# Previously on Computer Networks

- IPv4 header
- IP address: prefix, CIDR
- NAT
- IPv6
- ICMP
- ARP
- DHCP
- MPLS
- OSPF, RIP, and BGP

# CHAPTER 6:
# THE TRANSPORT LAYER
# (传输层)

- The Transport Service (传输服务)
- Elements of Transport Protocols (传输协议的若干问题)
- Congestion Control (拥塞控制)
- The Internet Transport Protocols (Internet传输协议): UDP
- The Internet Transport Protocols (Internet传输协议): TCP
- Performance Issues (性能问题)*
- Delay-Tolerant Networking (容延网络)*

# THE TRANSPORT SERVICE (传输服务)

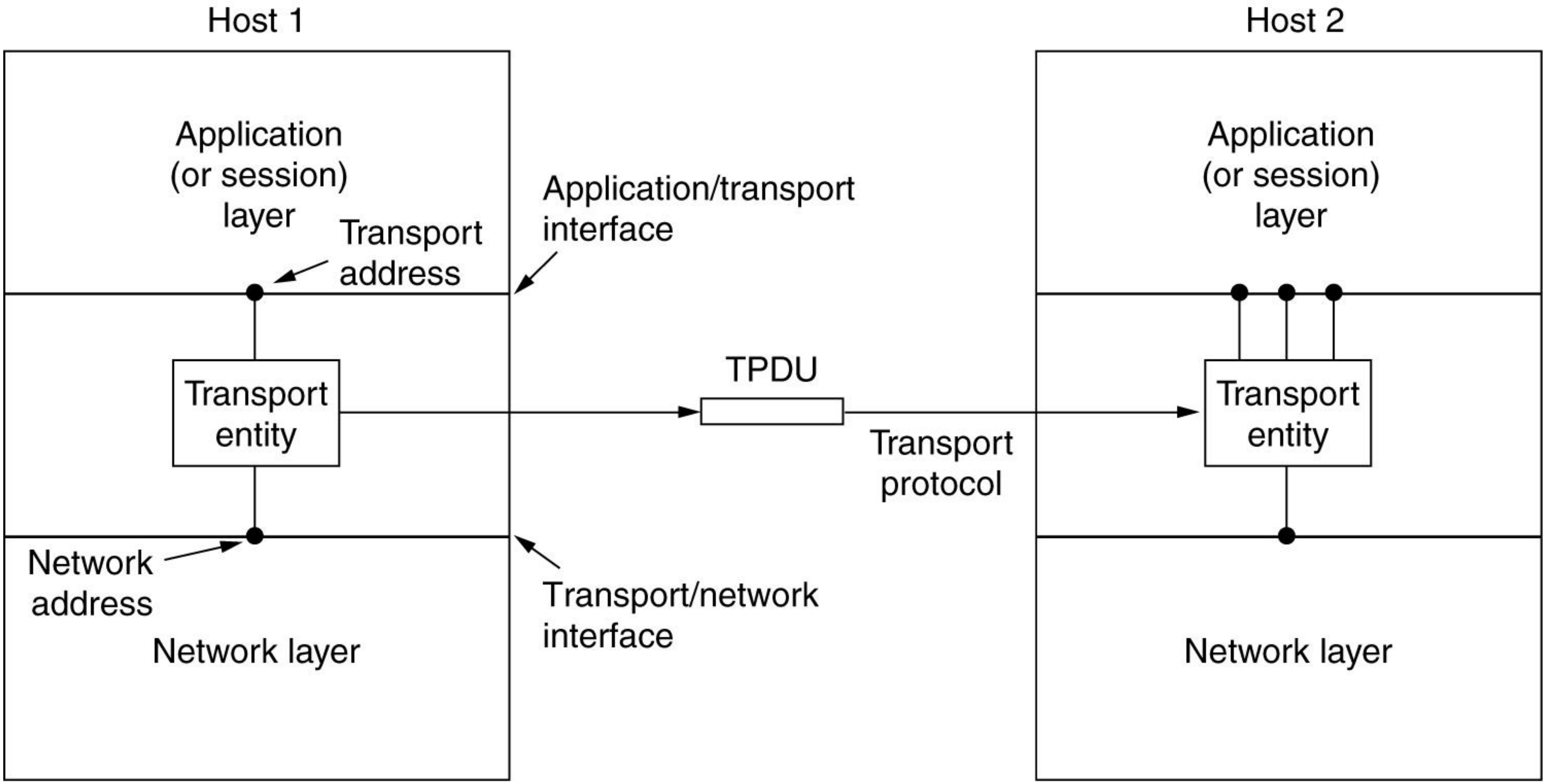
- Services Provided to the Upper Layers
- Transport Service Primitives (简单传输服务原语)
- Berkeley Sockets (套接字)
- An Example of Socket Programming:
  - An Internet File Server

# The Transport Service: Services

- Transport layer services (传输服务) :
  - To provide efficient, reliable, and cost-effective service to its users, normally processes in the application layer.
  - To make use of the services provided by the network layer.
- The transport entity (传输实体): the hardware and/or software within the transport layer that does the work. Its positions:
  - in the OS kernel, in a separate user process, in a library package bound to network applications, or
  - on the network interface card (NIC).

# The Transport Service: Services

## The network, transport and application layers



# The Transport Service: Services

- There are two types of transport services
  - Connection-oriented transport service
  - Connectionless transport service
- The similarities between transport services and network services
  - The connection-oriented service is similar to the connection-oriented network service in many ways:
    - Establishment, →data transfer, →release;
    - Addressing;
    - Flow control
  - The connectionless transport service is also similar to the connectionless network service.

# The Transport Service: Services

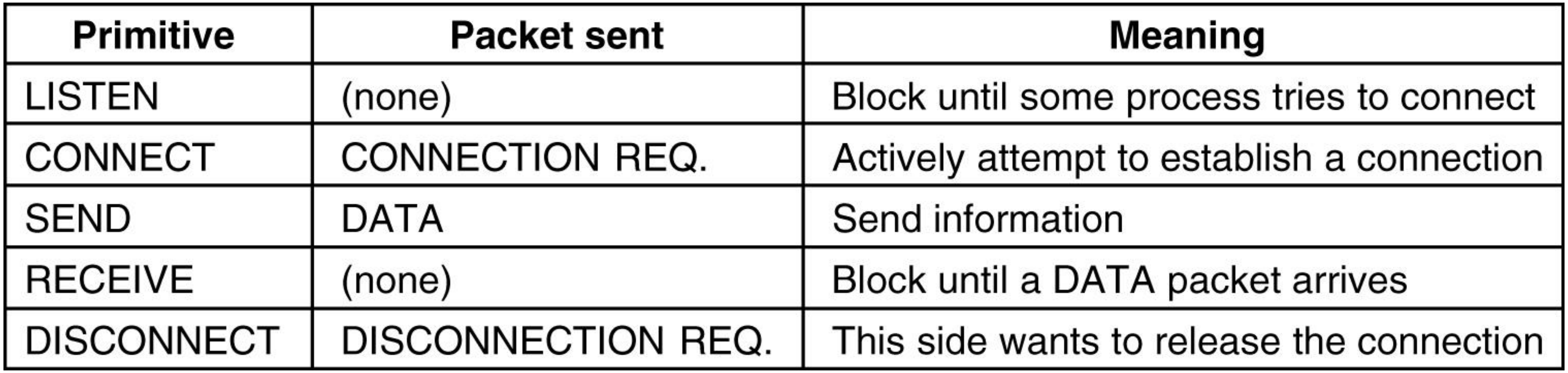
- The differences between transport services and network services. Why are there two distinct layers?
  - The transport code runs entirely on the user’s machines, but the network layer mostly runs on the routers which are usually operated by the carrier.
  - Network layer has problems (losing packets, router crashing, …)
  - The transport layer improves the QoS of the network layer.
  - → The transport service is more reliable than the network service.
  - →Application programmers can write code according to a standard set of transport service primitives and have these programs work on a wide variety of networks.

# The Transport Service: Hypothetical Primitives

- The transport layer provides some operations to application programs, i.e., a transport service interface.
- Each transport service has its own interface.
- The transport service is similar to the network service, but there are also some important differences:
  - Network service models real network → usually unreliable. Transport services improves real network → more reliable.
  - Network services are for network developers.

    Transport services are for application developers.

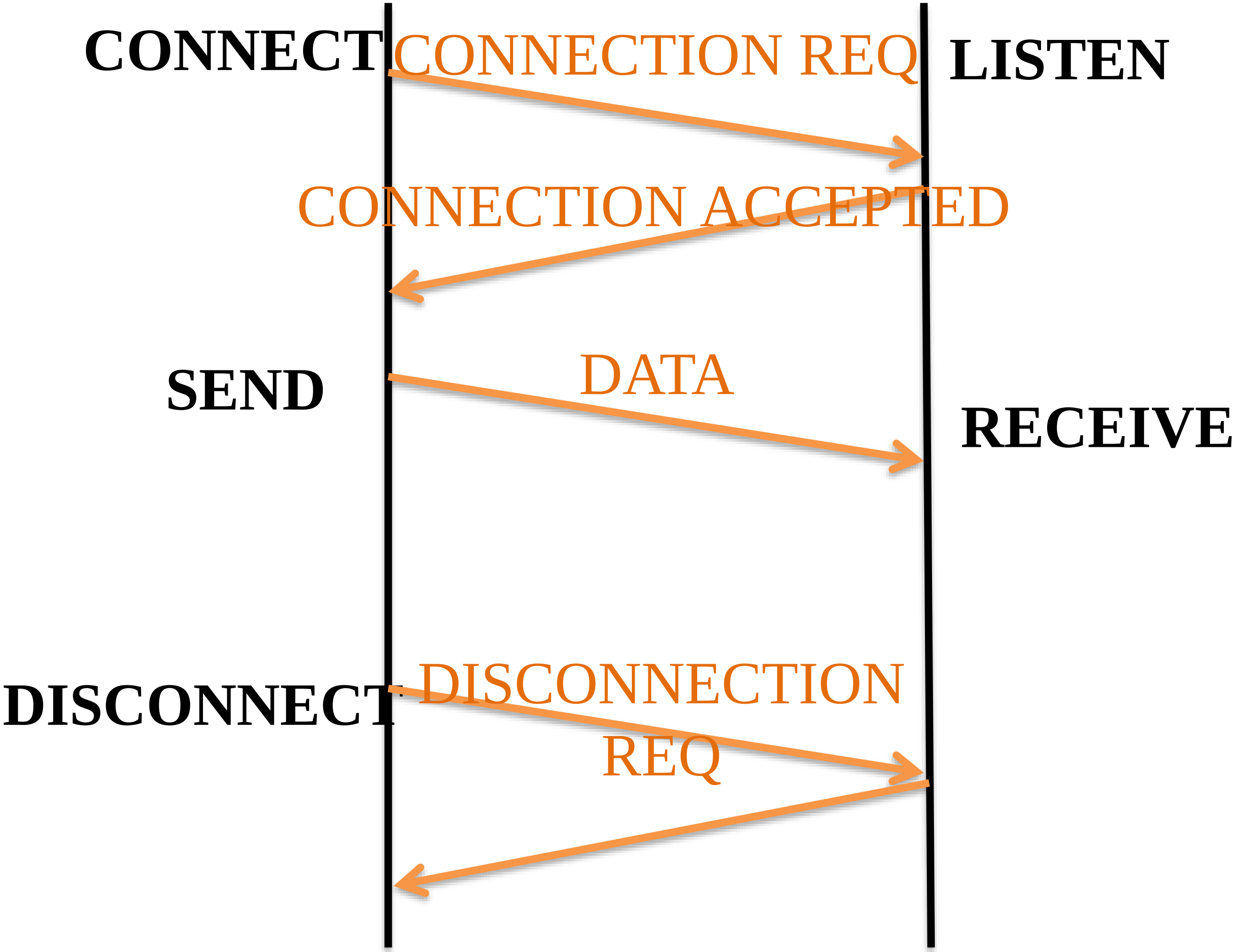
# The Transport Service: **Hypothetical** Primitives



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

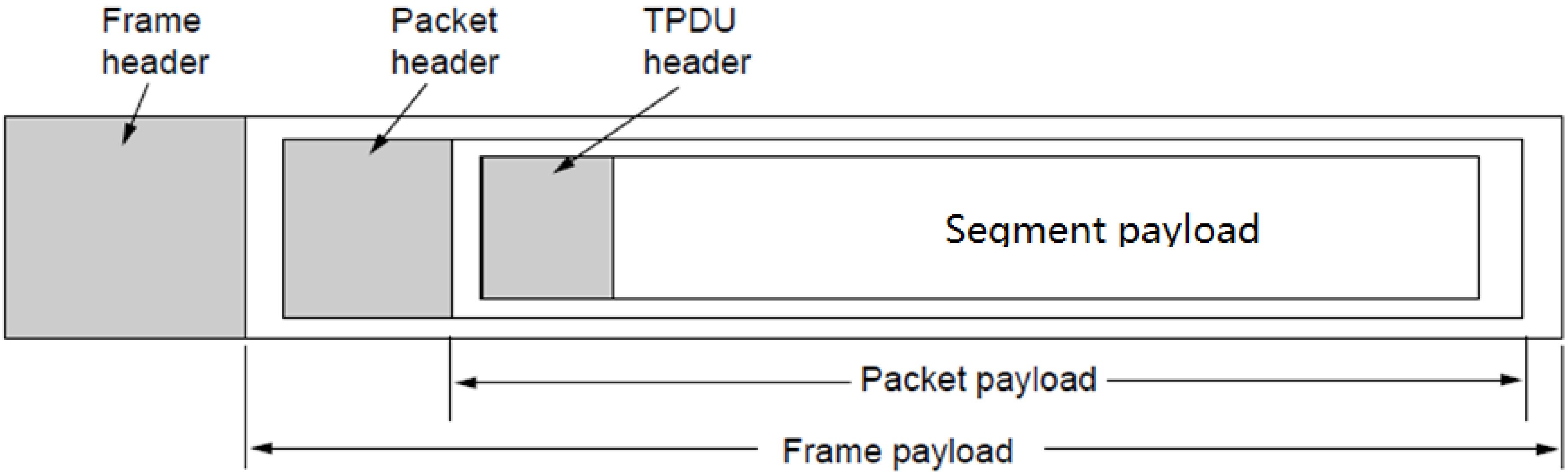
# The Transport Service: Hypothetical Primitives

How to use these **primitives** for an application?



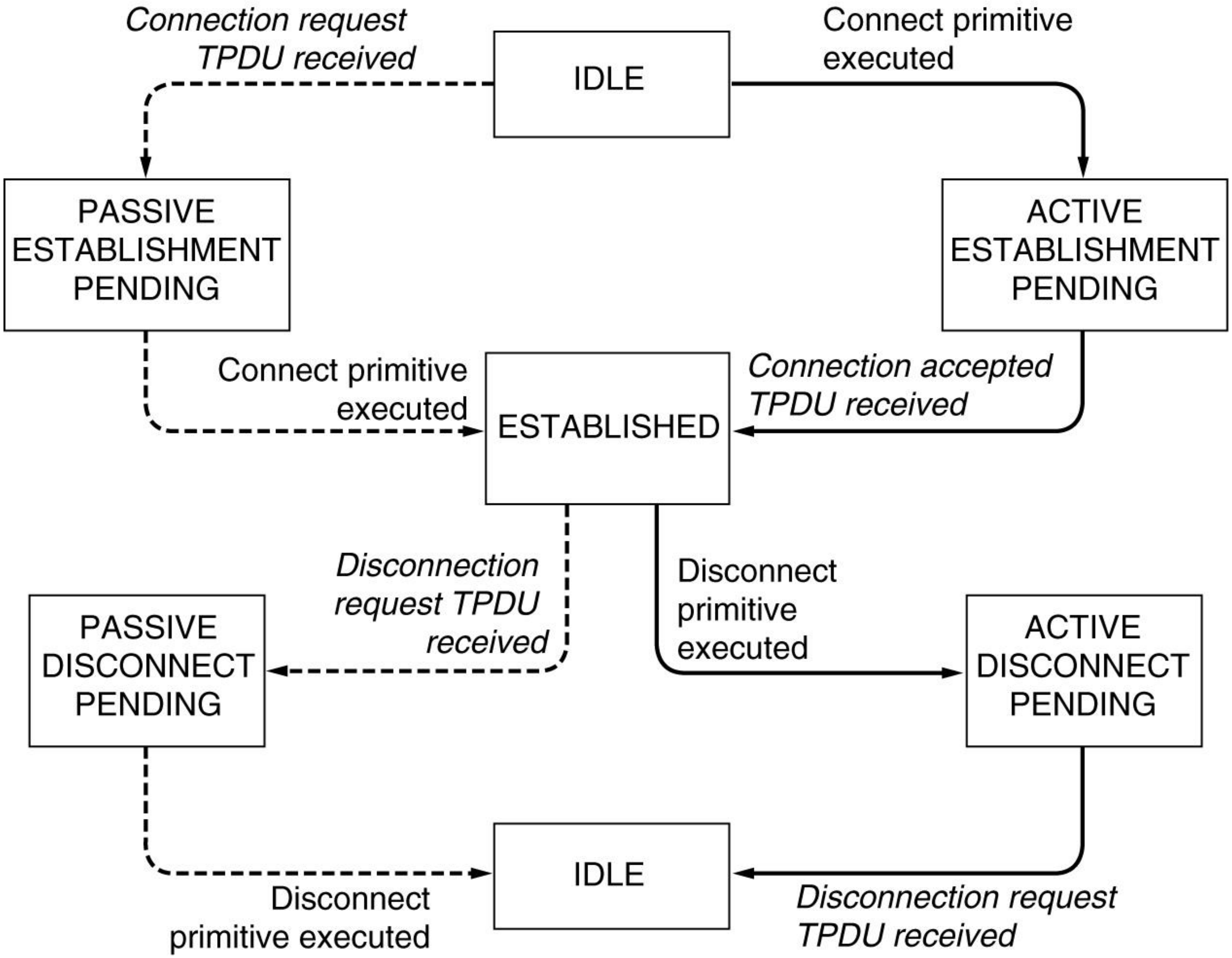
# The Transport Service: Hypothetical Primitives

Nesting of **Segments(=TPDU)**, **packets**, and **frames**



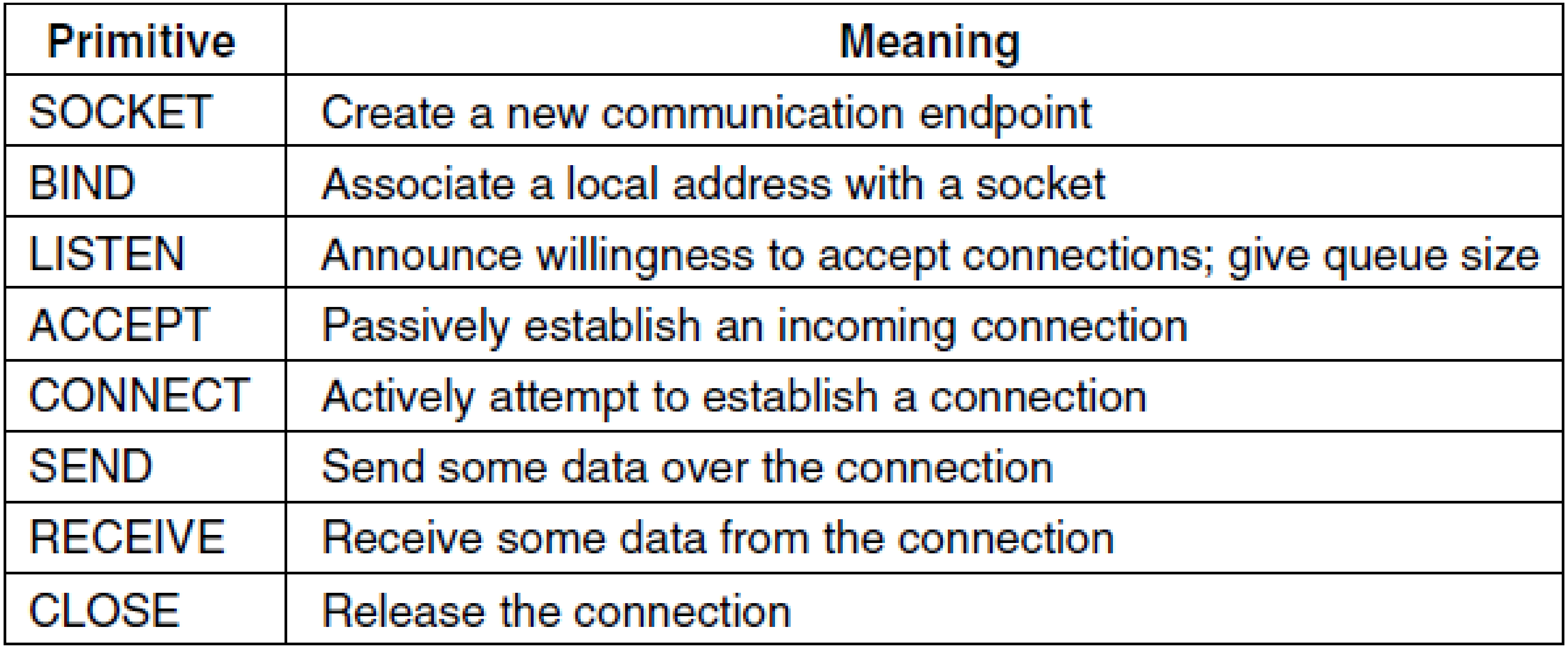
# The Transport Service: Hypothetical Primitives

