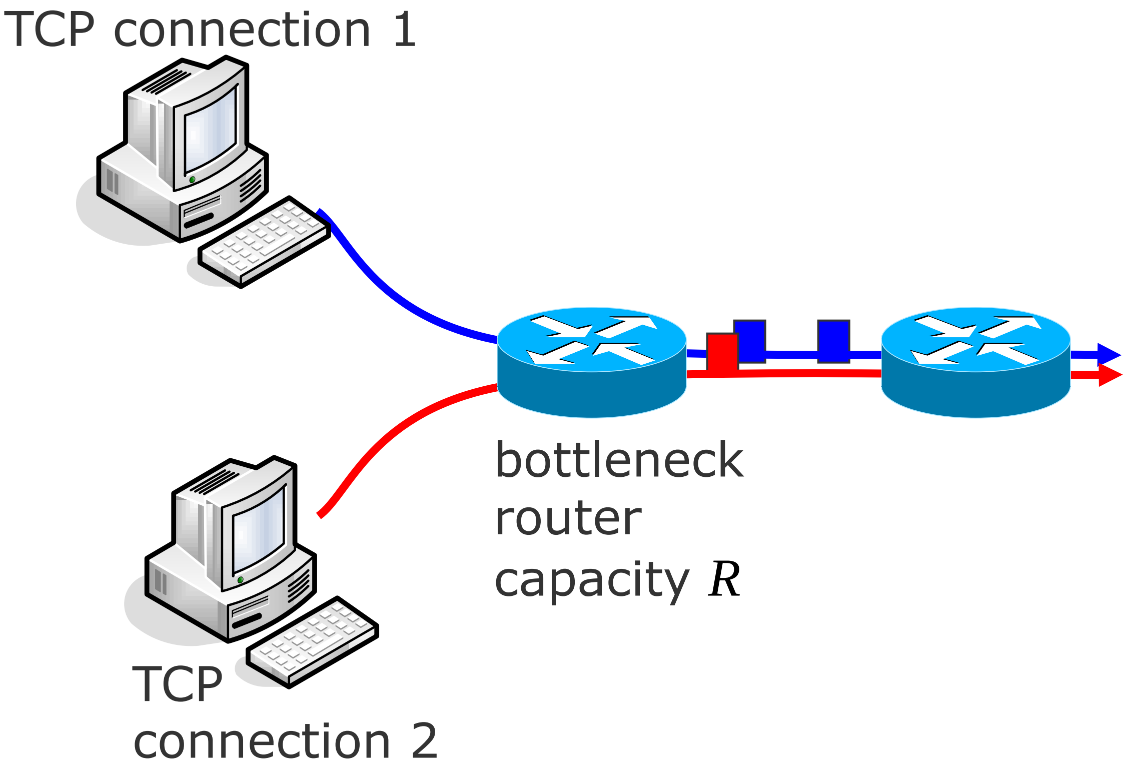
- New versions of TCP for high-speed

Transmission Control Protocol (TCP)

Fairness

TCP Fairness

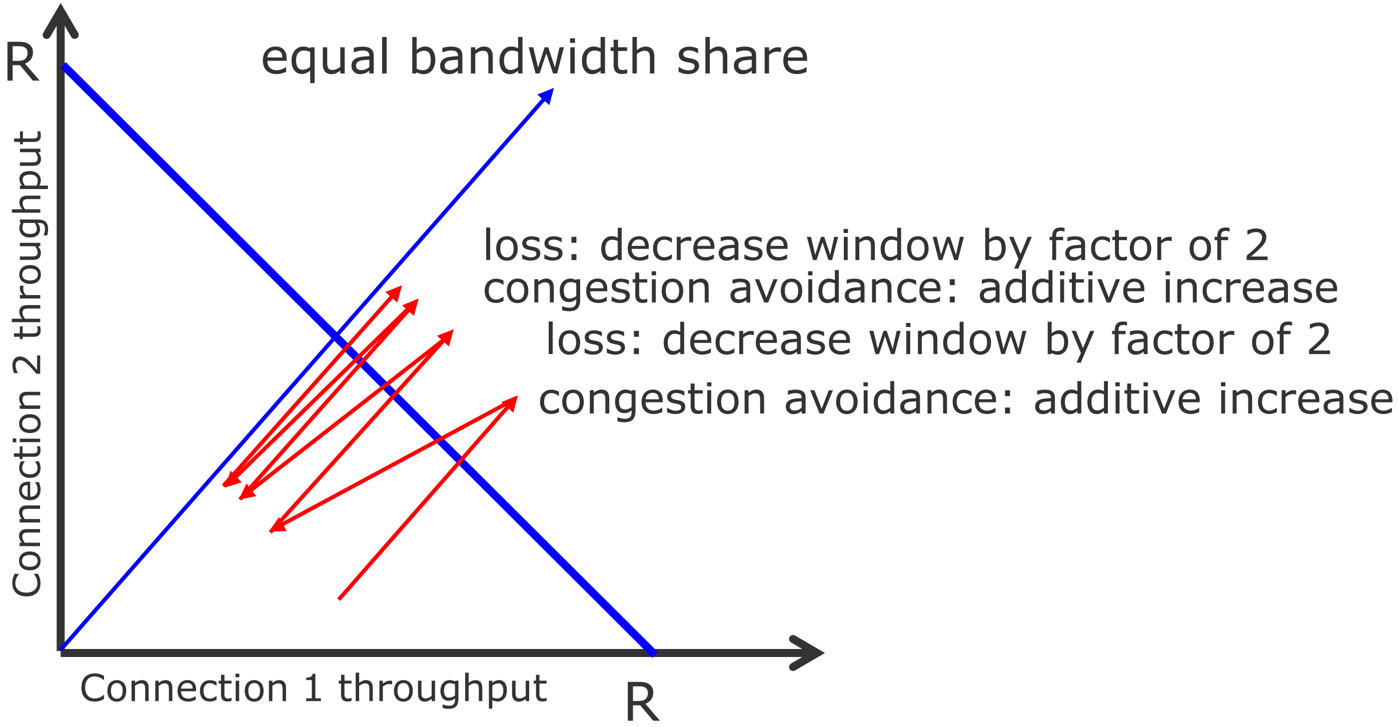
- Fairness goal: if k TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K



Why is TCP fair?