## Connection establishment and connection release



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

# The Transport Service: **Berkeley** Sockets



| Primitive | Meaning |
|---|---|
| SOCKET | Create a new communication endpoint |
| BIND | Associate a local address with a socket |
| LISTEN | Announce willingness to accept connections; give queue size |
| ACCEPT | Passively establish an incoming connection |
| CONNECT | Actively attempt to establish a connection |
| SEND | Send some data over the connection |
| RECEIVE | Receive some data from the connection |
| CLOSE | Release the connection |

# The Transport Service: Berkeley Sockets

- The C/S Application
  - Server: **SOCKET/BIND/LISTEN/ACCEPT**
  - Client: **SOCKET/CONNECT**
  - C/S: **SEND/RECEIVE**
  - C/S: **CLOSE** (symmetric)

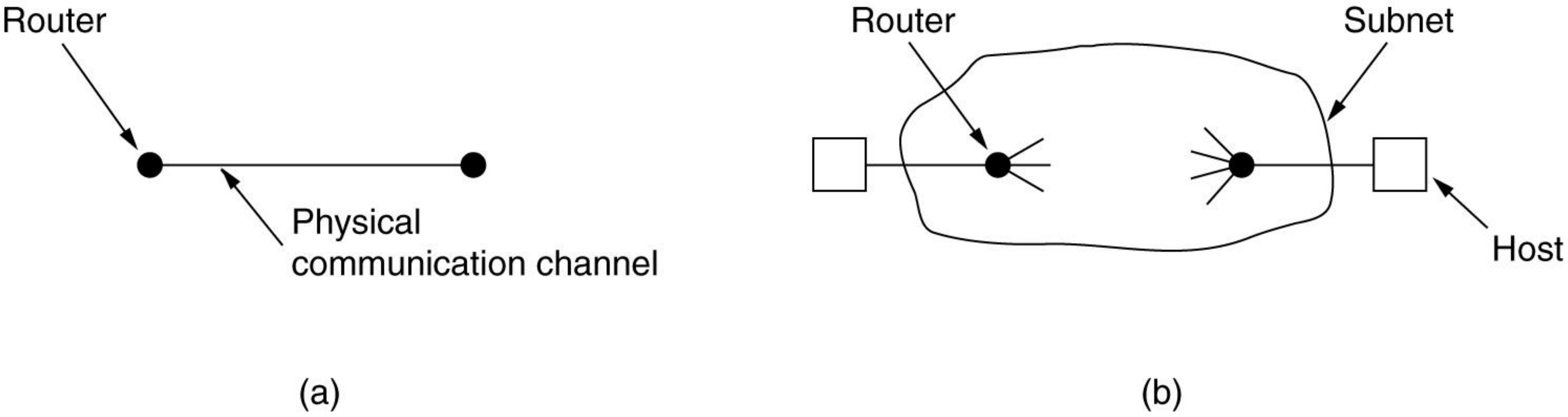
# ELEMENTS OF TRANSPORT PROTOCOLS

- Addressing
- Connection Establishment
- Connection Release
- Error Control and Flow Control
- Multiplexing
- Crash Recovery

# Elements of Transport Protocols

Transport protocol  and  data link protocol

- **Similarities**: Error control, sequencing, flow control, …
- **The key difference**: environments the two protocols operate
  - Link layer: two routers communicate directly via a physical channel
  - Transport layer: physical channel -> entire network



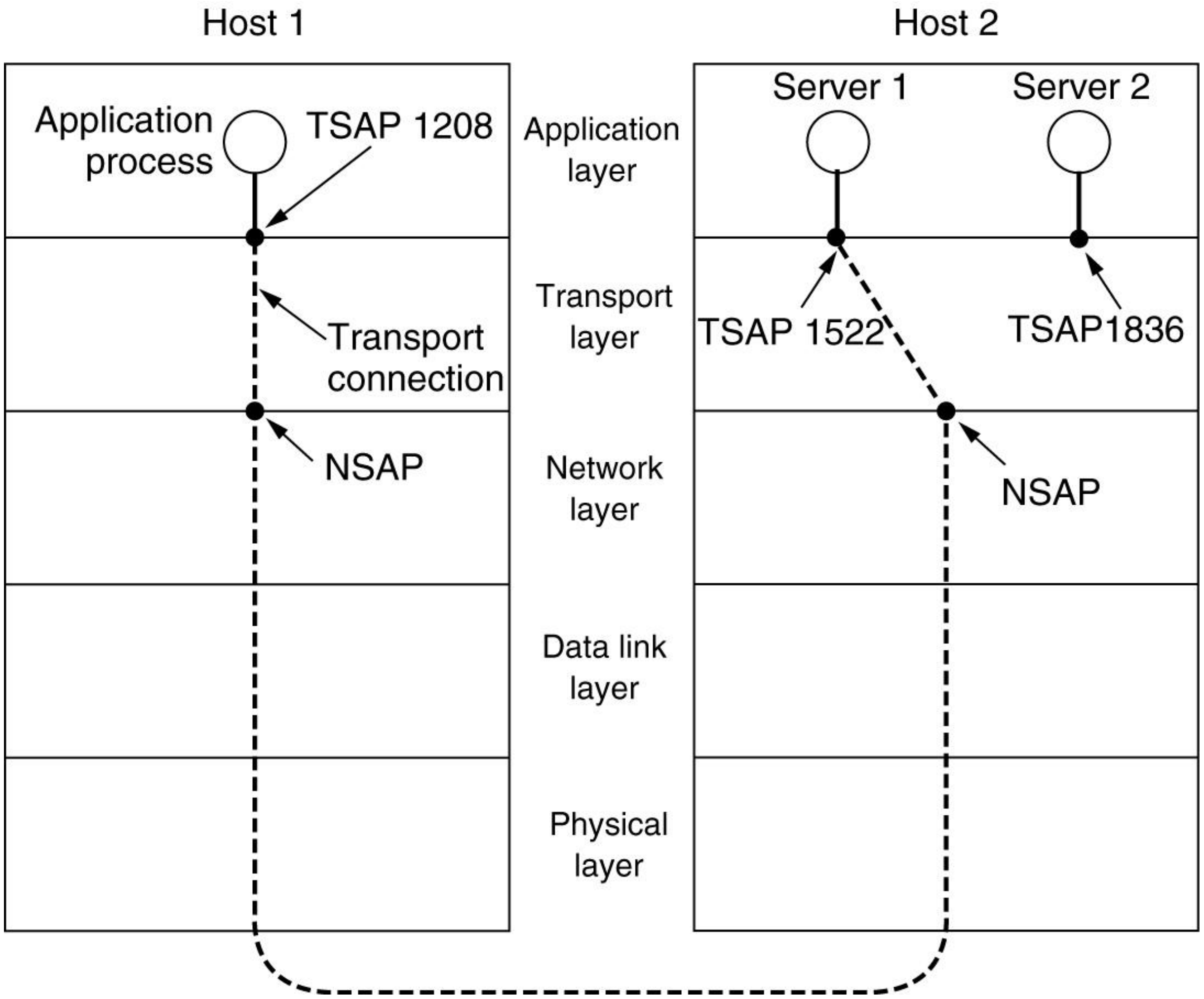
# Elements of Transport Protocols

## Other differences

- Addressing:
  - Link: **implicit**
  - Transport: **explicit**
- Connection establishment
  - Link: **simple**
  - Transport: **complicated**
- Storage capacity:
  - Link: **no**
  - Transport: **yes** => *a network can delay a packet*
- Buffering
  - Link: allocate a **fixed** number of buffers to each line
  - Transport: a large number of connections and variations in the bandwidth may require a **dynamic and complicated** approach.

# Elements of Transport Protocols: Addressing

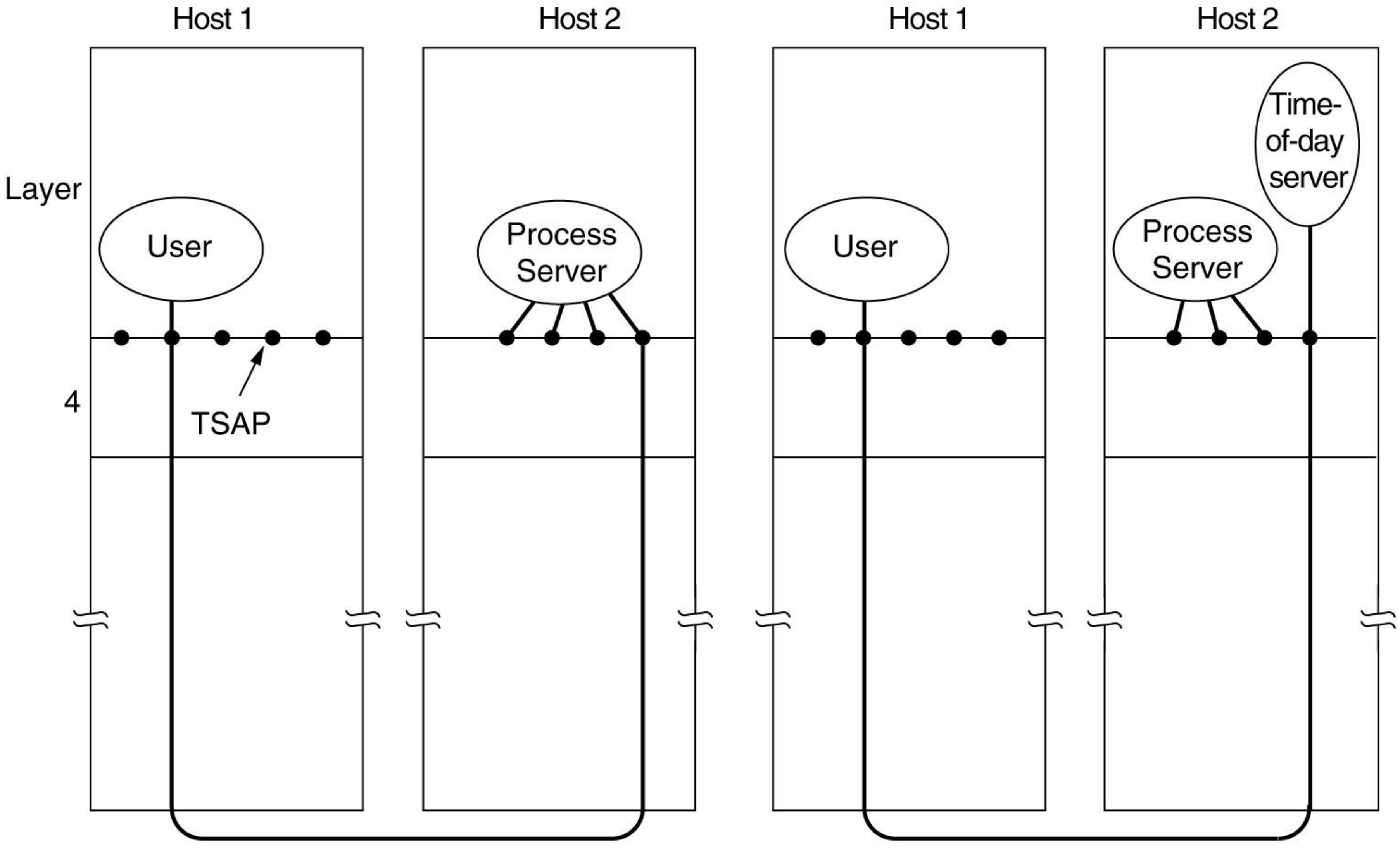
How to address? TSAP=Transport Service Access Point -> port; NSAP -> IP



# Elements of Transport Protocols: Addressing

- NSAP: Network service access point
  - IP (32 bits)
- TSAP: Transport service access point
  - Port (16 bits)
- How a host knows a TSAP address at a server?
  - Some TSAP addresses are so *famous* that they are fixed, e.g. HTTP=80 and others listed in /etc/services
  - A special process **portmapper**: service name -> TSAP address
- Many server processes will be used rarely and it is wasteful to have them active
  - **process server**: e.g. inetd on UNIX

# Elements of Transport Protocols: Addressing



(a)

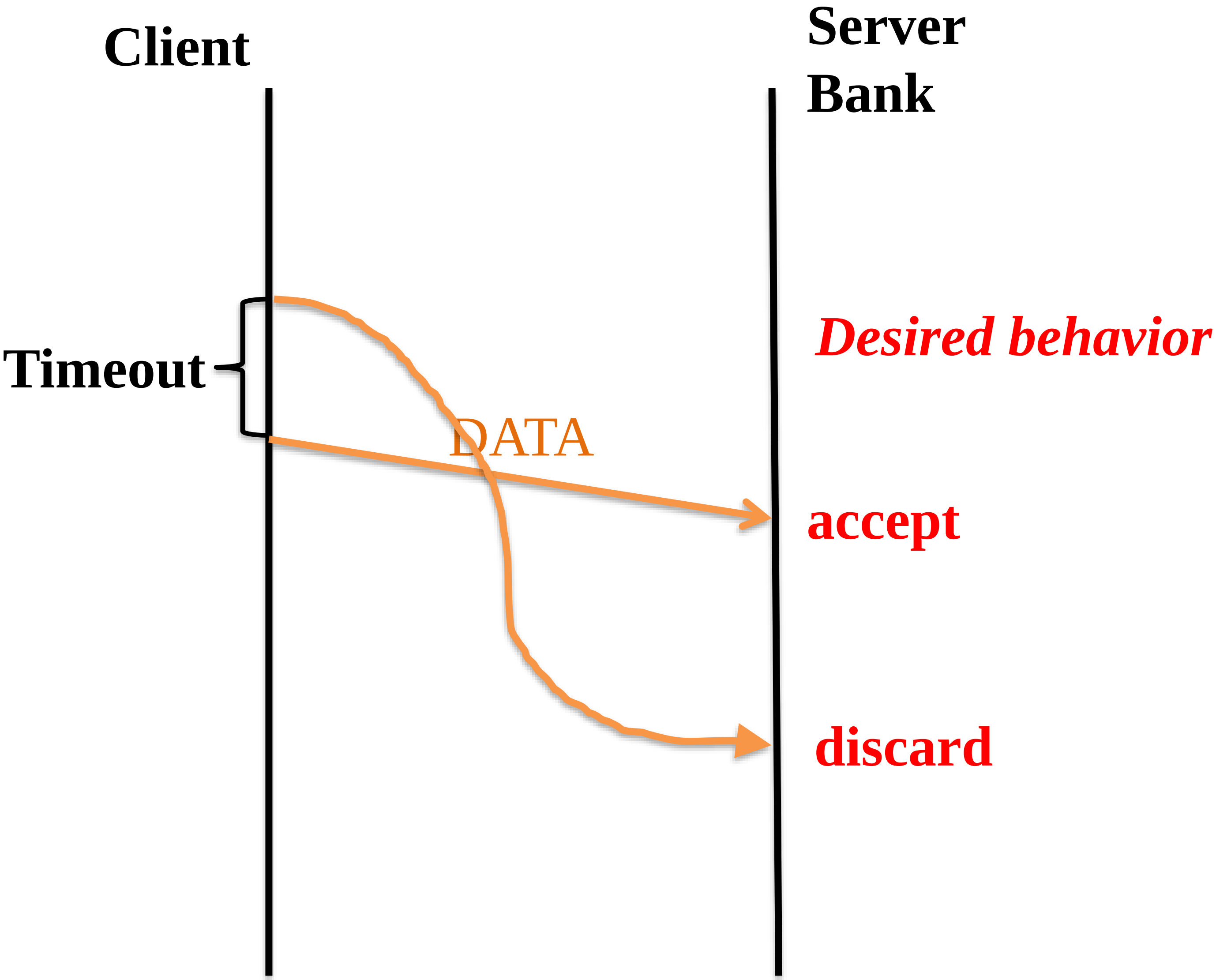
(b)

# Elements of Transport Protocols: Connection Establishment

Connection establishment sounds easy, but …

- Naïve approach:
  - One sends a **CONNECTION REQ** segment to the other and wait for a **CONNECTION ACCEPTED** reply.
  - If ok, done; otherwise, retry.
  - Possible nightmare:
    - A user establishes a connection with a bank, sends messages telling the bank to transfer a large amount of money to the account of a not-entirely-trustworthy person, and then releases the connection.
    - Moreover, assume each packet in scenario is ***duplicated*** and stored in the subnet.
  - **→ delayed duplicates problem**

# Delayed Duplicates Problem



Client

Server
Bank

Timeout

DATA

*Desired behavior*

**accept**

**discard**

# Solution 1

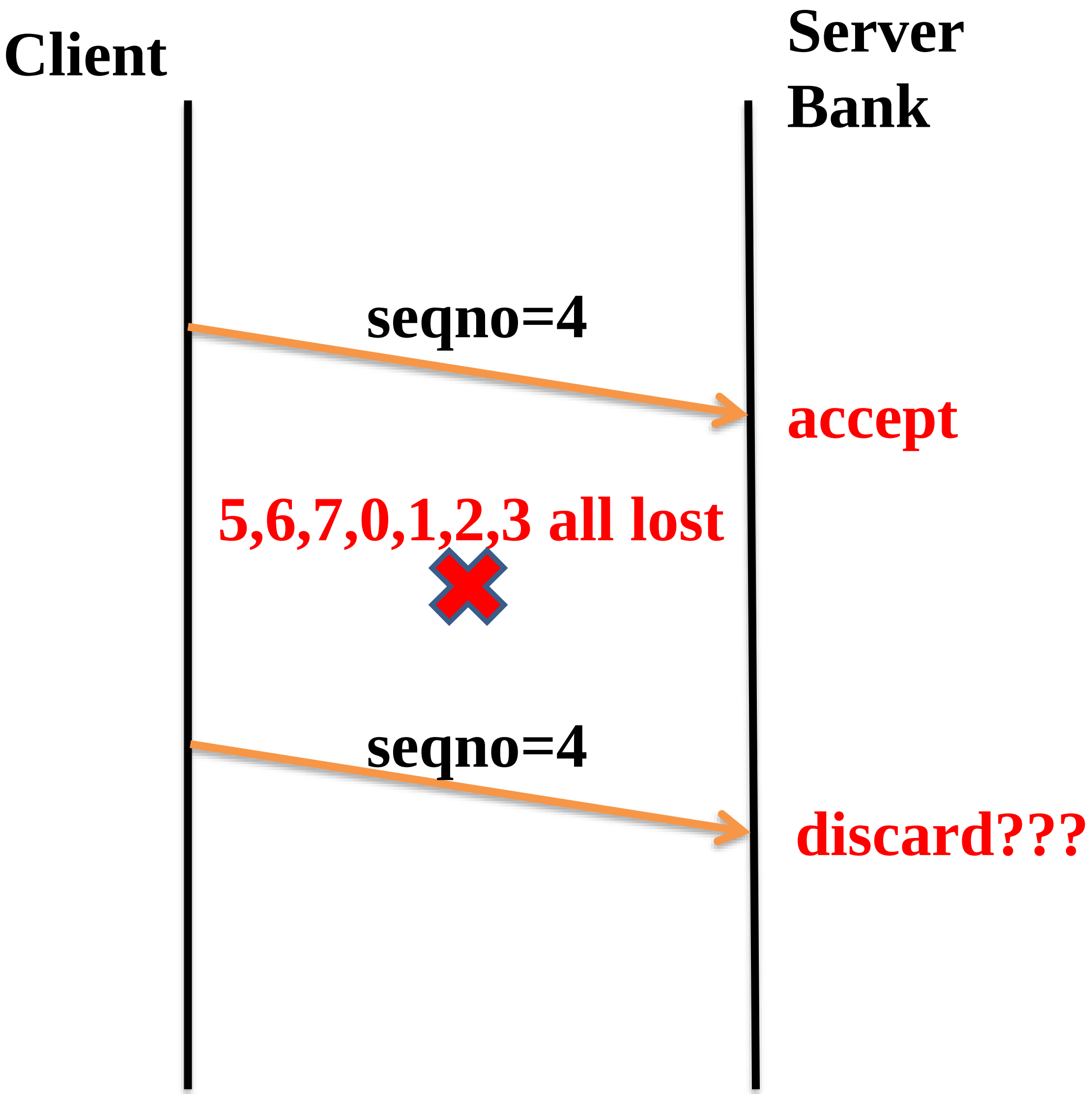
- It is easy to detect duplicates if there are sequence numbers
- Code at server:

```
packet arrival:
    if (packet.seq see before)
        discard packet
```

# Problems

- There are two problems which complicates the scenario:
  - Possible wrap around of seqno
  - Client/Host or Server may crash
- **Assume** seqno = 0,1,…,7 (i.e. 3 bits)

# Possible wrap around of seqno



Client

Server
Bank

seqno=4

accept

5,6,7,0,1,2,3 all lost

seqno=4

discard???