Two competing sessions:

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


R
equal bandwidth share
Connection 2 throughput
loss: decrease window by factor of 2
congestion avoidance: additive increase
loss: decrease window by factor of 2
congestion avoidance: additive increase
Connection 1 throughput
R

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 friendlyness
    - See DCCP

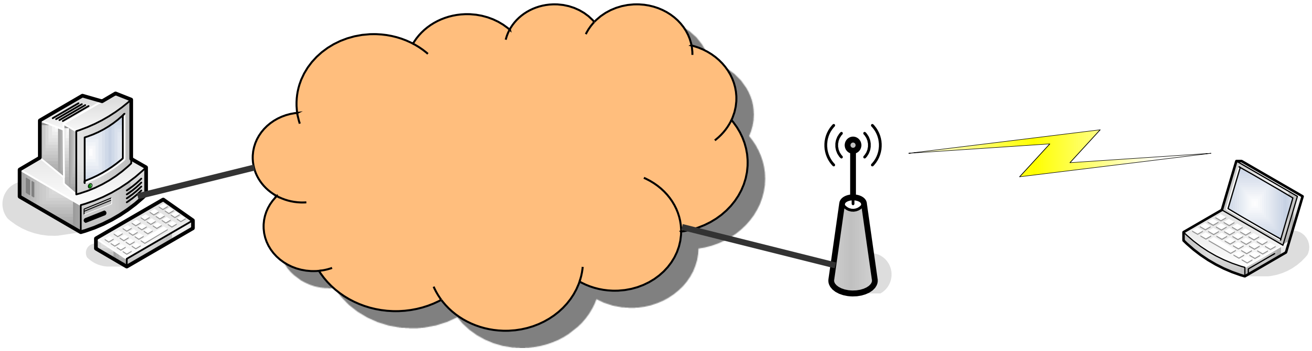
- Fairness and parallel TCP connections
  - nothing prevents app from opening parallel connections between 2 hosts.
  - Web browsers do this
  - Example: link of rate R supporting 9 connections;
    - new app asks for 1 TCP, gets rate R/10
    - new app asks for 11 TCPs, gets R/2

Transmission Control Protocol (TCP)

TCP in Wireless Networks

TCP in Wireless Networks

- Theoretically the transport layer protocol should be independent of lower layers
  - But TCP is optimized for wired networks
  - TCP assumes that packet loss is due to congestion in the network
- In wireless networks packet loss occurs due to the medium
  - Thus, performance of TCP in wireless networks is poor
  - Many approaches to solve the performance problem
- Indirect TCP (as an example)
  - The end-to-end connection is broken in two parts

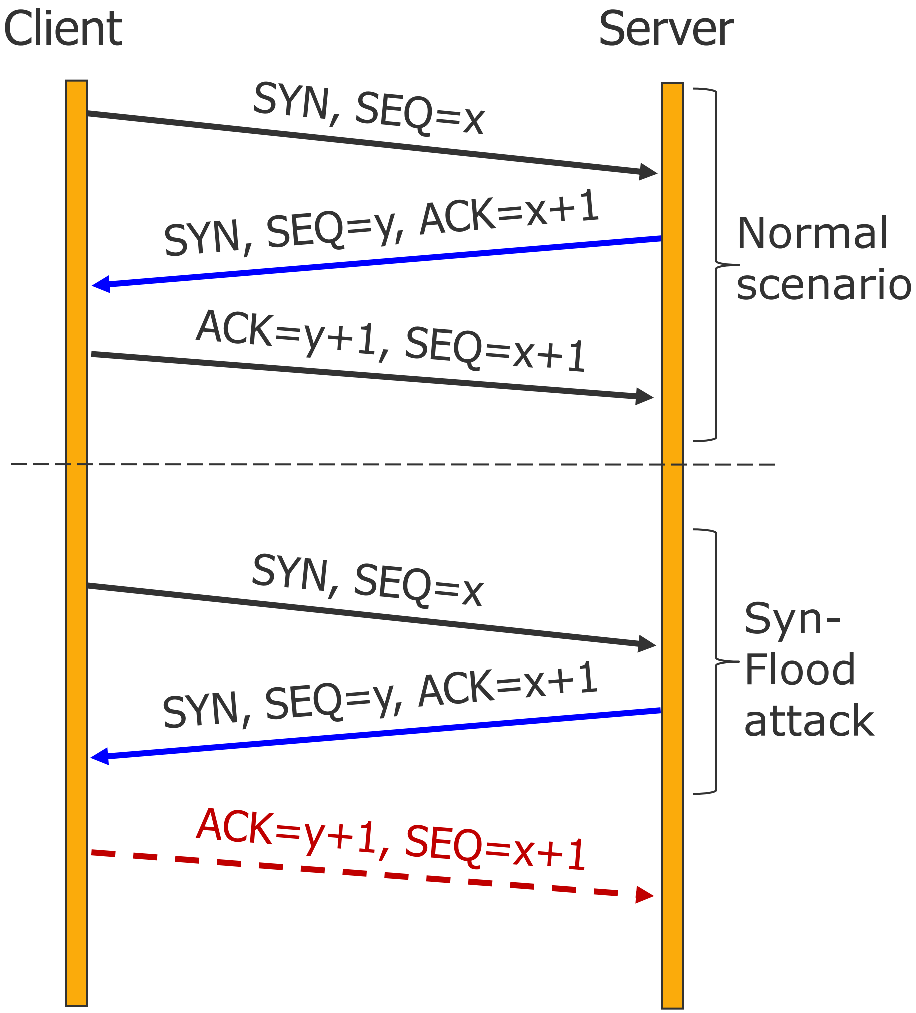


Transmission Control Protocol (TCP)

TCP and Security

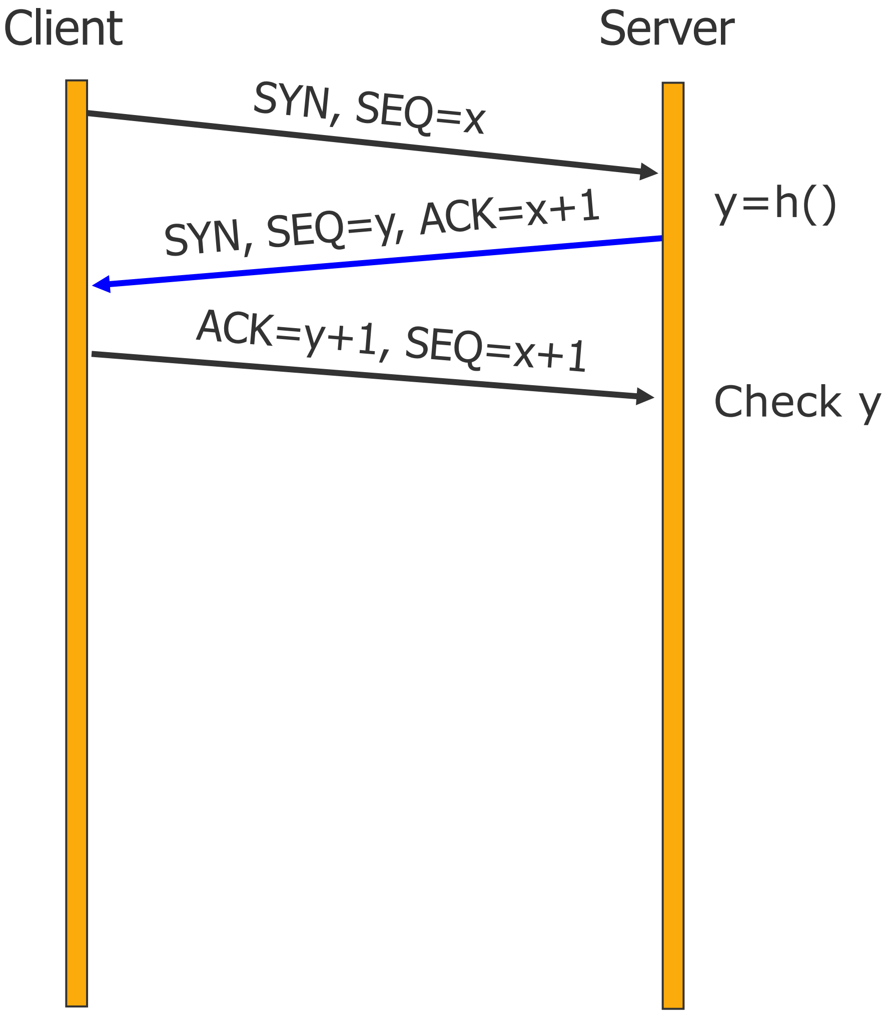
TCP and Security: SYN-Flood

- During connection establishment the client does not finish the Three Way Handshake procedure
  - A half open connection
  - Operating system reserves resources



TCP and Security: SYN-Flood

- Countermeasure: SYN cookies
  - Server does not creates a half-open connection
  - Server computes an initial sequence number y based on a hash function
    - This is the cookie
  - When client returns with ACK the server recomputes the hash function and checks it
    - For legitimate connection the check will be successful



Some Tools

Some Tools / netstat

- netstat: Displays protocol statistics and TCP/IP network connections
  - Flags (some examples)
    - n: display IP addresses
    - b: display executable
    - r: routing table

```
x:₩>netstat -n

Active Connections

  Proto  Local Address          Foreign Address        State
  TCP    127.0.0.1:3055         127.0.0.1:3056         ESTABLISHED
  TCP    127.0.0.1:3056         127.0.0.1:3055         ESTABLISHED
  TCP    160.45.114.21:2114     130.133.8.114:80       CLOSE_WAIT
  TCP    160.45.114.21:3027     160.45.113.73:1025     ESTABLISHED
  TCP    160.45.114.21:3029     160.45.113.73:1025     ESTABLISHED
  TCP    160.45.114.21:3043     160.45.113.89:1200     ESTABLISHED
  TCP    160.45.114.21:3362     207.46.108.69:1863     ESTABLISHED
  TCP    160.45.114.21:3704     130.133.8.114:80       CLOSE_WAIT
  TCP    160.45.114.21:3705     130.133.8.114:80       CLOSE_WAIT
  TCP    160.45.114.21:3907     160.45.114.28:139      ESTABLISHED
  TCP    160.45.114.21:3916     160.45.113.100:445     ESTABLISHED
```

Some Tools

- TTCP – Test TCP
  - A tool to measure throughput in a network via TCP and UDP
- Iperf
  - Similar tool to test network throughput

Stream Control Transmission Protocol (SCTP)

Stream Control Transmission Protocol (SCTP)

- Key motivation
  - Reliable transport of messages via IP networks (e.g. SS7 signaling)
  - TCP too strict in-order delivery (danger of head-of-line-blocking)
  - Limitations of sockets wrt. highly-available data transfer using multi-homed hosts
  - Vulnerability of TCP wrt. DoS attacks such as SYN flooding

- Key features
  - Reliable, connection-oriented transport protocol, multi-streaming, multi-homing, heartbeats, 4-way-handshake, TCP friendly, certain resistance to flooding/masquerade attacks, message oriented