# Problem & Assumptions

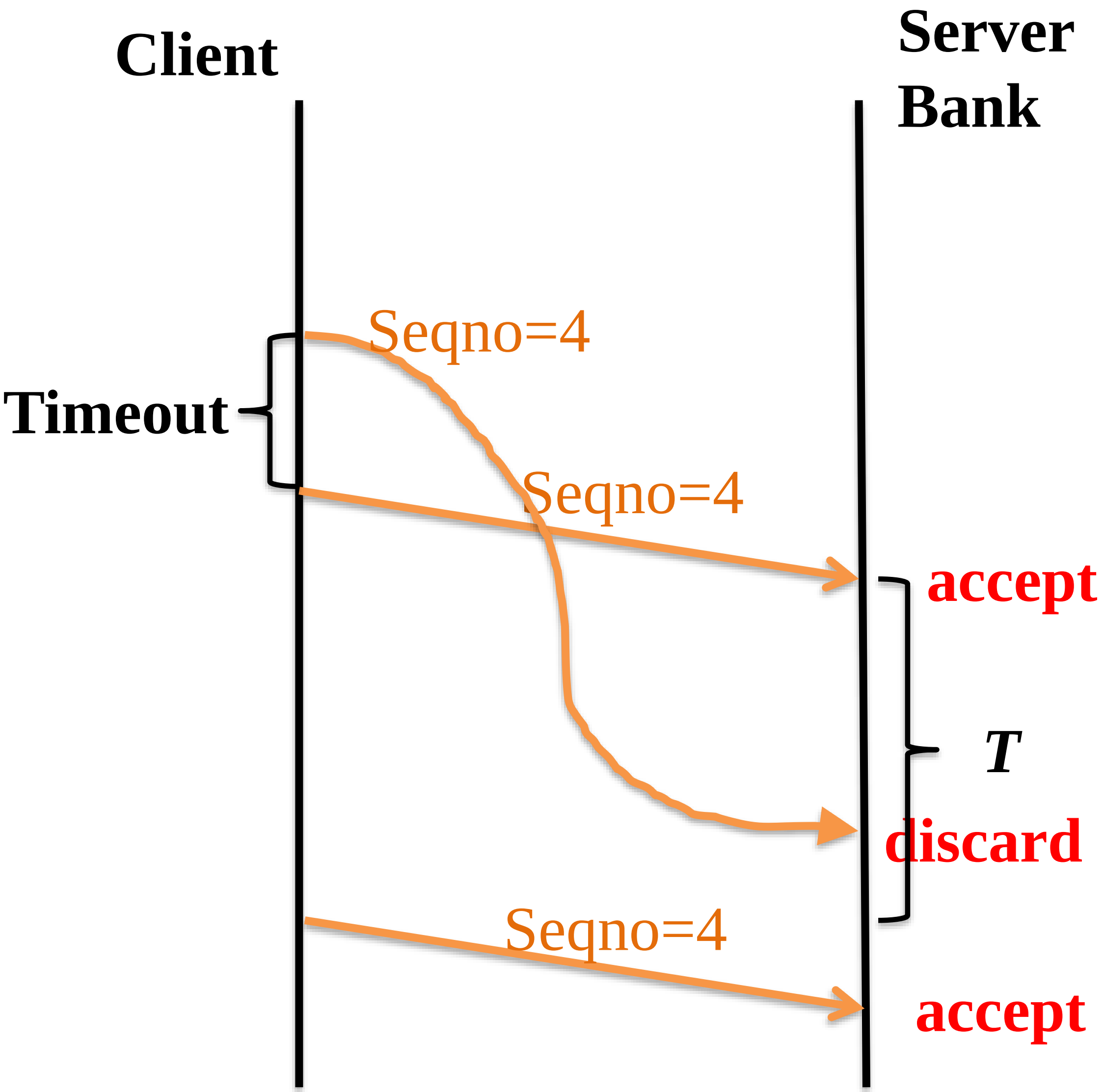
- Problem
  - How to **differentiate** ***delay duplicate*** and ***new pkt with wrapped around seqno***?
- Idea: use time
- ***Two assumptions*** that simplifies the problem
  - **Assumption 1.** Time for seqno wrap around (**T1**) is typically large if, e.g., seqno is 32 bits long.
  - **Assumption 2.** If pkt delays a relatively short time (**T2<T1**), we can use time to differentiate the two.

# Solution 2

- Revised code at server:

```
packet arrival:
    if (packet.seq see before and time is short ($<T$))
        discard packet
    if (packet.seq see before and time is large ($>T$))
        accept packet
```

# Client behavior of solution 2



# How to realize the assumptions?

- ***1. Restrict packet lifetime***
- Packet lifetime can be restricted to a known maximum using one of the following techniques
  - Restricted subnet design.
  - Putting a hop counter in each packet.
  - Timestamping each packet (router synchronization required)
- ***T*** = *n* * (pkt lifetime)
  - It is impossible to receive a delay duplicate after ***T***

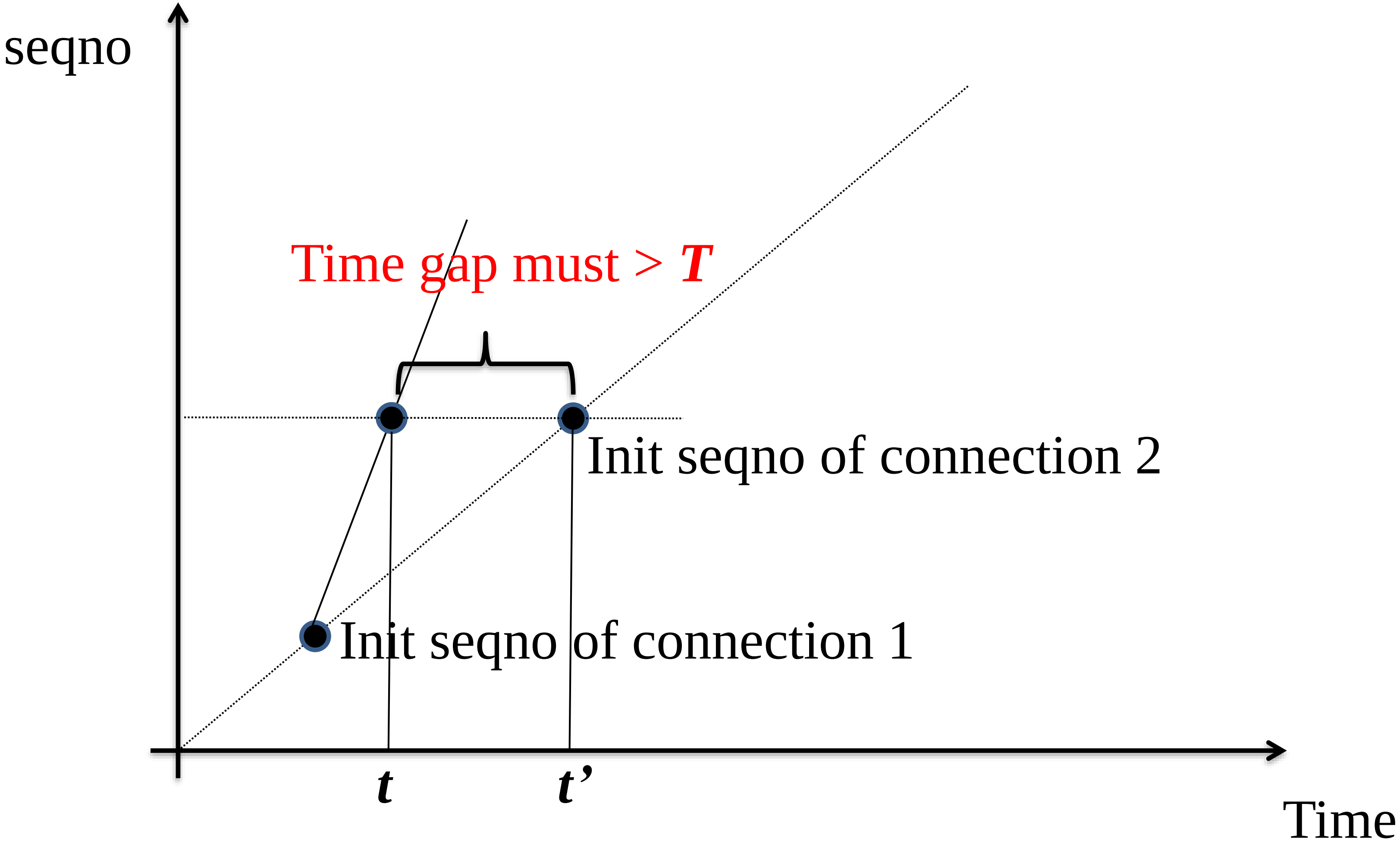
# How to realize the assumptions?

- ***2. use time-of-day clock to limit sending rate***
- Time-of-day clock at hosts
  1. Each clock is assumed to take the form of a binary counter that increments itself at uniform intervals.
  2. The number of bits in the counter must equal or exceed the number of bits in the sequence numbers.
  3. The clock is assumed to continue running even if the host goes down.
  4. The clocks at different hosts need not be synchronized.
- initial seqno of a connection = low ***k*** bits of time-of-day clock

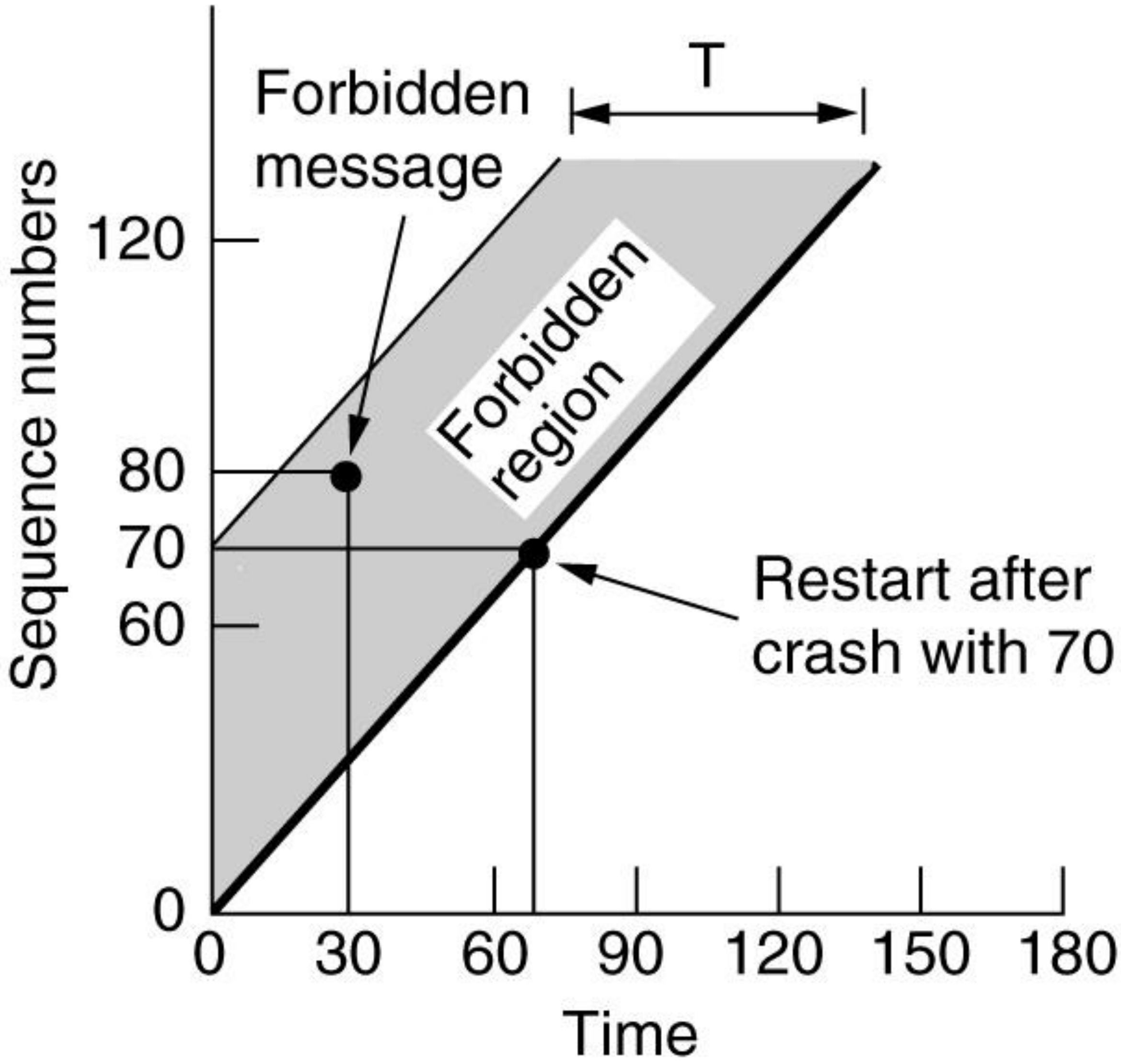
# How to realize the assumptions?

- **Assume** the same bits of seqno and time-of-day clock
- Two time instants, ***t*** and ***t’***, ***t<t’***
- If we allow legitimate reuse of the same seqno at ***t*** and ***t’***, we should ensure that ***t’-t>T***

# Forbidden region of seqno



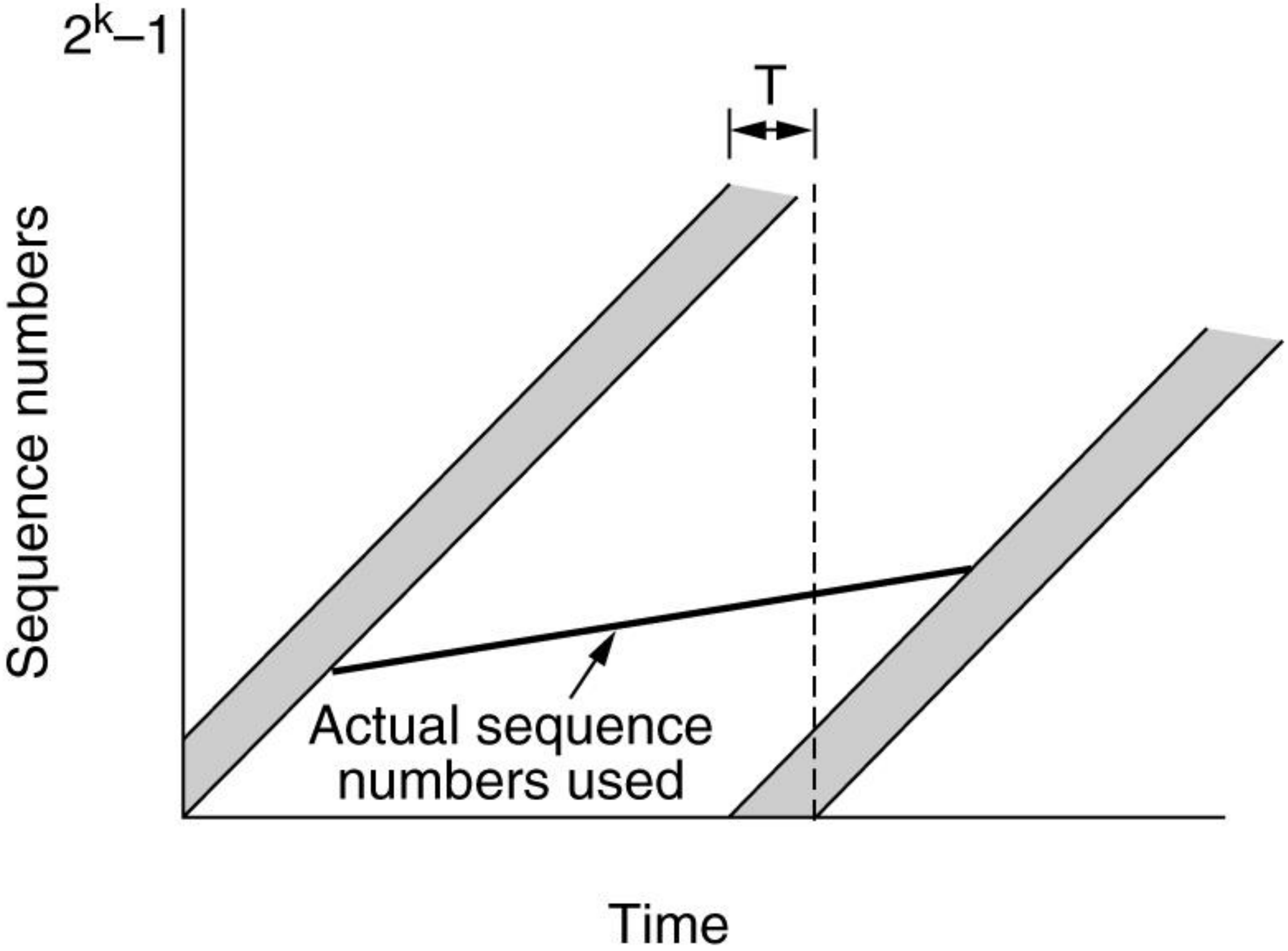
# Forbidden region



# Consequence of entering the forbidden region

- T = 60s; Clock ticks once per second
- The initial sequence number for a connection opened at time *x* will be *x*.
- *t*=30s, segment **X** being sent on (a previously opened) connection 5 is given sequence number 80. (**forbidden region!!!**)
- After sending **X**, the host crashes and then quickly restarts.
- *t* = 60s, it begins reopening connections 0 through 4.
- *t* = 70s, it reopens connection using initial sequence number 70 as required.
- During the next 15s it sends data segments 70 through 80. Thus at *t*=85s, a new segment **Y** with sequence number 80 and connection 5 has been injected into the subnet.
- Now time gap between **X** and **Y** is 85-30=55s < 60s and the protocol will mistakenly regard **Y** as a duplicate of **X** (**not desired!!!**)

# Elements of Transport Protocols: Connection Establishment



# Elements of Transport Protocols: The forbidden region: **summary**

- The increments of seqno cannot be too fast or too slow

- **Solution for the delayed duplicates**
  - **1. *the maximum data rate on any connection is one segment per clock tick*.** (i.e., cannot be too fast)
  - **2. *limits how slowly sequence numbers can advance on a connection (or how long the connections may last)*.** (i.e., cannot be too slow)

# Question

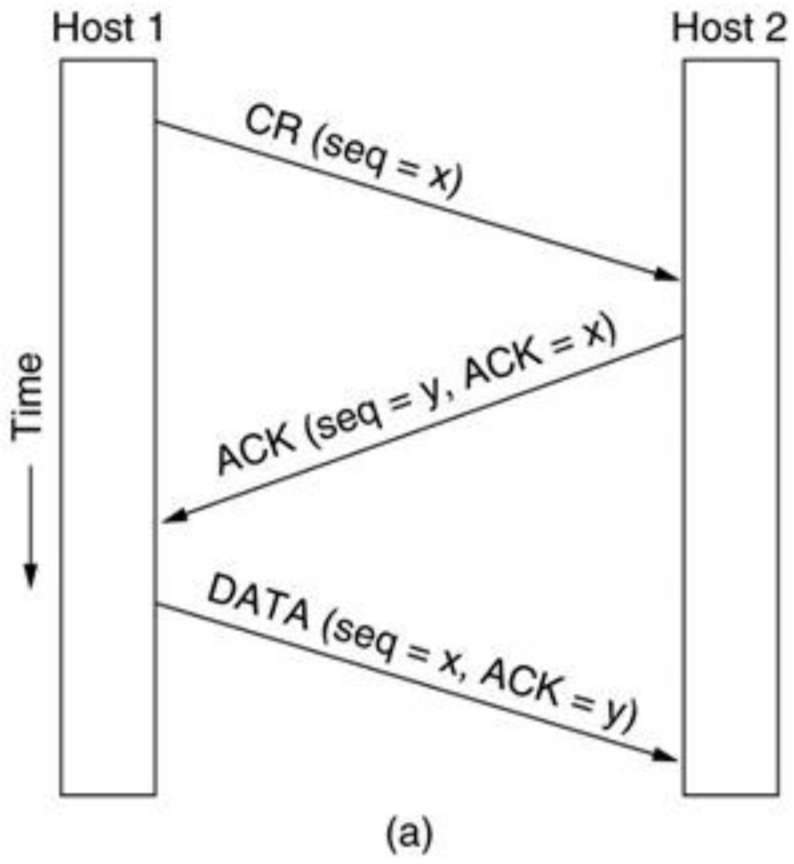
- In a network whose max segment is 128 bytes, max segment lifetime is 30 sec, and has 8-bit sequence numbers, what is the maximum data rate per connection?