- RFCs (originally developed by the IETF SIGTRAN working group)
  - RFC 4960 (was 2960, updated by 3309, 6096, 6335)
    - Specification of the protocol, packet formats, options, features
  - RFC 3286
    - High-level introduction to SCTP
  - And some more, like RFC 5062 Security Attacks against SCTP…

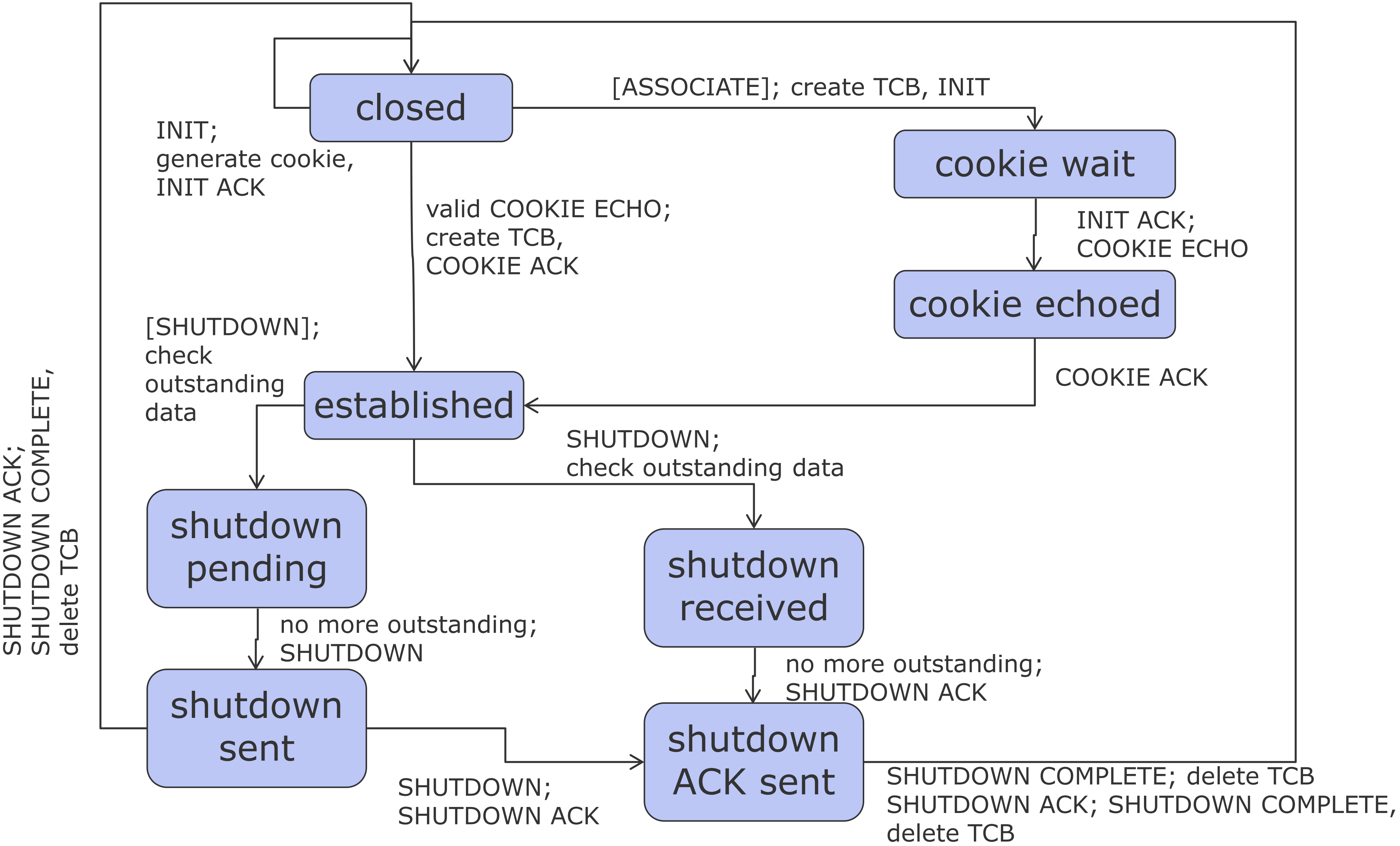
SCTP Multi-Streaming

- Allows partitioning of data into multiple streams within one *association*
- Stream are independent wrt. sequenced delivery
  - i.e. loss in any one stream will only affect delivery within that stream
- Allows for partial ordering of e.g. objects of a web page
  - Each object is assigned to a stream, order of object delivery irrelevant
  - However, all streams are subjected to a common flow and congestion control mechanism
- Two sequence numbers used
  - Transmission Sequence Number
    - Governs transmission of messages, detection of message loss
  - Stream Identification/Stream Sequence Number
    - Determines the sequence of delivery of received data
  - This allows separating streams affected from message loss from unaffected streams

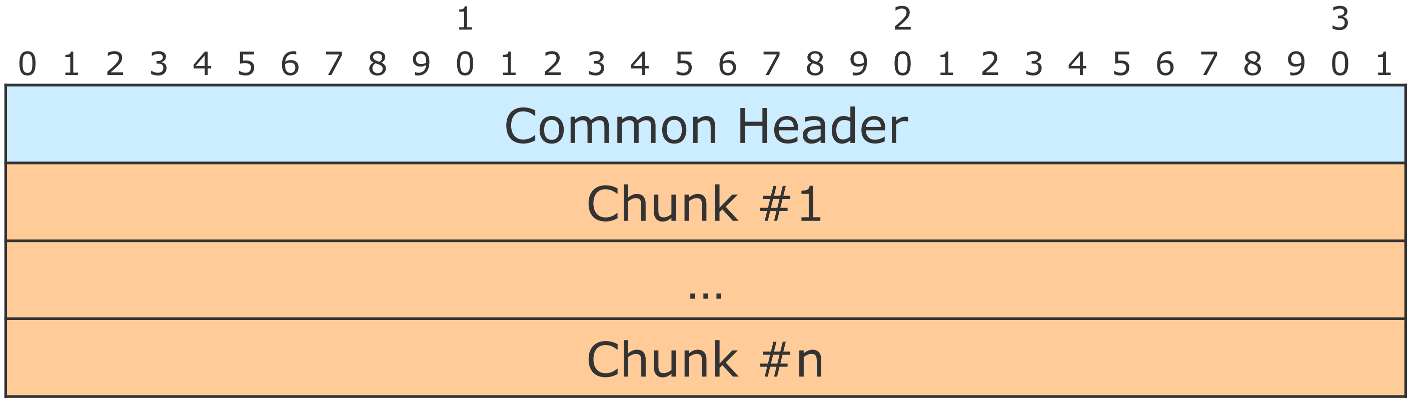
SCTP Multi-Homing

- Ability for a single endpoint to support multiple IP addresses
- Used for redundancy (not for load sharing up to now)
  - E.g. laptop could be connected via LAN, WLAN and UMTS to the Internet
  - Failure of one network will not cause a failure of the association
- One address chosen as primary IP address
  - Destination for all DATA chunks for normal transmission
  - Retransmitted DATA chunks use alternate address(es)
  - Continued failure of primary address results in transmitting all DATA chunks to alternate address (until heartbeats can reestablish primary address)

SCTP State Diagram (without timers, ABORT)


closed
[ASSOCIATE]; create TCB, INIT
cookie wait
INIT; generate cookie, INIT ACK
valid COOKIE ECHO; create TCB, COOKIE ACK
INIT ACK; COOKIE ECHO
cookie echoed
[SHUTDOWN]; check outstanding data
COOKIE ACK
established
SHUTDOWN; check outstanding data
shutdown pending
shutdown received
no more outstanding; SHUTDOWN
no more outstanding; SHUTDOWN ACK
shutdown sent
shutdown ACK sent
SHUTDOWN; SHUTDOWN ACK
SHUTDOWN COMPLETE; delete TCB
SHUTDOWN ACK; SHUTDOWN COMPLETE, delete TCB
SHUTDOWN ACK; SHUTDOWN COMPLETE, delete TCB

SCTP Packet Format



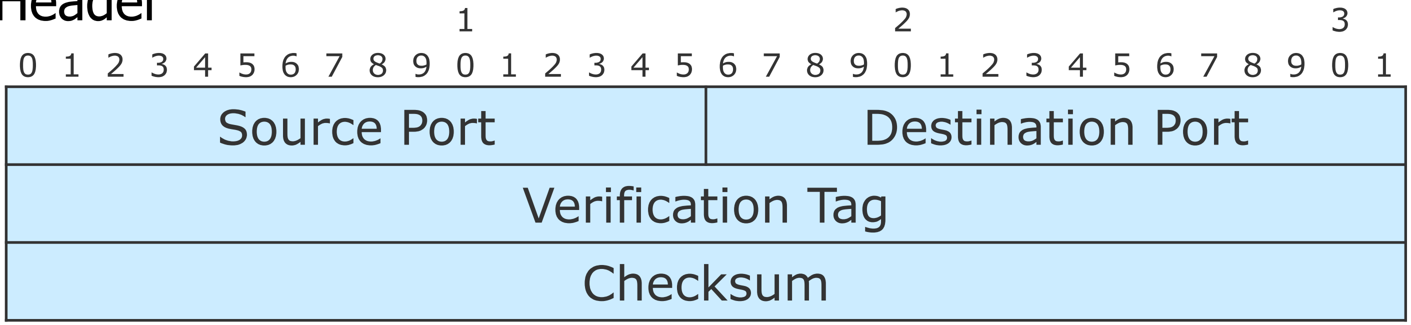
Common Header

Chunk #1

…

Chunk #n

- Common Header

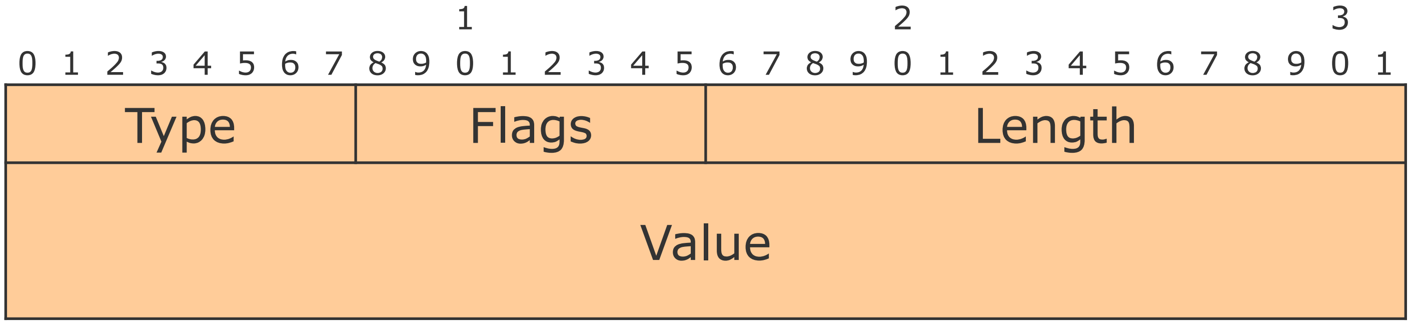


Source Port | Destination Port

Verification Tag

Checksum

- Chunk



Type | Flags | Length

Value

SCTP Common Header

- Ports
  - 16 bit as used in TCP, UDP

- Verification Tag
  - Validation of the sender, set to the value of the Initiate Tag received from the peer during association initialization
  - Set to 0 in packets with INIT chunk
  - See RFC for use during SHUTDOWN and ABORT