# Connection establishment

- The above introduced method **solves** the problem of not being able to distinguish delayed duplicate segments from new segments.
- However, there is **a practical issue** for using it for establishing connections, i.e., we do not normally remember seqno across connections at the destination, we still have no way of knowing if a CONNECTION REQ segment containing an initial seqno is a duplicate of a recent connection.
- This issue does not exist during a connection because the sliding window protocol does remember the current sequence number.

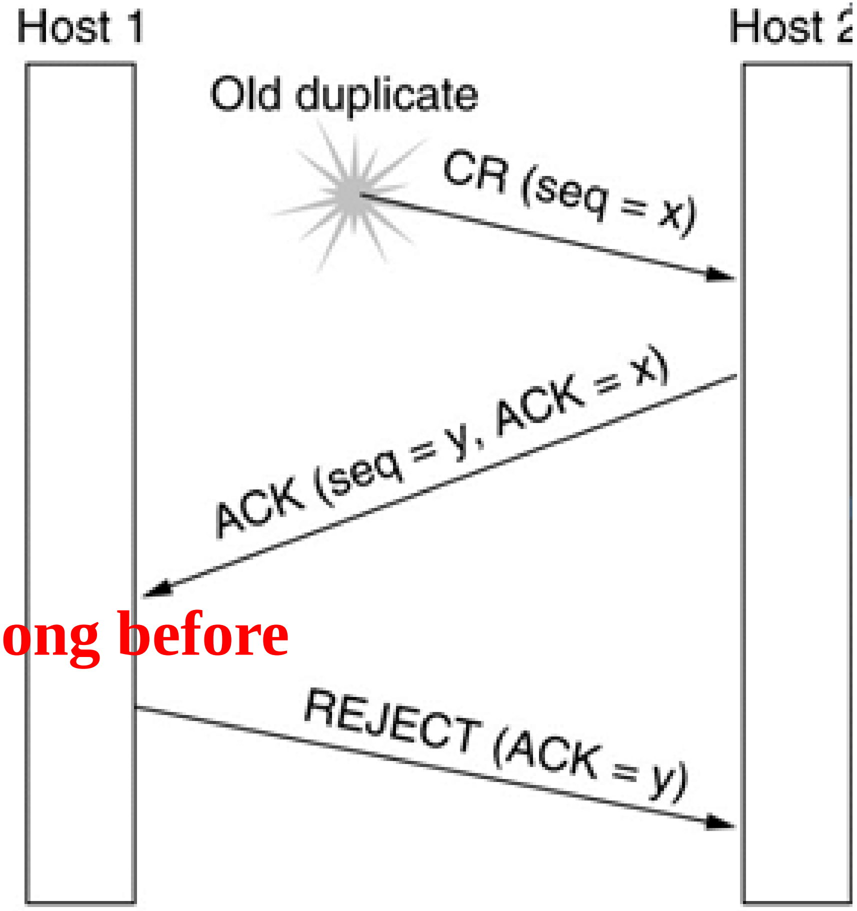
# Elements of Transport Protocols: Connection Establishment(1)

- **Three-way handshake** for connection establishment



**Normal operation**

# Elements of Transport Protocols: Connection Establishment(2)

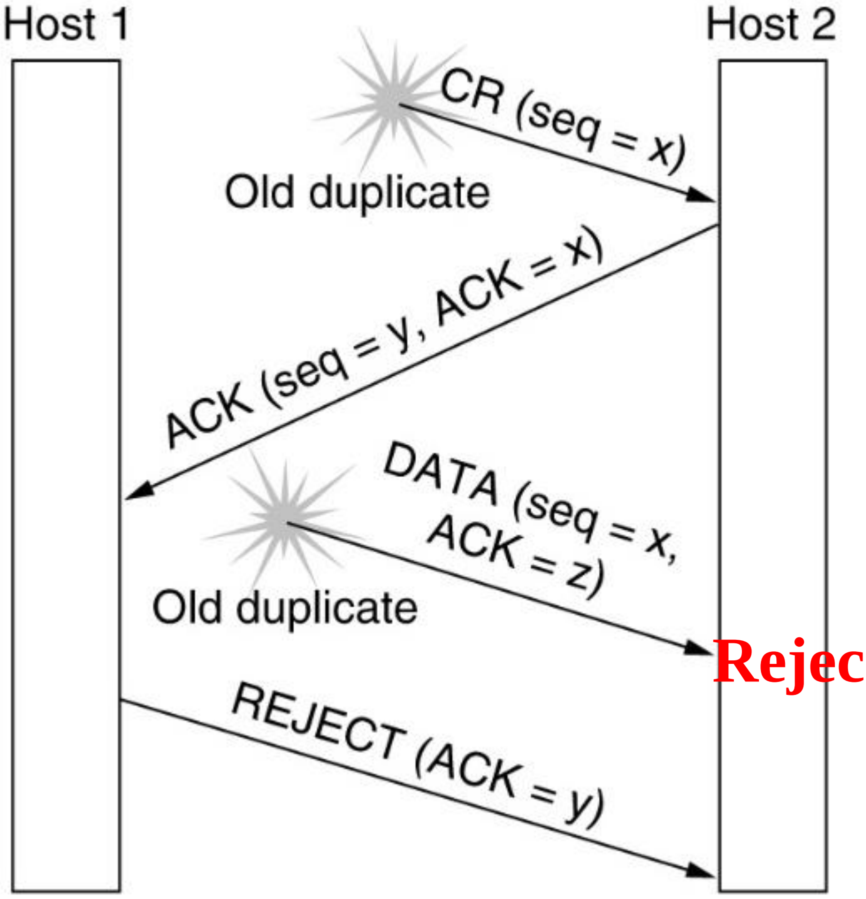


**Reject since x is long before**

(b)

**Old duplicate CONNECTION REQUEST appearing out of nowhere.**

# Elements of Transport Protocols: Connection Establishment(3)



Reject since y is not acked

(c)

**Duplicate CONNECTION REQUEST and duplicate ACK.**

# Elements of Transport Protocols: Connection Establishment

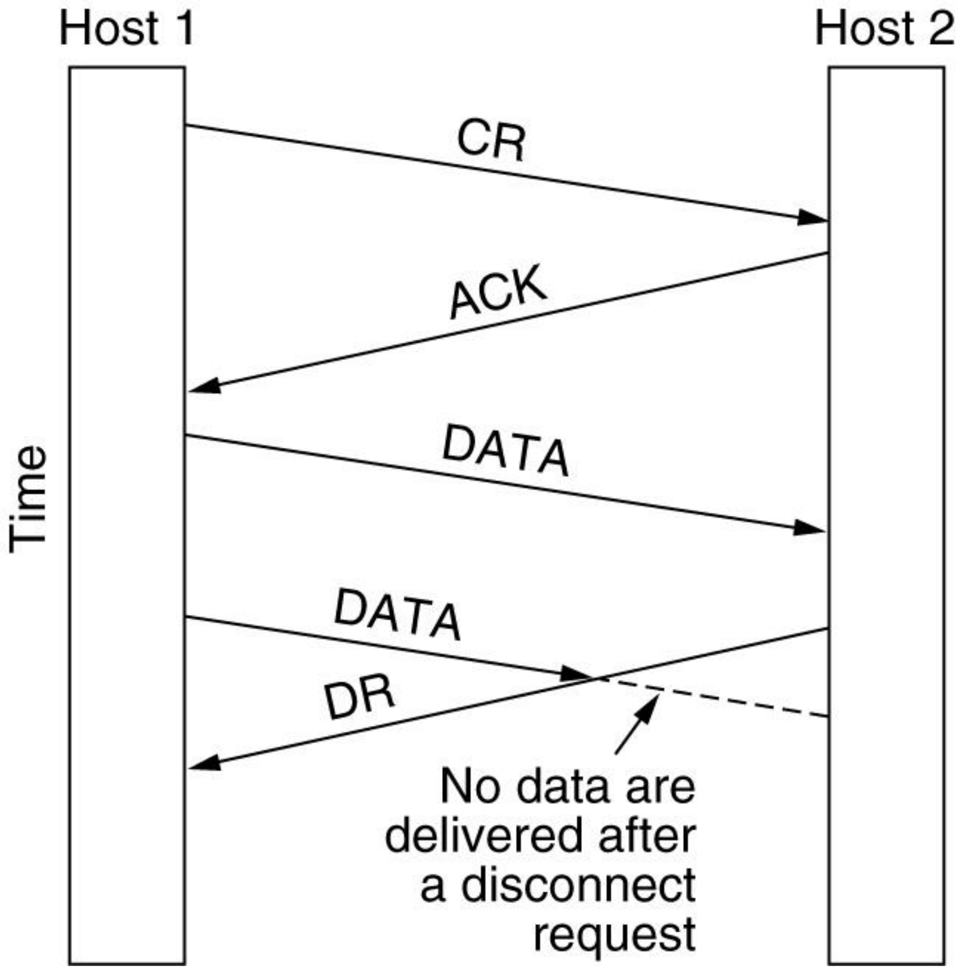
- **Conclusion**: there is no combination of old CONNECTION REQ, CONNECTION ACCEPTED, or other data segments that can cause the protocol to fail and have a connection setup by accident when no one wants it.

# Elements of Transport Protocols: Connection Release

- **Asymmetric** release and **symmetric** release
  - Asymmetric release: When one part hangs up, the connection is broken.
  - Symmetric release: to treat the connection as two separate unidirectional connections and require each one to be released separately.
- Asymmetric release is abrupt and may result in data loss (See the next slide)
- One way to avoid data loss is to use symmetric release.

# Elements of Transport Protocols: Connection Release

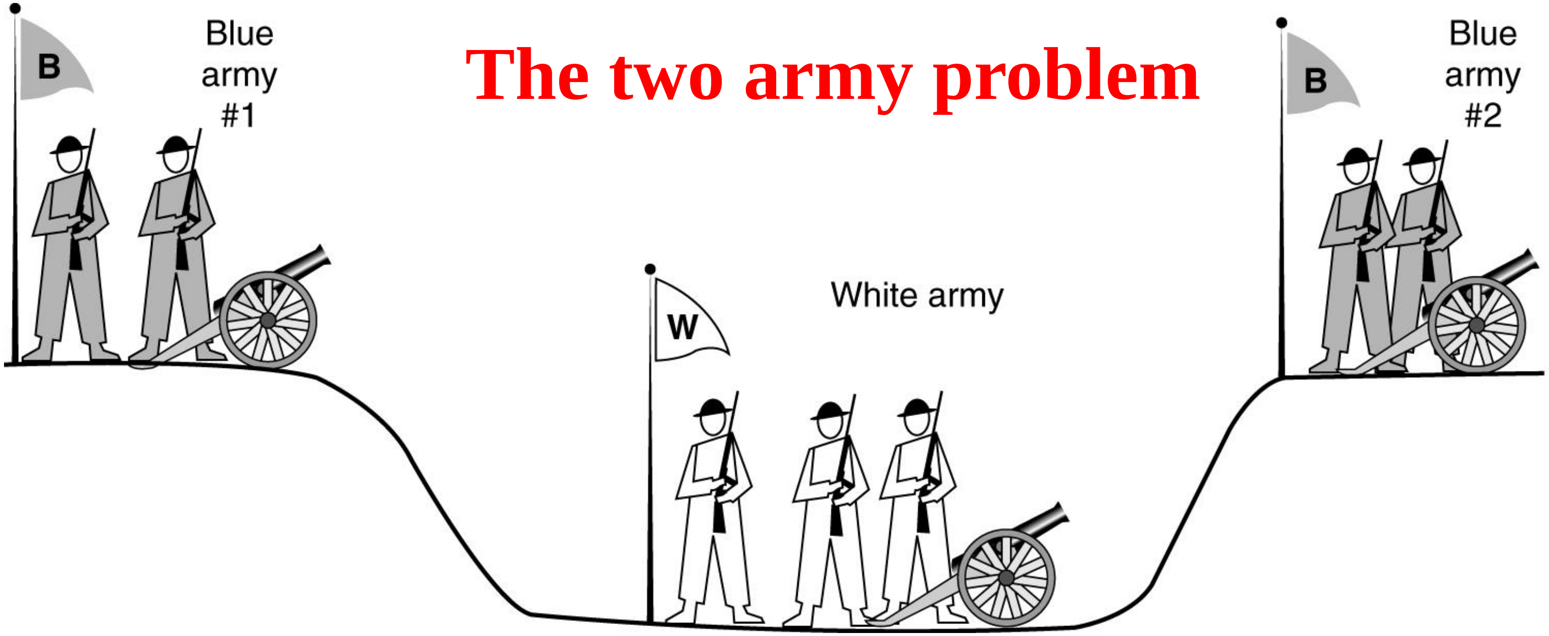
- Abrupt disconnection with loss of data



# Elements of Transport Protocols: Connection Release

- Symmetric release does the job when each process has a fixed amount of data to send and clearly knows when it has sent it.
- Symmetric release has its problems if determining that all the work has been done and the connection should be terminated is not so obvious.

**The two army problem**



# Elements of Transport Protocols: Connection Release

- **Two army-problem**: A white army is encamped in a valley. On both of the surrounding hillsides are blue armies.

  ***one blue army < white army < two blue armies***

- Does a protocol exist that allows the blue armies to win?
  - **Two-way** handshake

    The commander of blue army #1: “I propose we attack at down on March 29. How about it?”

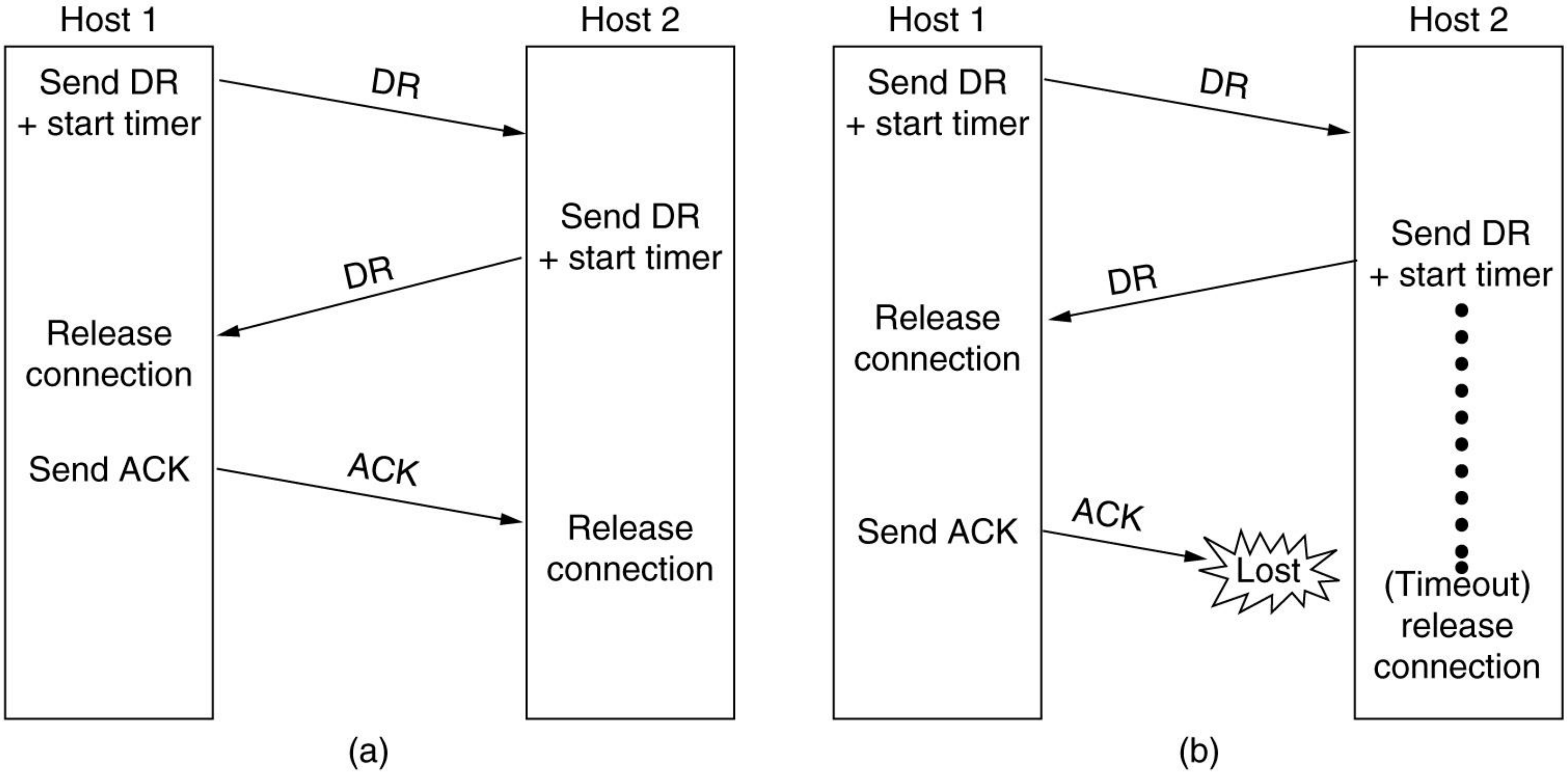
    The commander of blue army #2: “OK.”

    → Will the attack happen?
  - Three-way handshake, …, N-way handshake
  - ➔No protocol exists that works since the sender of the final message can never be sure of its arrival, he will not risk attacking
- Substitute “disconnect” for “attack”. If neither side is prepared to disconnect until it is convinced that the other side is prepared to disconnect too, the disconnection will never happen.