- Checksum
  - CRC32c checksum algorithm using the polynomial $x^{32}+x^{28}+x^{27}+x^{26}+x^{25}+x^{23}+x^{22}+x^{20}+x^{19}+x^{18}+x^{14}+x^{13}+x^{11}+x^{10}+x^{9}+x^{8}+x^{6}+1$
  - Calculated over the whole SCTP packet including common header (with checksum initialized to 0) and all chunks
  - Invalid packets are typically silently discarded

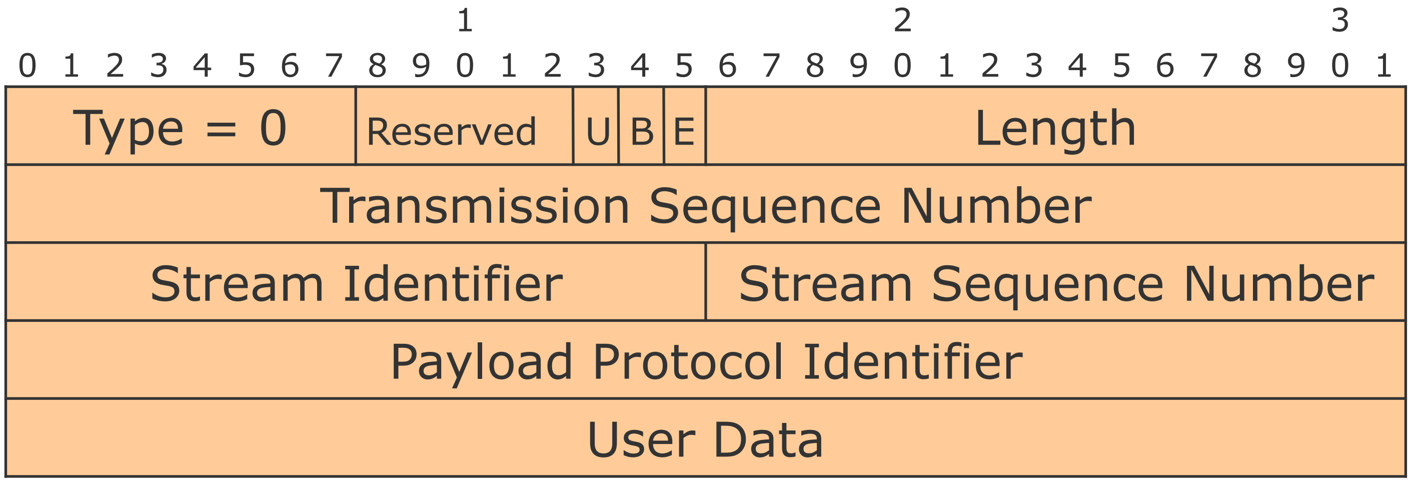
SCTP Chunks

- Type (Examples)
  - 0: Payload Data (DATA)
  - 1: Initiation (INIT)
  - 2: Initiation Acknowledgement (INIT ACK)
  - 3: Selective Acknowledgement (SACK)
  - 4: Heartbeat Request (HEARTBEAT)
  - 5: Heartbeat Acknowledgement (HEARTBEAT ACK)
  - 6: Abort (ABORT)
  - 7: Shutdown (SHUTDOWN)
  - 8: Shutdown Acknowledgement (SHUTDOWN ACK)
  - 9: Operation Error (ERROR)
  - 10: State Cookie (COOKIE ECHO)
  - 11: Cookie Acknowledgement (COOKIE ACK)
  - 12: Reserved for Explicit Congestion Notification Echo (ECNE)
  - 13: Reserved for Congestion Window Reduced (CWR)
  - 14: Shutdown Complete (SHUTDOWN COMPLETE)
- Flags depend on type
- Length
  - Overall length of chunk in bytes
- Value
  - Actual data of the chunk

SCTP DATA Chunk

- U: Indicates unordered DATA chunk, ignore Stream Sequence Number
-

B	E	
1	0	first fragment of a user message
0	0	middle fragment of a user message
0	1	last fragment of a user message
1	1	Unfragmented user message



SCTP DATA Chunk

- Transmission Sequence Number
  - Sequence number for this DATA chunk
- Stream identifier
  - Identification of data stream to which the following data belongs
- Stream Sequence Number
  - Sequence number of the following user data within the stream
  - When a user message is fragmented all fragments must carry the same stream sequence number
- Payload Identifier
  - Identifies application layer protocol
  - Not used by SCTP but by end or intermediate systems

Datagram Congestion Control Protocol (DCCP)