# Elements of Transport Protocols: Connection Release

**three-way handshake** for connection release. While this protocol is not infallible, it is usually adequate.

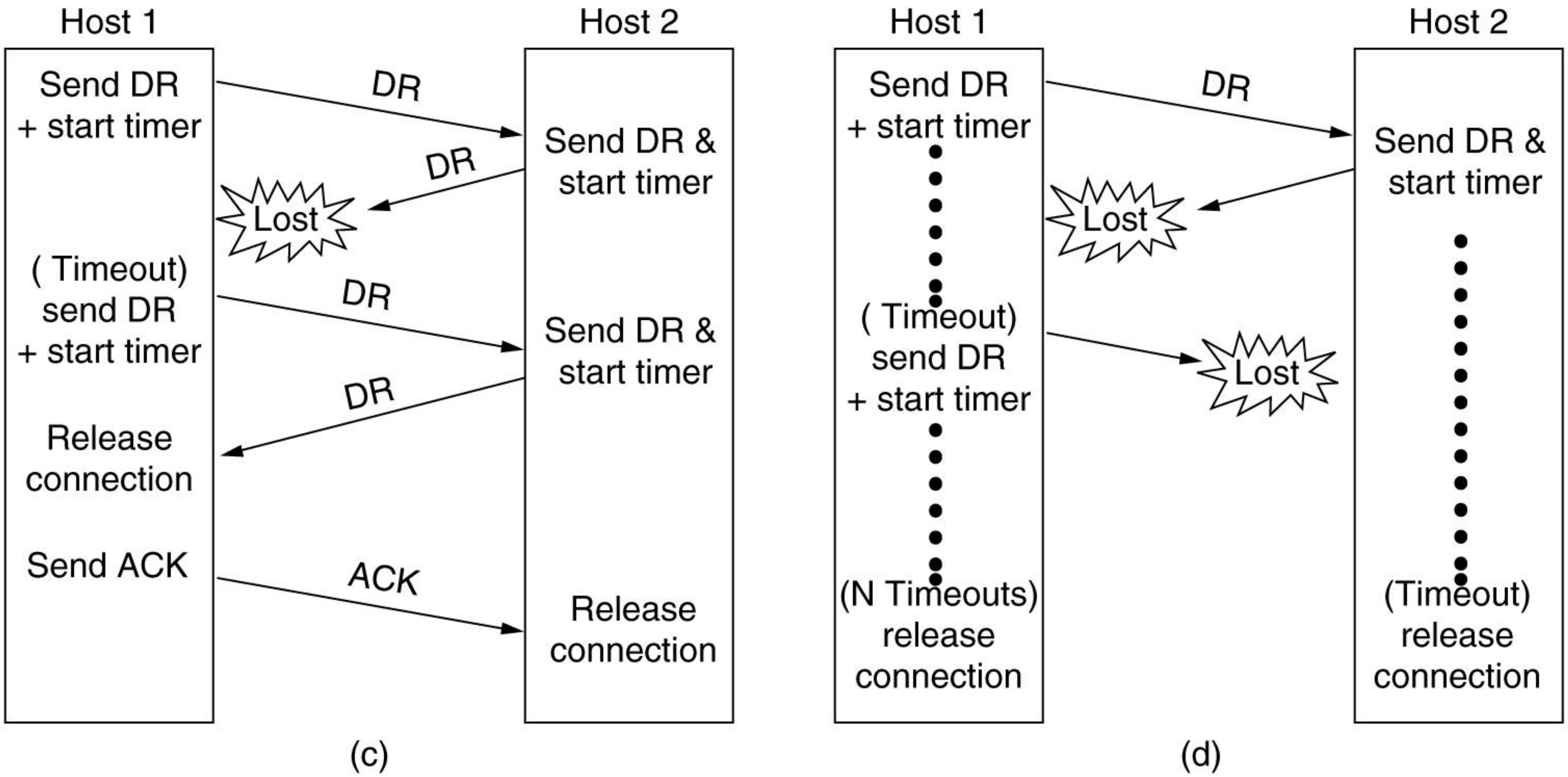
(a) Normal case of three-way handshake (b) Final ACK lost



# Elements of Transport Protocols: Connection Release

(c) Response lost

(d) Response lost and subsequent DRs Lost



# Elements of Transport Protocols: Connection Release

- **Automatic** disconnect rule
  - If no segments have arrived for a certain number of seconds, the connection is then *automatically* disconnect.
  - Thus, if one side ever disconnects, the other side will detect the lack of activity and also disconnect.
- **Conclusion**: releasing a connection without data loss is not nearly as simple as it first appears.

# Elements of Transport Protocols: Error Control and Flow Control

**Error control** is ensuring that the data is delivered with the desired level of reliability, usually that all of the data is delivered without any errors.

**Flow control** is keeping a fast transmitter from overrunning a slow receiver.

Solutions to these issues (same as data link layer):

- A frame carries **error detecting codes**, e.g., CRC
- **ARQ**: Automatic Repeat reQuest
- **Max send window size** *w*: **stop and wait** (*w*=1), large *w* enables **pipelining**
- **Sliding window protocol**: combines the above features and supports bidirectional data transfer.

# Elements of Transport Protocols: Error Control and Flow Control

**Differences**:

- Error detection:
  - The link layer checksum protects a frame while it crosses a single link.
  - The transport layer checksum protects a segment while it crosses an entire network path. It is an end-to-end check, which is not the same as having a check on every link.
  - **End-to-end argument:** transport check is **essential** for correctness while link layer check is **not essential** but **valuable** for performance.
- Sliding window protocol:
  - Link layer: stop and wait usually suffices, e.g., 802.11
  - Transport layer: use larger window size

# Elements of Transport Protocols: Error Control and Flow Control

Buffering

- The sender needs buffering to hold all transmitted but as yet not acked segments which may lost and need to be retransmitted
- The receiver may or may not dedicate specific buffers

**The best trade-off between sender buffering and receiver buffering** depends on the type of traffic carried by the connection.

- For *low-bandwidth bursty* traffic, such as that produced by an interactive terminal, not to dedicate any buffers.
- For *high-bandwidth* traffic, e.g. file transfer, it is better if the receiver does delicate a full window of buffers. TCP uses.

**How to organize buffers?**

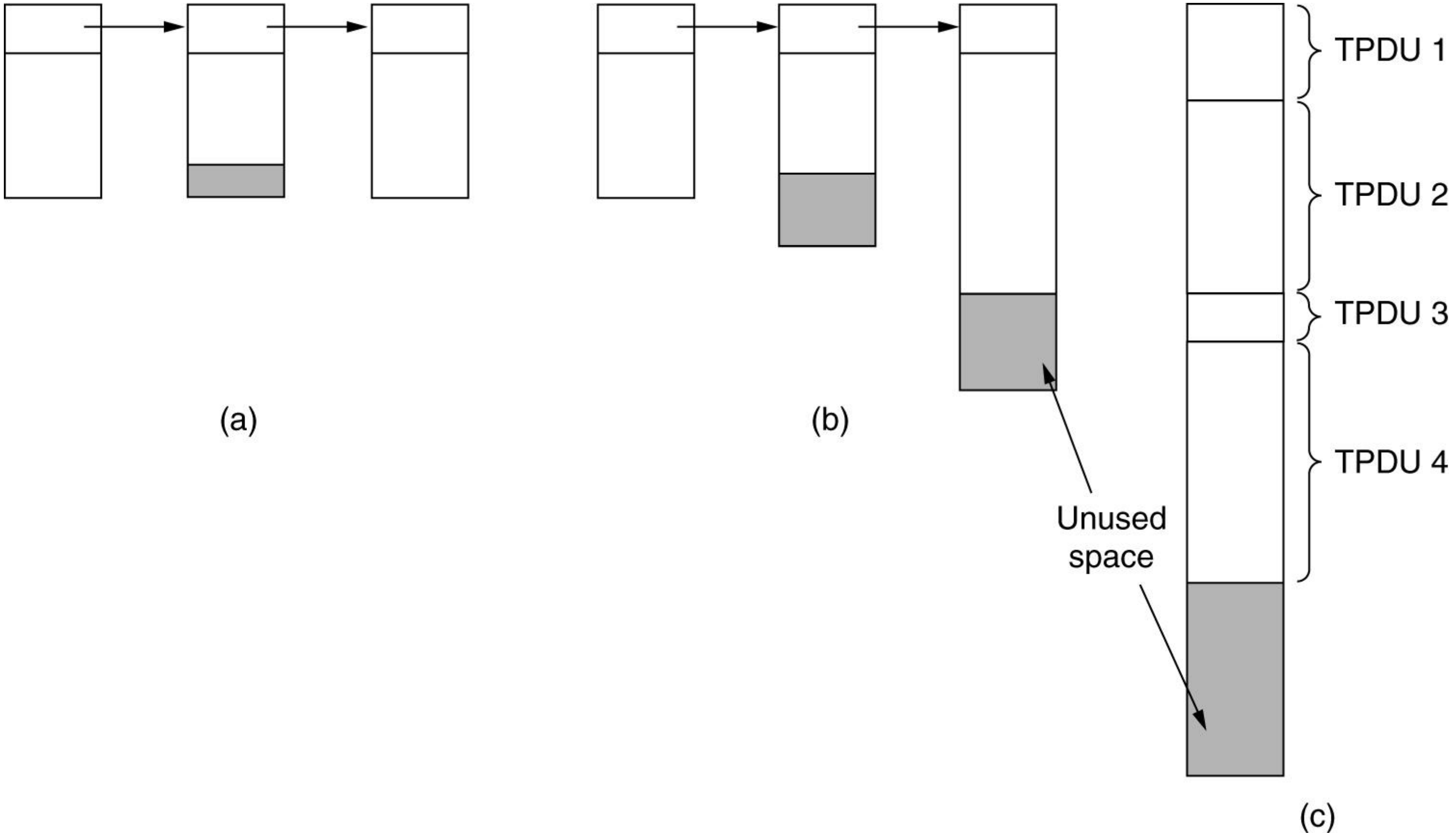
# Elements of Transport Protocols: Flow control and buffering: (The receiver buffering)

Buffer sizes

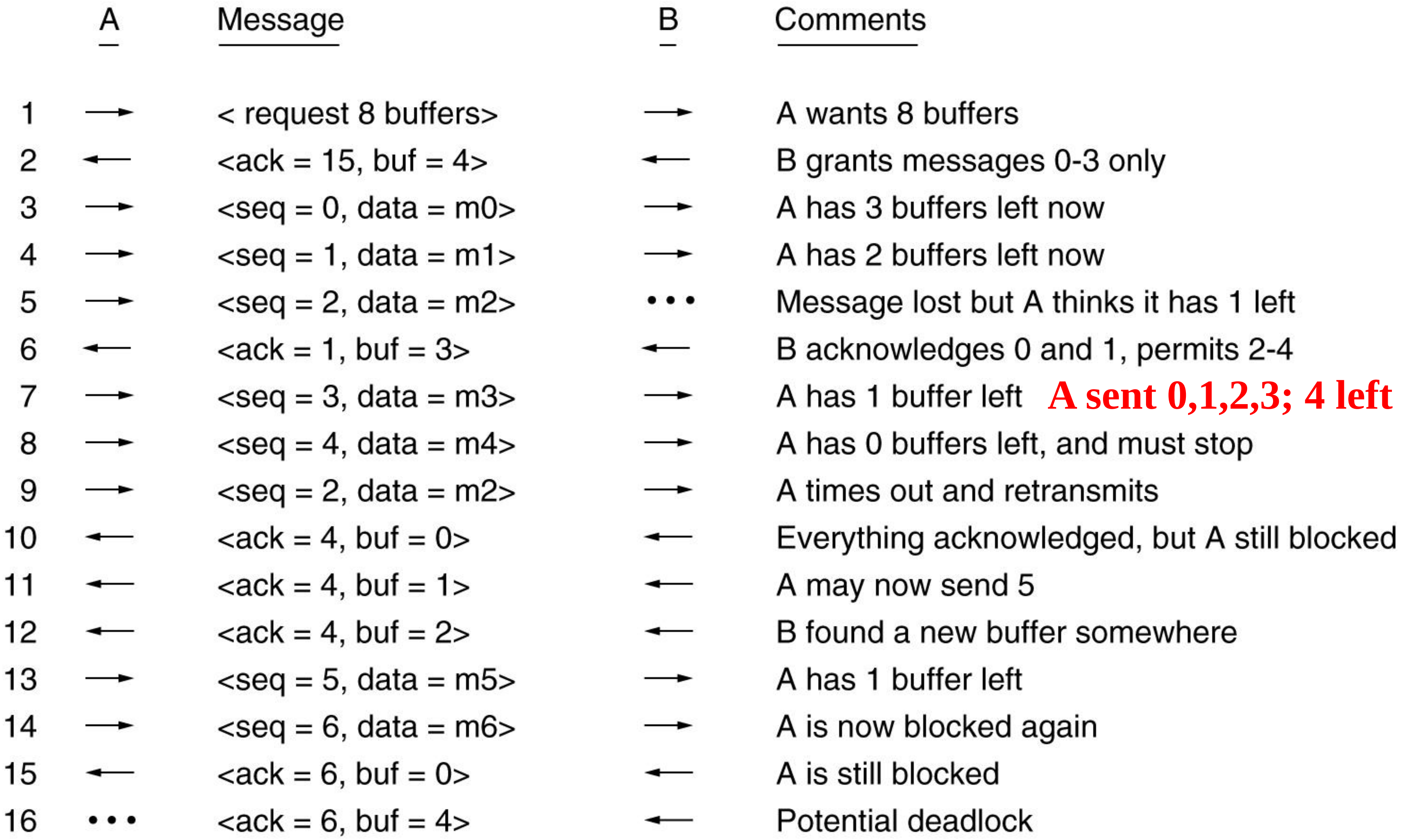
(a) Chained fixed-size buffers.

(b) Chained variable-sized buffers.

(c) One large circular buffer per connection.



# Elements of Transport Protocols: Flow control and buffering Dynamic buffer allocation (receiver's buffering capacity).



| | 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 **A sent 0,1,2,3; 4 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 |

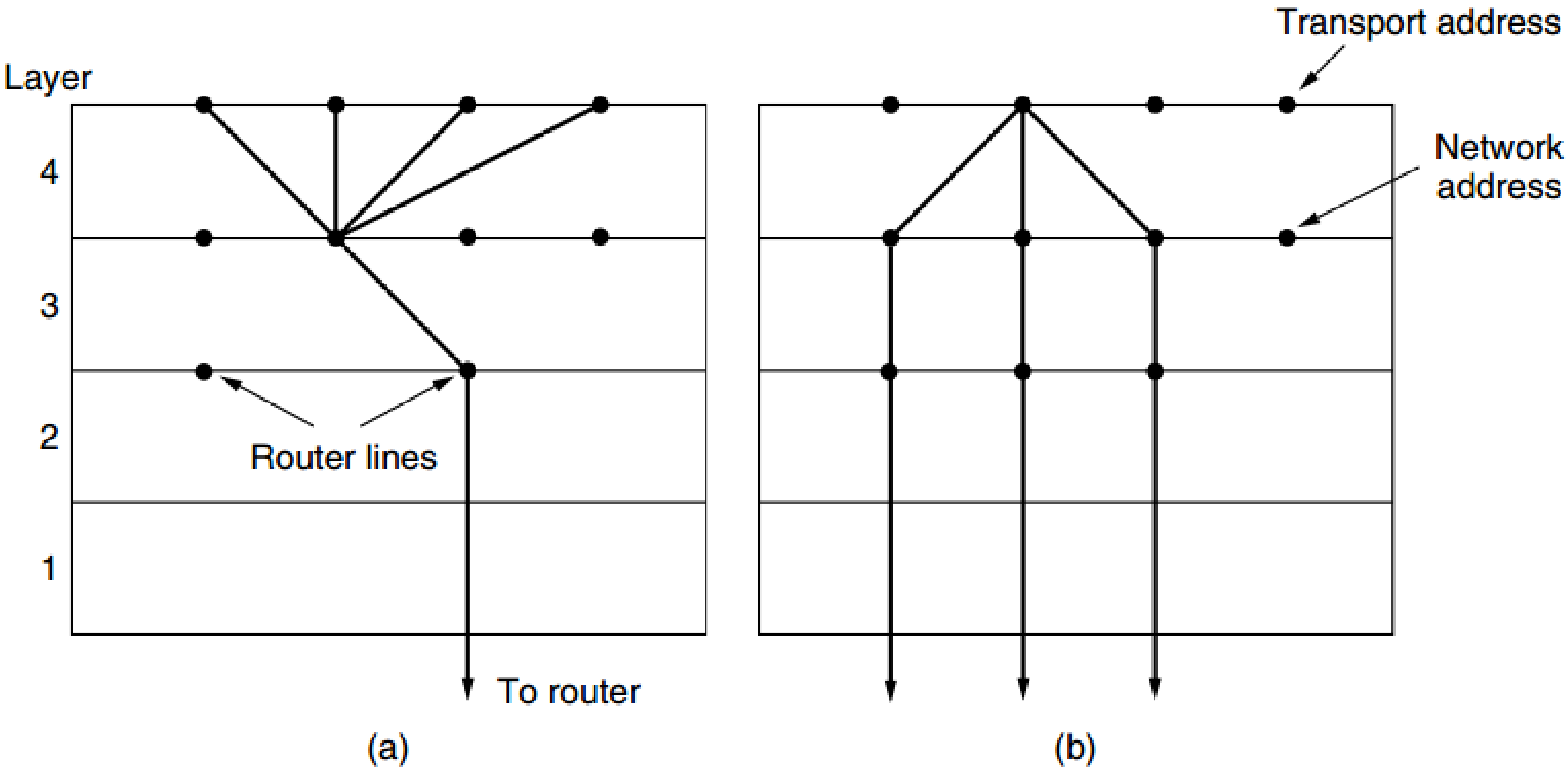
# Elements of Transport Protocols: Flow control and buffering

- When buffer space no longer limits the maximum flow, another bottleneck will appear: the carrying capacity of the subnet.
  - The sender dynamically adjusts the window size to match the network's carrying capacity.
  - In order to adjust the window size periodically, the sender could monitor both parameters and then compute the desired window size.

# Elements of Transport Protocols: Multiplexing

- **Multiplexing**: sharing several conversations over connections, virtual circuits, and physical links, e.g., four distinct transport connections all use the same network connection (e.g., IP address) to the remote host
- **Demultiplexing**. e.g., when a segment comes in, some way is needed to tell which process to give it to
- Two multiplexing:
  - multiplexing
  - *Inverse* multiplexing. e.g., **SCTP** (Stream Control Transmission Protocol), which can run a connection using multiple network interfaces.

# Elements of Transport Protocols: Multiplexing



# Elements of Transport Protocols: Crash Recovery

- Recovery from network and router crash is *straightforward*. The transport entities expect lost segments and know how to cope with them by **using retransmissions**
- Recovery from host crash is *difficult*.
- Assume that a client is sending a long file to a file server using a simple **stop-and-wait** protocol.
  - Part way through the transmission, the server crashes.
  - The server might send a broadcast segments to all other hosts, announcing that it had just crashed and requesting that its clients inform it of the status of all open connections.
  - Each client can be in one of **two states**: one segment outstanding, *S1*, or no segments outstanding, *S0*.

# Elements of Transport Protocols: Crash Recovery

- Some situations
  - The client should retransmit only if has an unacknowledged segment outstanding (S1) when it learns the crash.
    - If a crash occurs after the ack has been sent but before the write has been done, the client will receive the ack and thus be in state S0. → LOST. Problem!
    - If a crash occurs after the write has been done but before the ack has been sent, the client will not receive the ack and thus be in state S1. → Dup. Problem!
    - For more, see the next slide

# Elements of Transport Protocols: Crash Recovery

| Strategy used by sending host | Strategy used by receiving host | | | | | |
|---|---|---|---|---|---|---|
| | First ACK, then write | | | First write, then ACK | | |
| | 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 |

OK = Protocol functions correctly
DUP = Protocol generates a duplicate message
LOST = Protocol loses a message

# Elements of Transport Protocols: Crash Recovery

- One may ask: always retransmit and use seqno to detect duplicates?