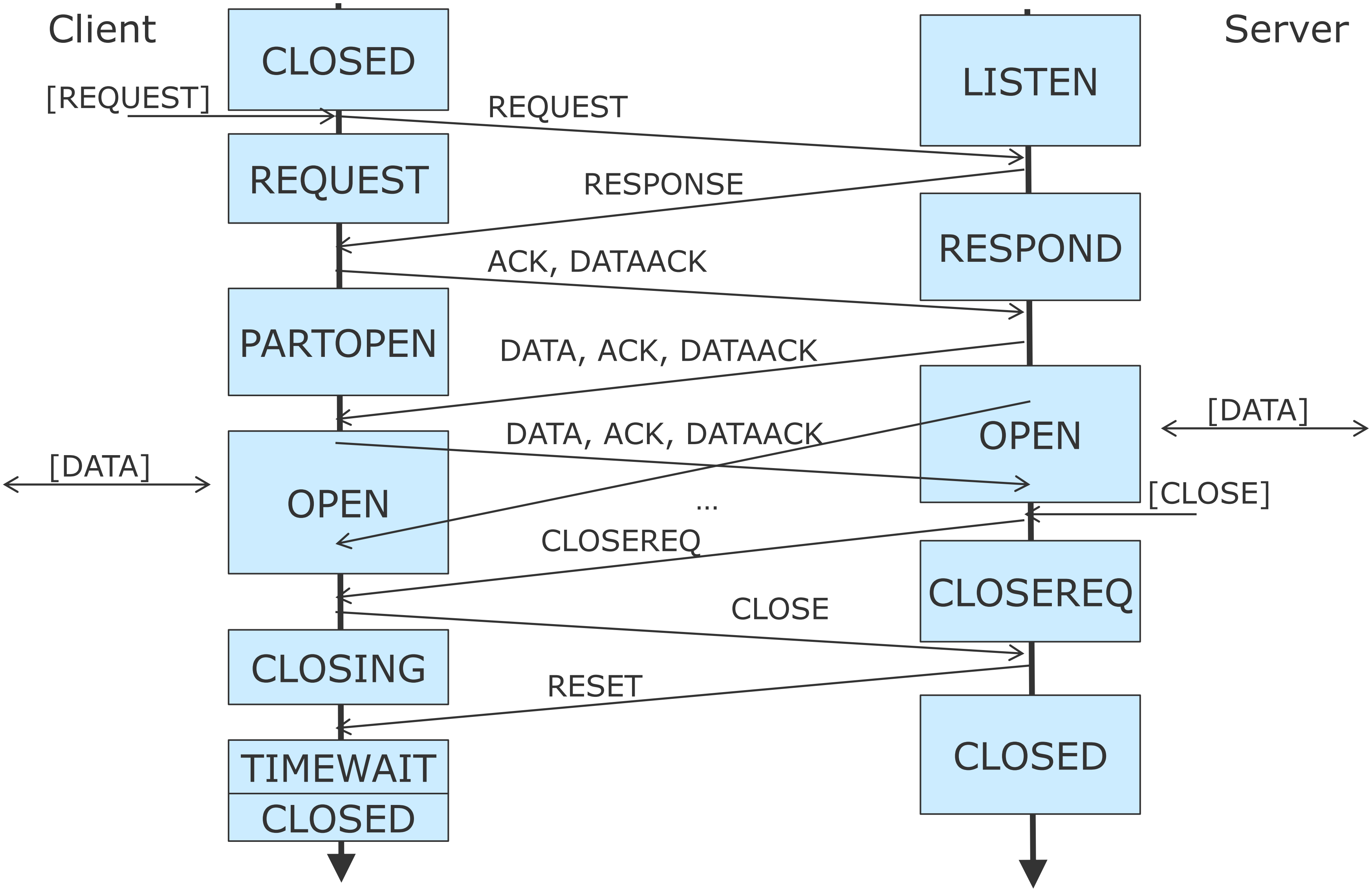
Datagram Congestion Control Protocol (DCCP)

- Key motivation
  - rapid growth of applications using UDP
    - streaming media, on-line games, internet telephony
    - apps preferring timeliness instead of reliability or short start-up delay
  - non TCP-friendly behavior of UDP poses threat to the health of the Internet
  - clean, understandable, low-overhead, minimal protocol

- Key features
  - unreliable flow of datagrams, end-to-end congestion control, simpler firewall traversal, parameter negotiation

- RFCs
  - RFC 4340, updated by 5595, 5596, 6335
  - RFC 4336: Problem Statement for the Datagram Congestion Control Protocol

DCCP Time Sequence Diagram (Incomplete Example)

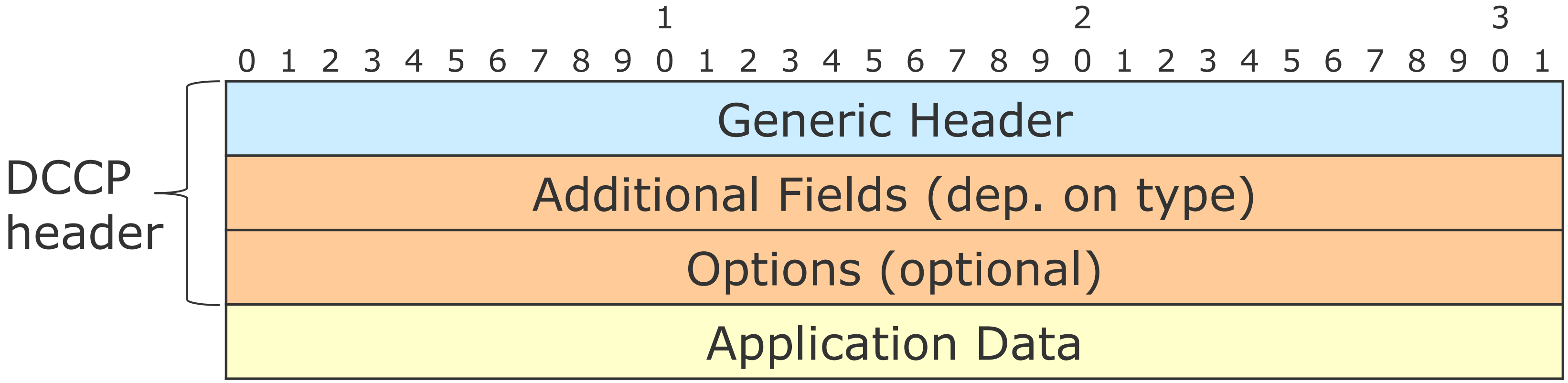


DCCP vs. TCP/UDP

- Choice of congestion control mechanisms, even different per half-connection (eg TCP-like, TCP-friendly Rate Control etc.)
- Feature negotiation mechanism
- Per packet sequence numbers
- Different ACK formats (Still: no reliability! Up to the app to decide!)
- DoS protection (e.g. cookies against flooding)
- Distinguishing of different kinds of loss
  - Corruption, receiver buffer overflow…
- No receive window, simultaneous open, half-closed states…
- …

- Very roughly
  - DCCP = TCP – bytestream semantics - reliability
  - DCCP = UDP + congestion control + handshake + acknowledgements

DCCP Packet Format



1 2 3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1

DCCP header

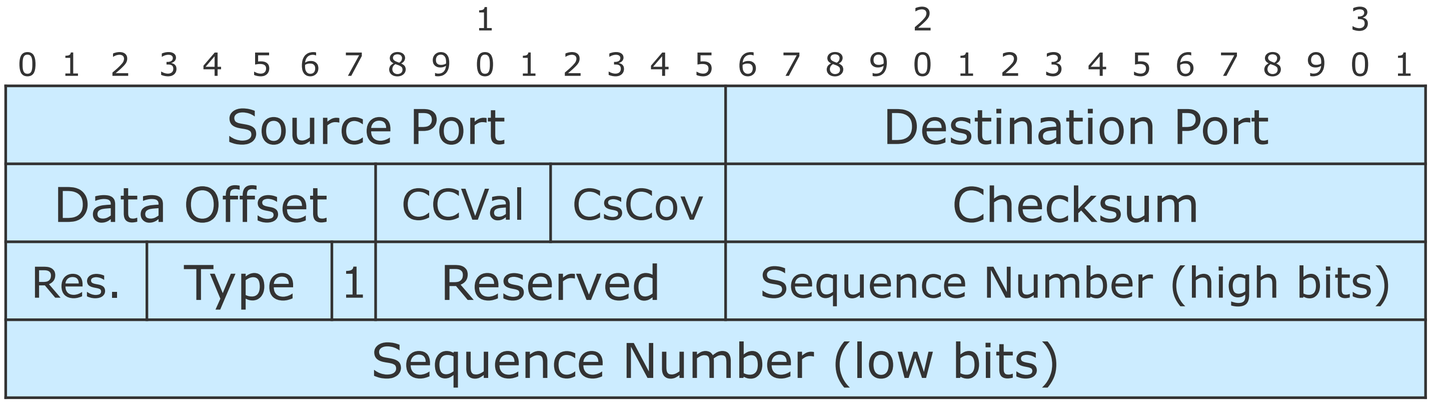
Generic Header

Additional Fields (dep. on type)

Options (optional)

Application Data

- Generic Header
  - 16 byte version, increased protection against wrapping sequence numbers



1 2 3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1

Source Port | Destination Port

Data Offset | CCVal | CsCov | Checksum

Res. | Type | 1 | Reserved | Sequence Number (high bits)

Sequence Number (low bits)

- There is also a shorter 12 byte Generic Header

DCCP Generic Header

- Ports
  - 16 bit as used in TCP, UDP
- Data Offset
  - Start of application data area in 32 bit words
- CCVal
  - Determines congestion control method
- CsCov
  - Determines the checksum coverage
  - Header and options are always included in the checksum
- Checksum
  - Based on TCP/IP checksum algorithm
  - Covers all, some, or none of application data depending on CsCov

DCCP Generic Header

- Reserved/Res
  - All bits set to 0

- Type
  - Type of packet: REQUEST, RESPONSE, DATA, ACK, DATAACK, CLOSEREQ, CLOSE, RESET, SYNC, SYNCACK, reserved (10-15)

- Sequence Number
  - 48 bit (extended sequence number as shown) or 24 bit
  - Counts every packet, including ACKs

DCCP State Diagram a la RFC

```
+---------------------------+    +---------------------------+
|                           v    v                           |
|                        +----------+                        |
|          +-------------+  CLOSED  +------------+           |
|          | passive     +----------+  active    |           |
|          |  open                      open     |           |
|          |                         snd Request |           |
|          v                                     v           |
|     +----------+                          +----------+     |
|     |  LISTEN  |                          | REQUEST  |     |
|     +----+-----+                          +----+-----+     |
|          | rcv Request            rcv Response |           |
|          | snd Response             snd Ack    |           |
|          v                                     v           |
|     +----------+                          +----------+     |
|     | RESPOND  |                          | PARTOPEN |     |
|     +----+-----+                          +----+-----+     |
|          | rcv Ack/DataAck         rcv packet  |           |
|          |                                     |           |
|          |             +----------+            |           |
|          +------------>|   OPEN    |<-----------+           |
|                        +--+-+--+--+                        |
|       server active close | |  |   active close            |
|           snd CloseReq    | |  | or rcv CloseReq           |
|                           | |  |    snd Close              |
|                           | |  |                           |
|     +----------+          | |  |          +----------+     |
|     | CLOSEREQ |<---------+ |  +--------->| CLOSING  |     |
|     +----+-----+            |             +----+-----+     |
|          | rcv Close        |        rcv Reset |           |
|          | snd Reset        |                  |           |
|<---------+                  |                  v           |
|                             |             +----+-----+     |
|                   rcv Close |             | TIMEWAIT |     |
|                   snd Reset |             +----+-----+     |
+-----------------------------+                  |           |
                                                 +-----------+
                                              2MSL timer expires
```

Summary

- The classical TCP/IP reference model has only two protocols on the transport layer
  - UDP for connectionless, unreliable, but lightweight protocol
  - TCP for connection oriented and reliable communication, but really complex
    - Connection establishment
    - Flow control / congestion control
    - Connection termination
    - Fairness

- New applications motivate new transport layer protocols
  - SCTP initially for transporting signaling messages offers multi-stream/multi-home
  - DCCP offers unreliable datagram service, but with congestion avoidance

- Man more protocols exist – but will they be used?
  - MPTCP (Multipath TCP), RUDP (Reliable UDP), UDP Lite, …
  - See http://www.iana.org/assignments/protocol-numbers/protocol-numbers.xml for some more as IPv4 (protocol) and IPv6 (next header) point also to layer 4 protocols