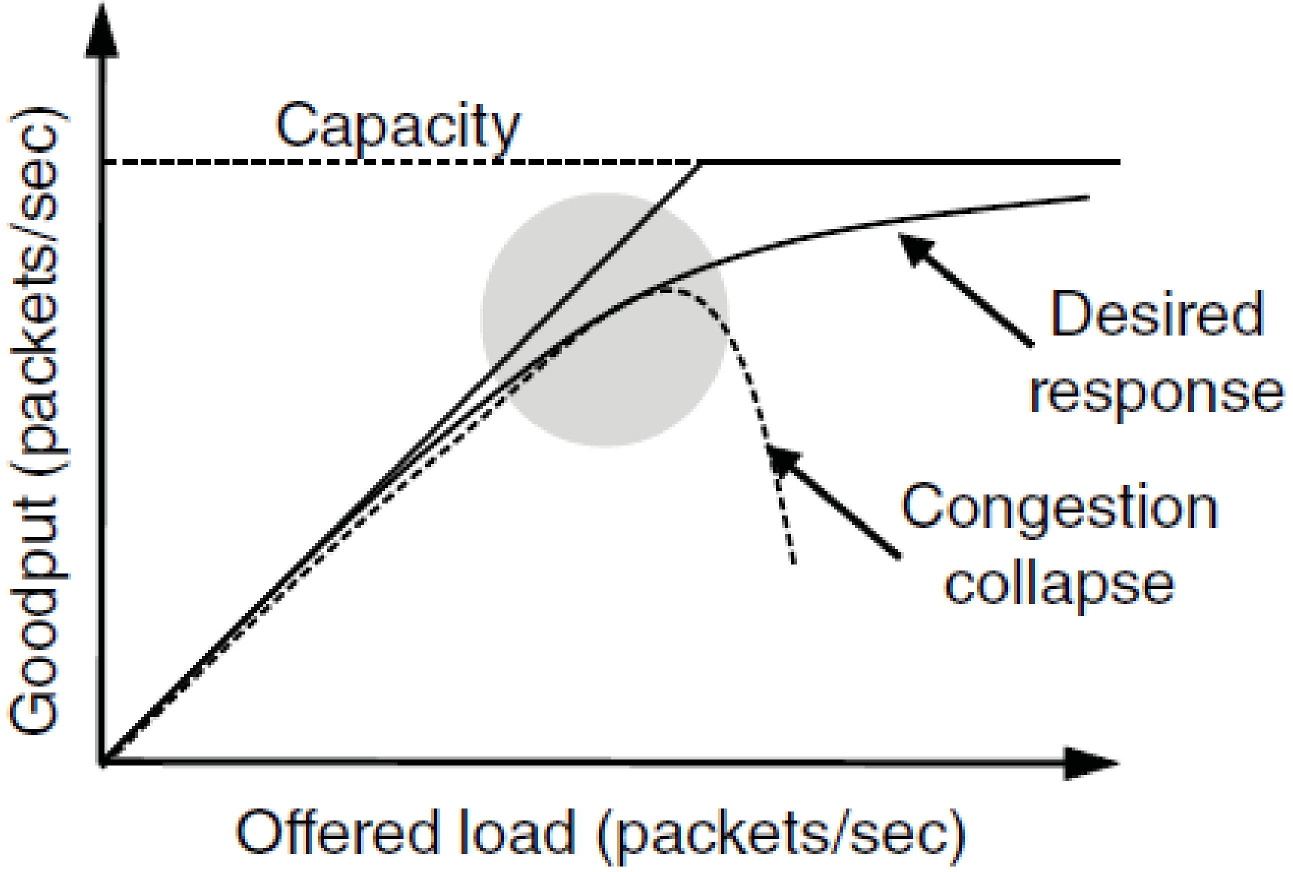
- **Observation**: No matter how the sender and receiver are programmed, there are always situations where the protocol fails to recover properly.
- **Conclusion:** in more general terms, recovery from a layer *N* crash can only be done by layer *N*+1 and only if the higher layer retains enough status information.

# Congestion Control

- Desirable Bandwidth Allocation
- Regulating the sending rate
- Wireless Issues

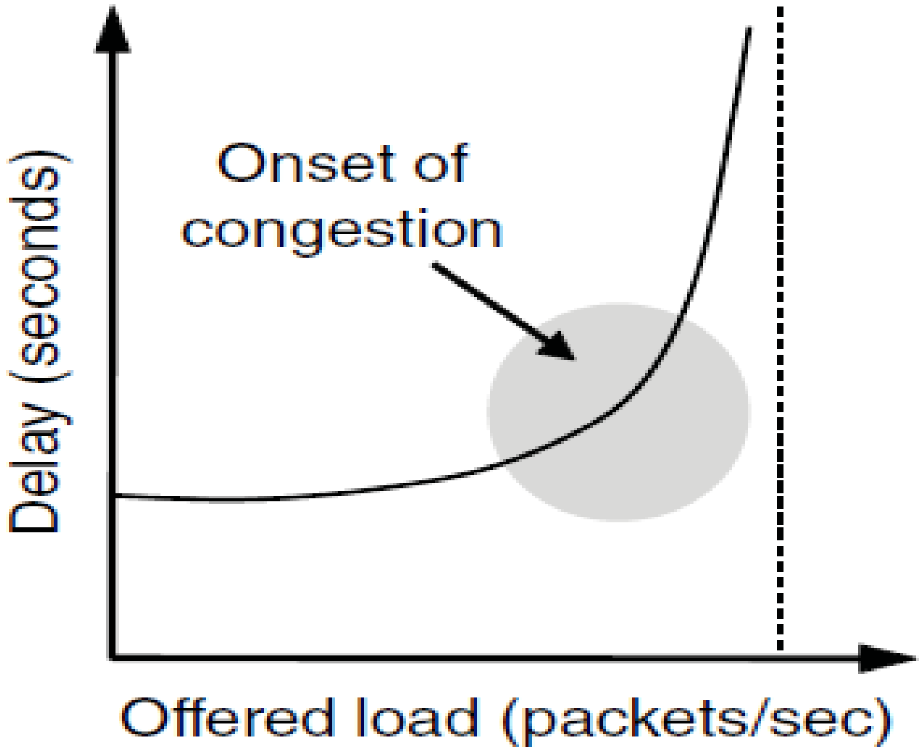
# Congestion Control:
## Desirable Bandwidth Allocation

- Goodput: rate of useful packets arriving at the receiver



# Congestion Control:
## Desirable Bandwidth Allocation

- Delay as a function of offered load



# Congestion Control:
## Desirable Bandwidth Allocation

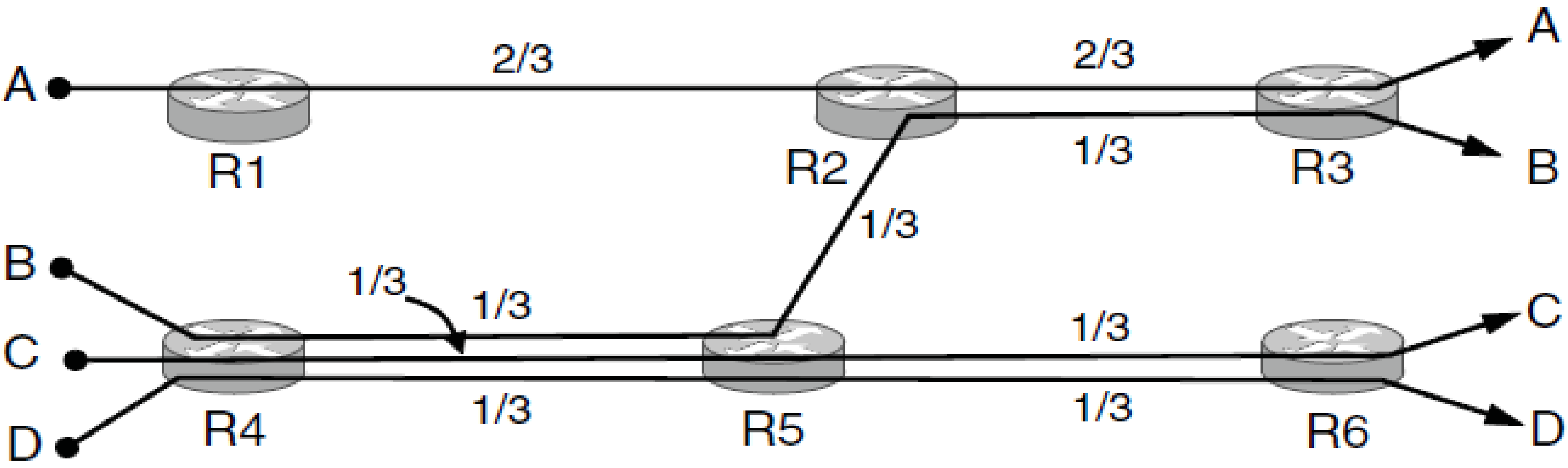
- To analyze the desirable bandwidth allocation, Kleinrock (1979) proposed the metric of **power**,

$$power = \frac{load}{delay}$$

- **Power**
  - will initially rise with offered load, as delay remains small and roughly constant,
  - but will reach a maximum and
  - fall as delay grows rapidly.
- The load with the highest power represents an efficient load for the transport entity to place on the network.

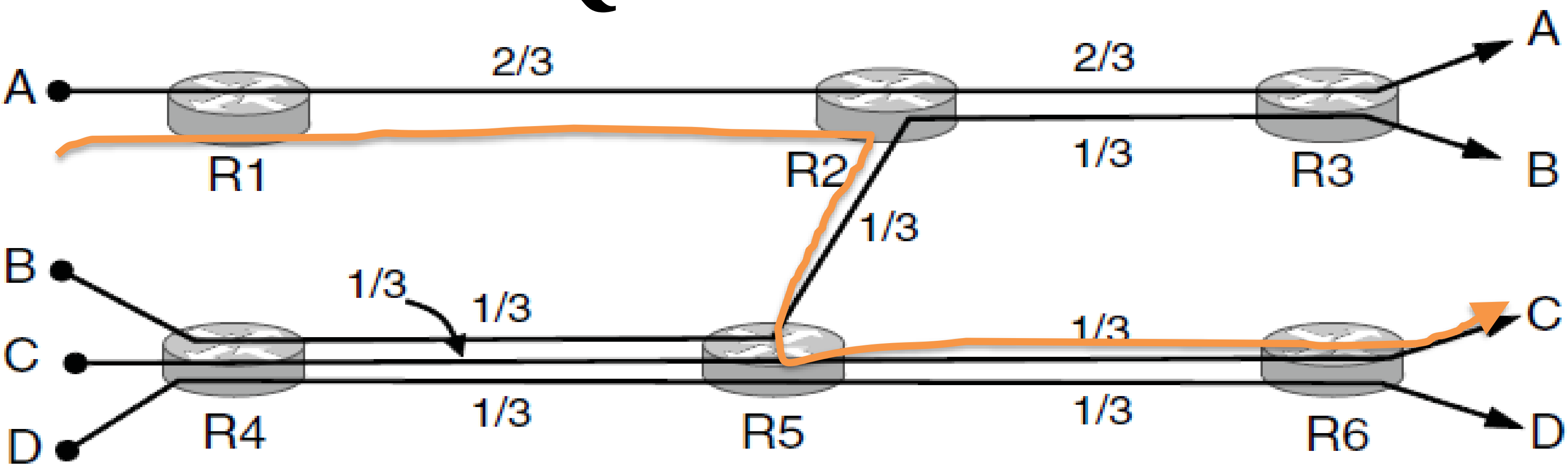
# Congestion Control:
## Max-Min Fairness

- How to divide bandwidth between different transport senders:
  - The first consideration is to ask what this problem has to do with congestion control.
  - A second consideration is what a fair portion means for flows in a network. The form of fairness that is often desired for network usage is **max-min** fairness.
  - A third consideration is the level over which to consider fairness.



- Assume the bandwidth of each link is unit 1.
- A **max-min** bandwidth allocation for 4 flows.

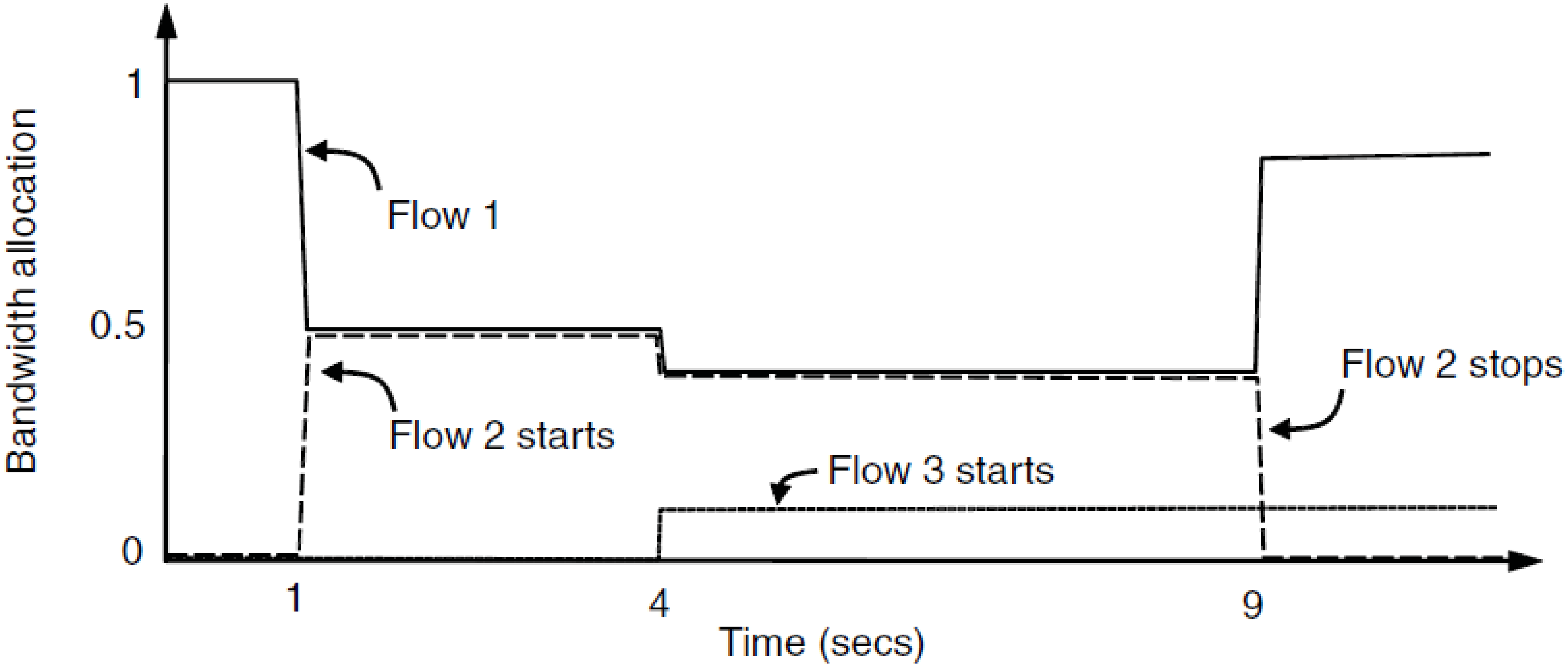
# Question



- In the above figure, suppose a new flow *E* is added that takes a path from *R1* to *R2* to *R6*. How does the max-min bandwidth allocation change for the five flows?

# Congestion Control:
## Desirable Bandwidth Allocation

- **Convergence**: A final criterion is that the congestion control algorithm converge quickly to a fair and efficient allocation of bandwidth.

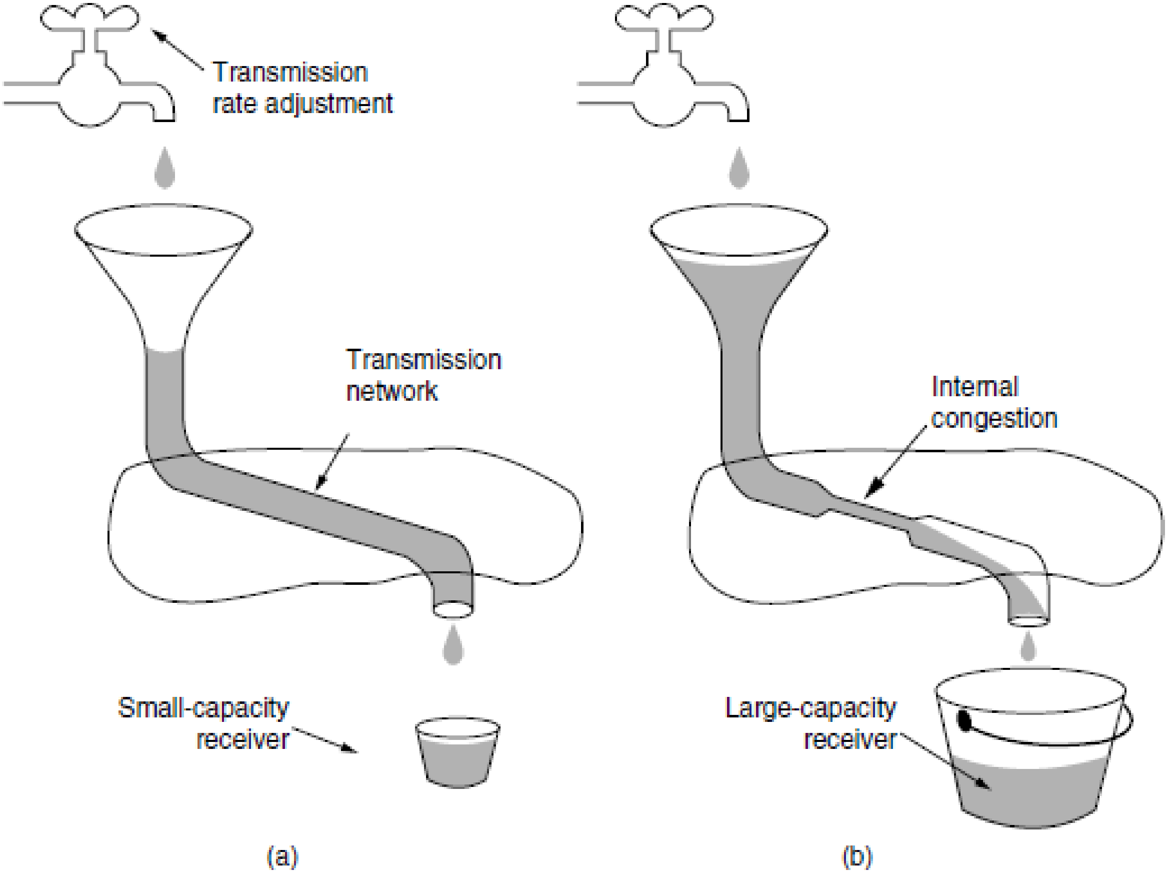


# Congestion Control:

## Regulating the sending rate

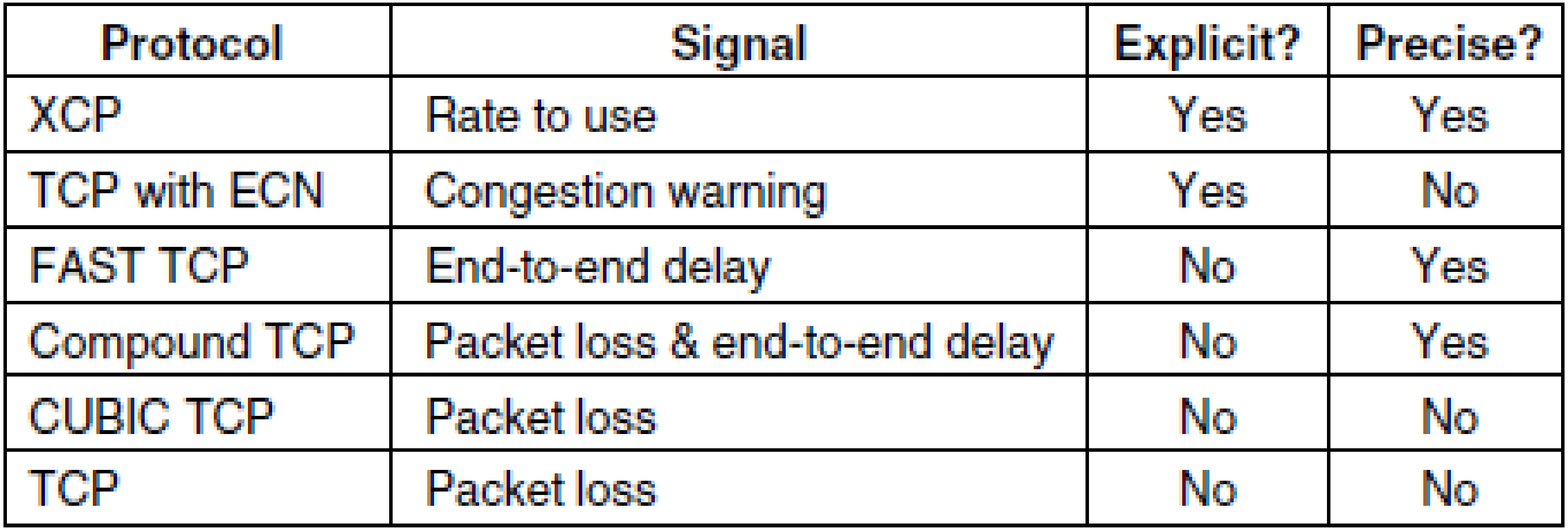
(a) A fast network feeding a low-capacity receiver.

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



# Congestion Control:
## Regulating the sending rate

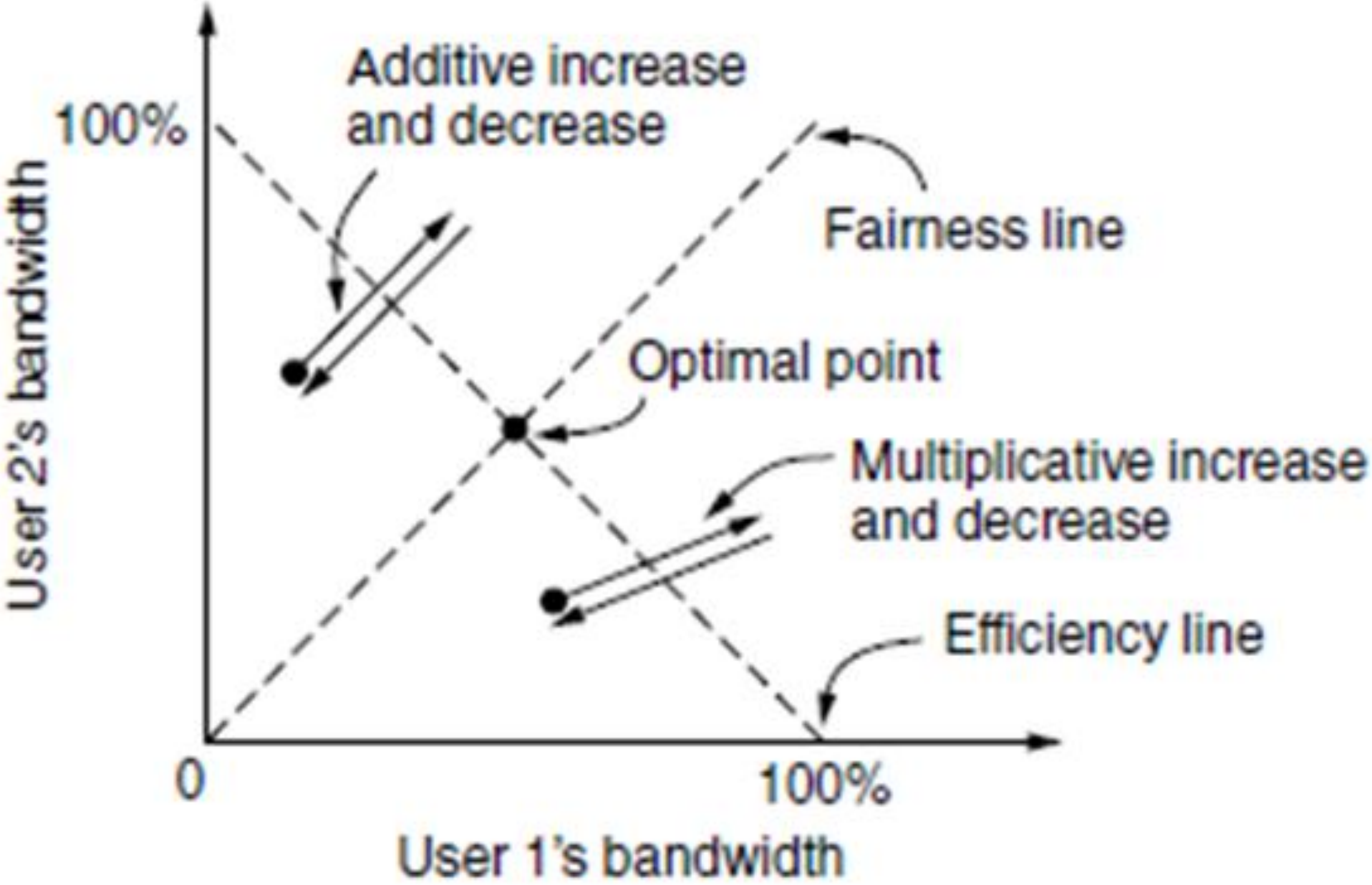
- Some congestion control protocols



| Protocol | Signal | Explicit? | Precise? |
|---|---|---|---|
| XCP | Rate to use | Yes | Yes |
| TCP with ECN | Congestion warning | Yes | No |
| FAST TCP | End-to-end delay | No | Yes |
| Compound TCP | Packet loss & end-to-end delay | No | Yes |
| CUBIC TCP | Packet loss | No | No |
| TCP | Packet loss | No | No |

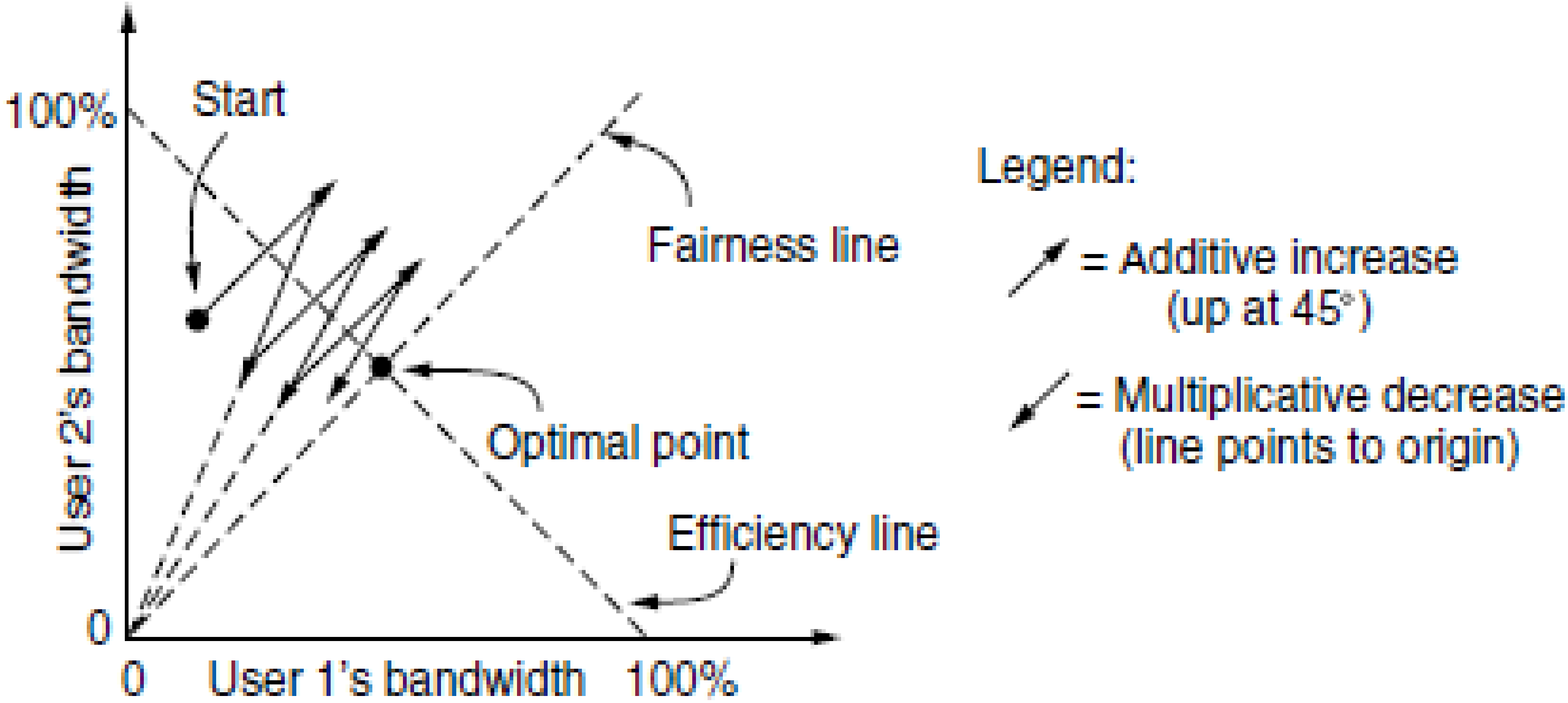
# Congestion Control:

## Regulating the sending rate



Additive and multiplicative bandwidth adjustments.

# Congestion Control:
## Regulating the sending rate (TCP)



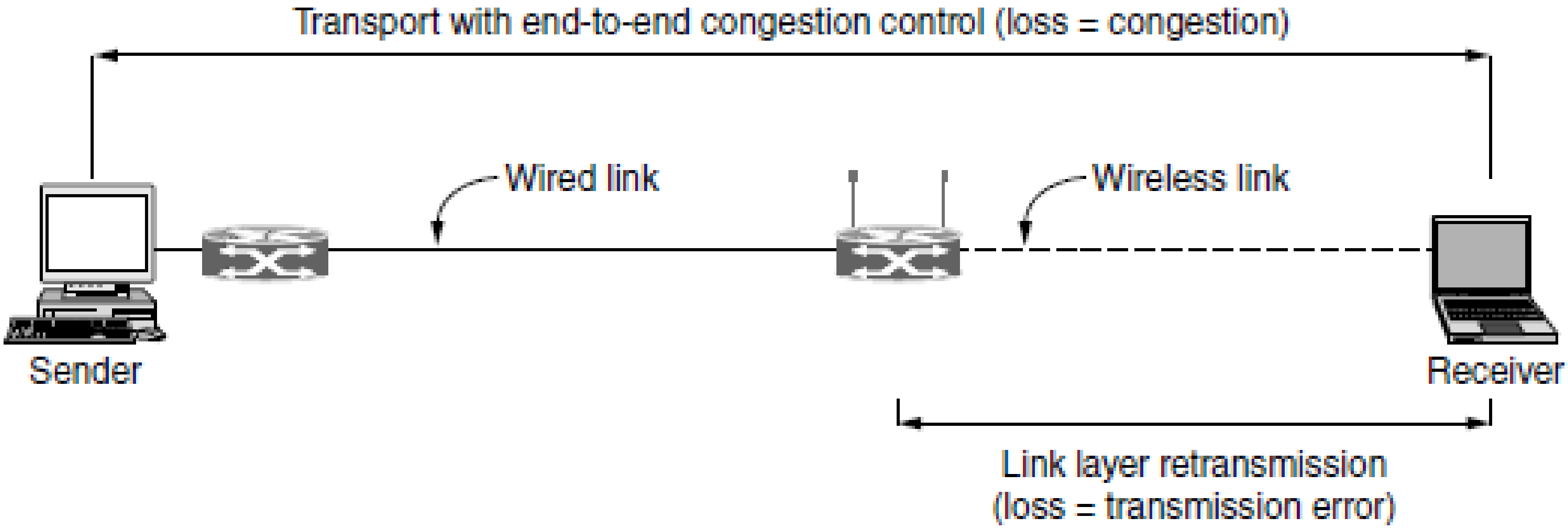
Additive Increase Multiplicative Decrease (AIMD) control law.

# Congestion Control:
## Wireless Issues

- Analysis by Padhye *et al*. (1998) show that the throughput goes up as the inverse square-root of the packet loss rate.
- What this means in practice is that the loss rate for fast TCP connections is very small; 1% is a moderate loss rate, and by the time the loss rate reaches 10% the connection has effectively stopped working.
- However, for wireless networks such as 802.11 LANs, frame loss rates of at least 10% are common.
- This difference means that, absent protective measures, congestion control schemes that use packet loss as a signal will unnecessarily throttle connections that run over wireless links to very low rates.

# Congestion Control: Wireless Issues

- There are two aspects to note
  - First, the sender does not necessarily know that the path includes a wireless link, since all it sees is the wired link to which it is attached.
  - The second aspect is a puzzle. The figure shows two mechanisms that are driven by loss: link layer frame retransmissions, and transport layer congestion control.
- The solution is that the two mechanisms act at different **timescales**


Transport with end-to-end congestion control (loss = congestion)
Wired link
Wireless link
Sender
Receiver
Link layer retransmission
(loss = transmission error)

Congestion control over a path with a wireless link.

# Previously on Computer Networks

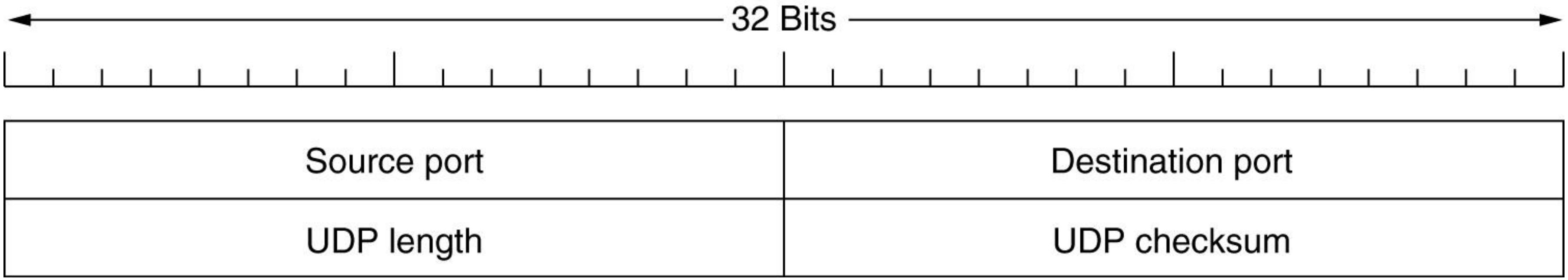
- Transport services, primitives
- Berkeley sockets
- Addressing
- Connection establishment
- Connection release
- Error control and flow control
- Crash recovery

# THE INTERNET TRANSPORT PROTOCOLS: UDP, RPC, RTP

- Two main transport protocols in the Internet
  - Connectionless protocol (UDP)
  - Connection-oriented protocol (TCP)
- UDP (User Datagram protocol)
- RPC (Remote Procedure Call)
- RTP (Real-time Transport Protocol)

# The Internet Transport Protocols: UDP

- UDP (RFC768) provides a way for applications to
  - send encapsulated IP datagrams and
  - send them without having to establish a connection.
- UDP transmits segments consisting of an 8-byte header followed by the payload.



32 Bits

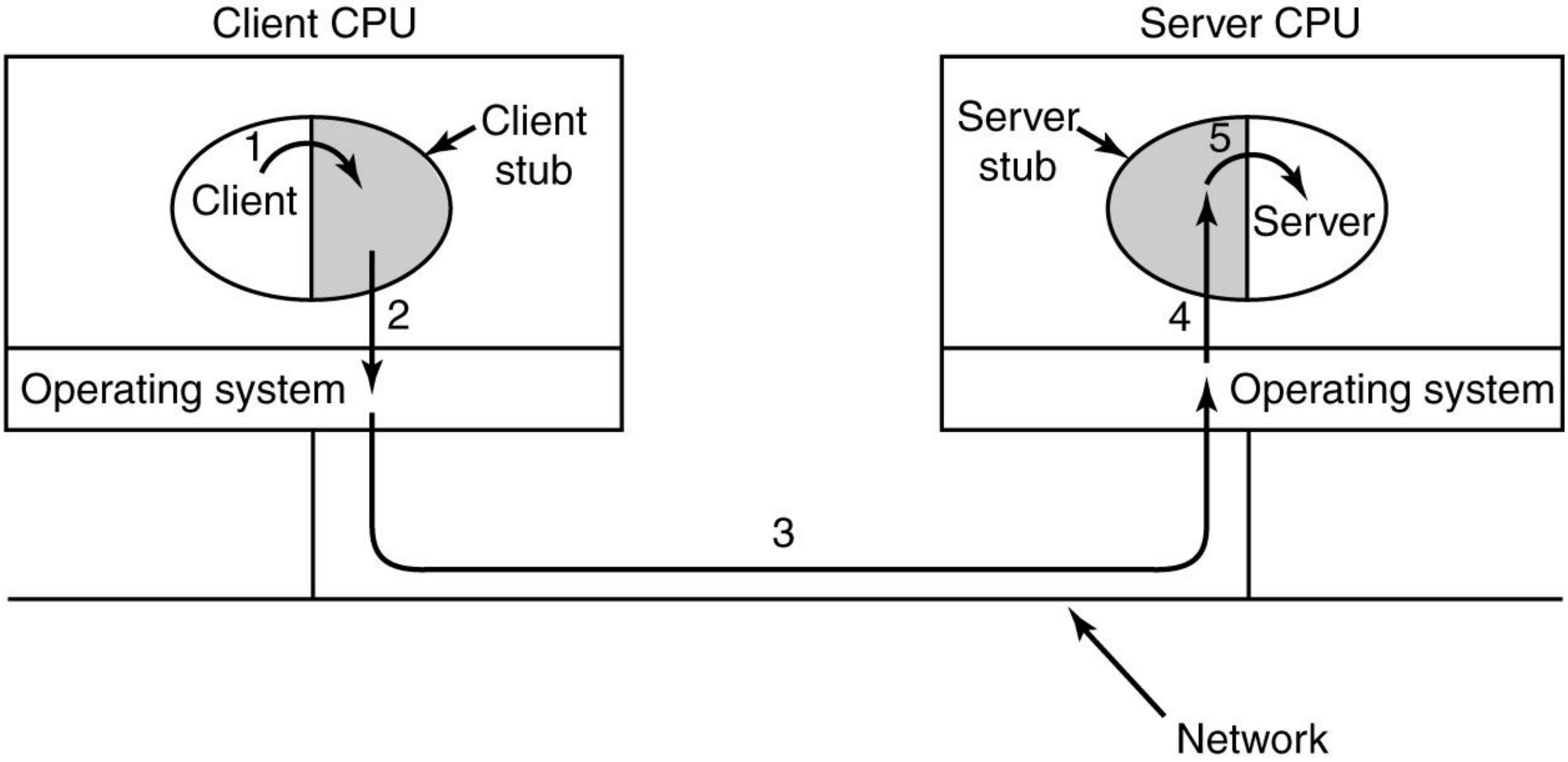
| | |
|---|---|
| Source port | Destination port |
| UDP length | UDP checksum |

- UDP
  - Multiplexing and demultiplexing using ports.
  - No flow control, error control or retransmission.
  - Applications: RTP, DNS (Domain Name System)

# The Internet Transport Protocols: RPC/UDP

- RPC (Remote Procedure Call) allows programs to call procedures located on remote hosts.
  - When a process on machine 1 calls a procedure on machine 2, the calling process on 1 is suspended and execution of the called procedure takes place on 2
  - Information can be transported from the caller to the callee in the parameters and can come back in the procedure result.
  - No message passing is visible to the programmer.
- **The idea behind RPC is to make a remote procedure call look as much as possible like a local one.**

# The Internet Transport Protocols: RPC/UDP



C Proc ←→ C Stub ←→ S Stub ←→ S Proc

# The Internet Transport Protocols: RPC/UDP

The steps in making an RPC

- Step 1 is the client calling the client stub.
- Step 2 is the client stub packing the parameters into a message (marshaling) and making a system call to send the message.
- Step 3 is the kernel sending the message from the client machine to the server machine.
- Step 4 is the kernel passing the incoming packet to the server stub and unpacking the packet to extract the parameters (unmarshaling).
- Step 5 is the server stub calling the server procedure with the unmarshaled parameters.

- The reply traces the same path in the other direction.

# The Internet Transport Protocols: RPC/UDP

- A few snakes hiding under the grass (RPC)
  - The use of pointer parameters.
  - Some problems for weakly-typed languages (The length of an array).
  - It is not always possible to deduce the types of the parameters, not even from a formal specification or the code itself. (printf)
  - The use of global variables.
- **→Some restrictions are needed to make RPC work well in practice.**
- **RPC/TCP vs RPC/UDP**

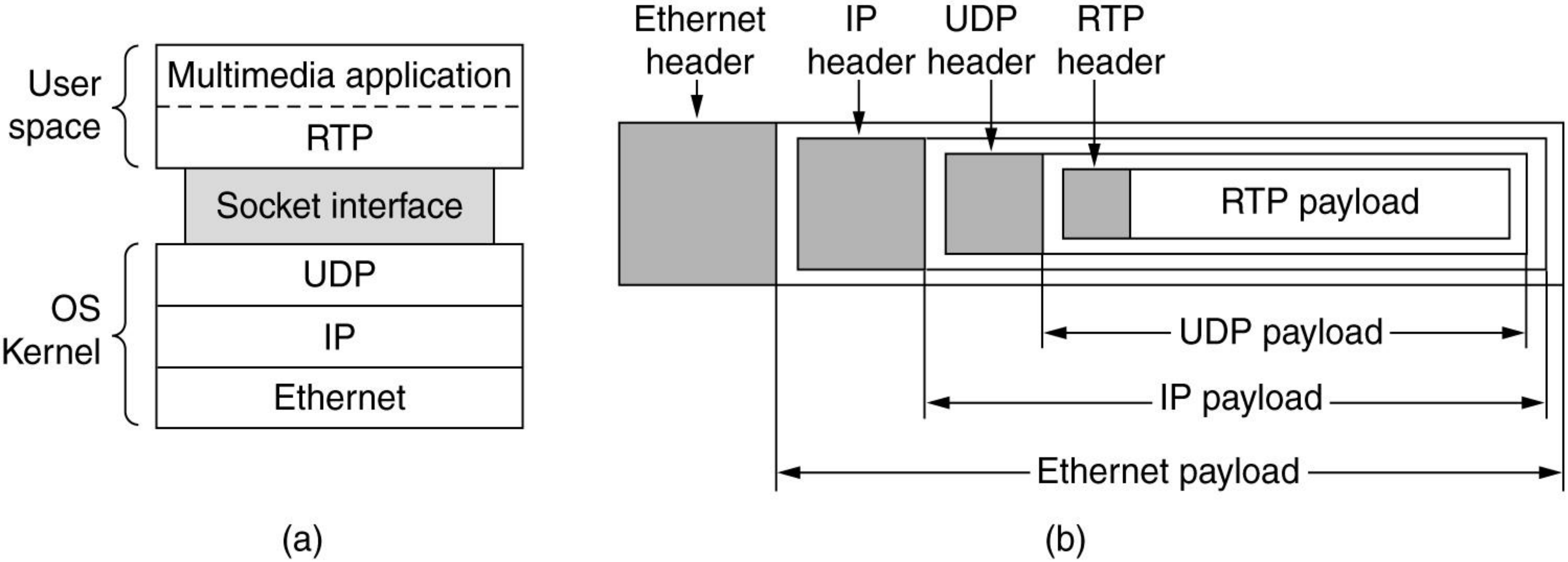
# The Internet Transport Protocols: RTP/UDP

- Multimedia applications such as Internet radio, Internet telephony, music-on-demand, videoconferencing, video-on-demand, require real-time transport protocols → RTP (Real-time Transport Protocol) (RFC1889)
- The RTP is in user space and runs over UDP. The RTP Ops:
  - The multimedia applications consist of multiple audio, video, text, and possibly other streams. These are fed into the RTP library.
  - This library then multiplexes the streams and encodes them in RTP packets, which it then puts into a socket.
  - UDP packets are generated and embedded in IP packets.
  - The IP packet are then put in frames for transmission.
  - …

# The Internet Transport Protocols: RTP/UDP

(a)The position of RTP in the protocol stack

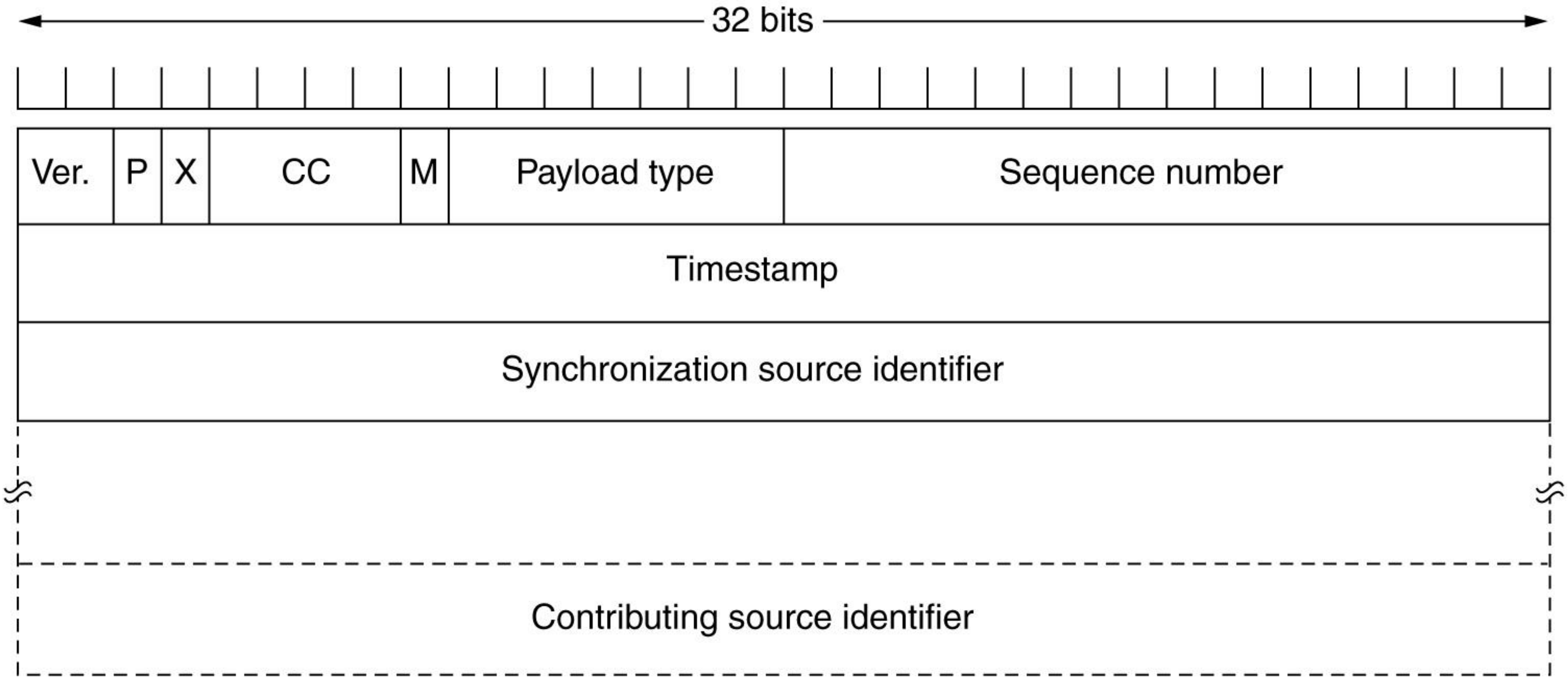
(b) packet nesting



# The Internet Transport Protocols: RTP/UDP

- RTP is a transport protocol realized in the application layer.
- RTP is to multiplex several real-time data streams onto a single stream of UDP packets and unicast or multicast the UDP packets.
- Each RTP packet  is given a number one higher than its predecessor. RTP has no flow control, no error control, no acknowledgements, and no retransmission support.
- Each RTP payload may contain multiple samples and they can be coded any way that the application wants. For example a single audio stream may be encoded as 8-bit PCM samples at 8kHz, delta encoding, predictive encoding, GSM encoding, MP3, and so on.
- RTP allows timestamping.

# The Internet Transport Protocols: RTP/UDP



32 bits

Ver. P X CC M Payload type Sequence number

Timestamp

Synchronization source identifier

Contributing source identifier

# The Internet Transport Protocols: RTP/UDP

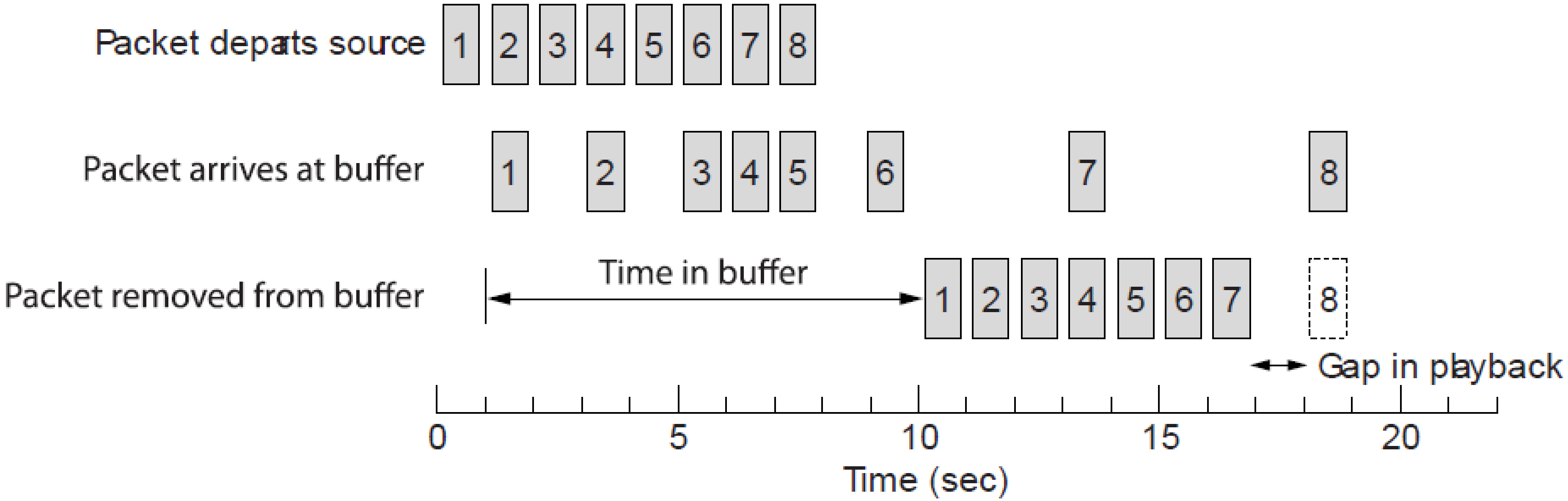
- Version: 2 bits, already at 2.
- P bit: padded to a multiple of 4 bytes or not.
- X bit: an extension header or not.
- CC: how many contributing sources are present (0-15).
- M bit: marker bit.
- Payload type: which encoding algorithm has been used.
- Sequence number: incremented on each RTP packet sent.
- Timestamp: reducing jitter.
- Synchronization source identifier: which stream the packet belongs to.
- Contributing source identifiers.

# The Internet Transport Protocols: RTP/UDP

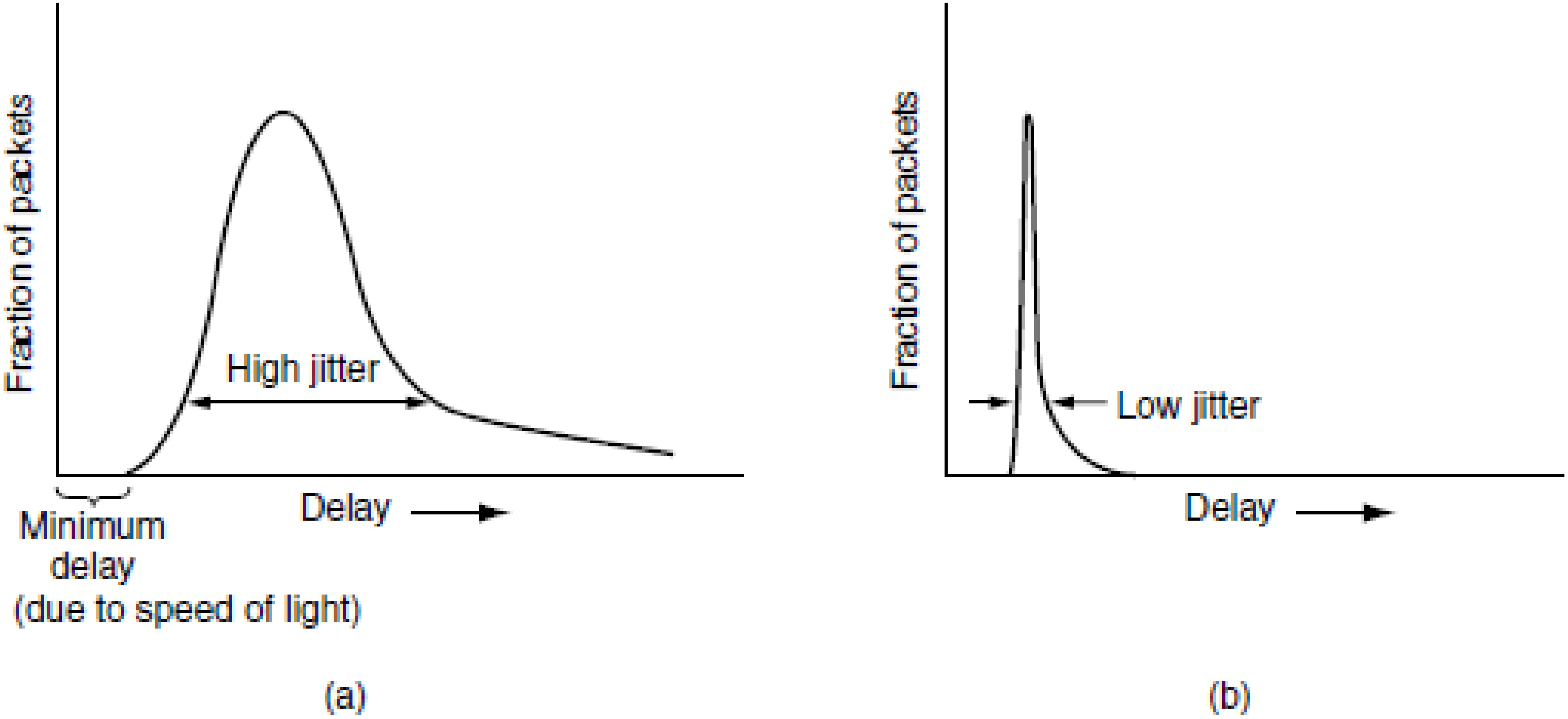
- RTCP (Real-time Transport Control Protocol) is a little sister  protocol (little sibling protocol?) for RTP.
  - Does not transport any data
  - To handle feedback, synchronization, and the user interface
  - To handle interstream synchronization.
  - To name the various sources.
- For more information about RTP,  *RTP: Audio and Video for the Internet*  (Perkins, C.E. 2002, Addison-Wesley)

# The Internet Transport Protocols: RTP/UDP

## Smoothing the output stream by buffering packets



# The Internet Transport Protocols: RTP/UDP



(a) High jitter. (b) Low jitter.

# THE INTERNET TRANSPORT PROTOCOLS: **TCP**

- Introduction to TCP
- The TCP Service Model
- The TCP Protocol
- The TCP Segment Header
- TCP Connection Establishment
- TCP Connection Release
- TCP Connection Management Modeling
- TCP Sliding Window
- TCP Timer Management
- TCP Congestion Control
- The Future of TCP

# TCP: Introduction

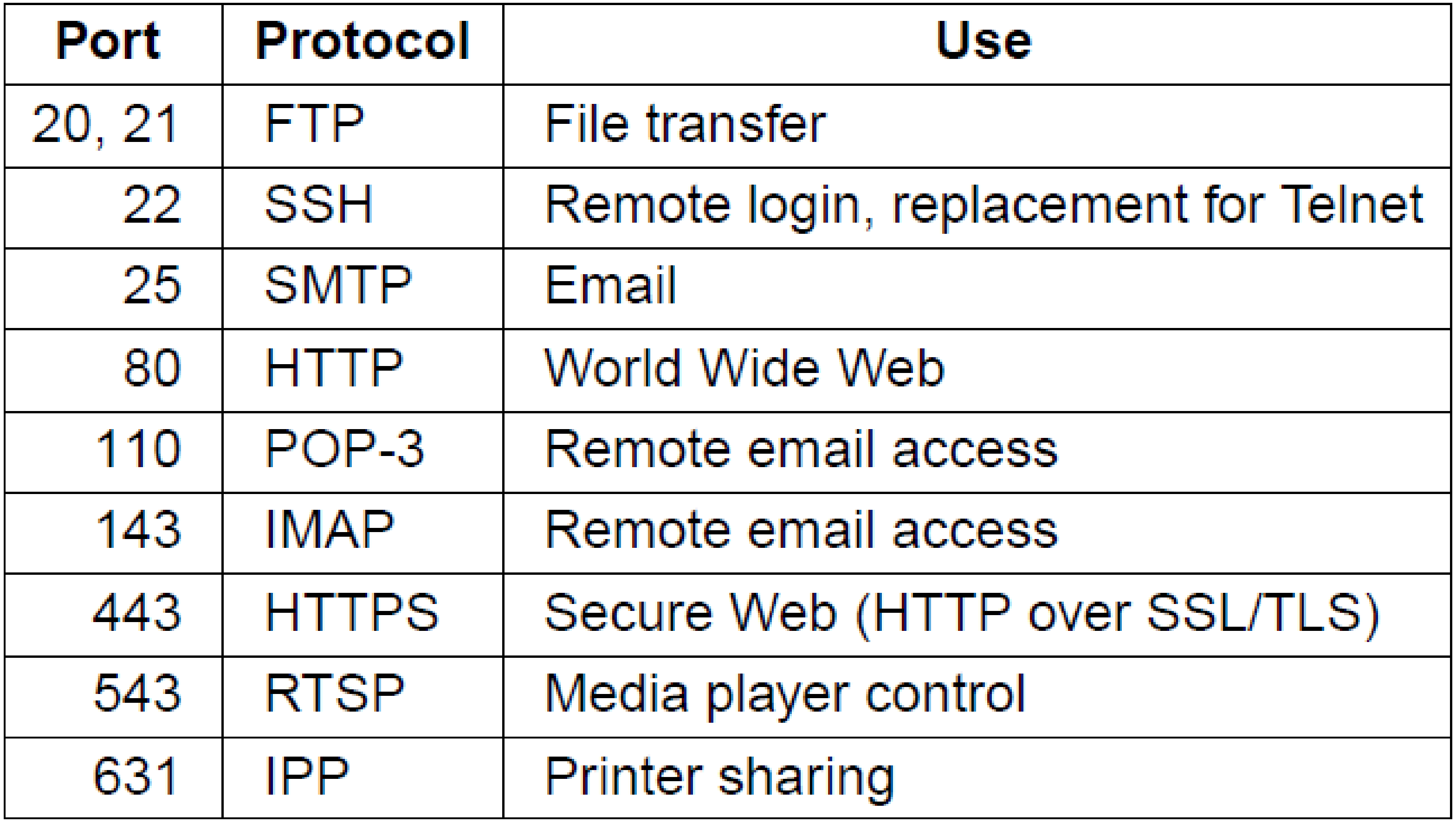
- TCP (Transmission Control Protocol) provides a *reliable end-to-end byte stream* over an *unreliable* internetwork. For TCP, see RFC 793, 1122, 1323, 2108,2581, 2873, 2988, 3168, 4614.
- The communication between TCP entities
  - A TCP entity accepts user data streams from local processes, breaks them up into pieces not exceeding 64KB (in practice, often 1500 − 20 − 20 data bytes), sends each piece as a separate IP datagram.
  - When datagrams containing TCP data arrive at a machine, they are given to the TCP entity, which constructs the original byte streams.
- TCP must furnish (提供) the reliability that most users want and that IP does not provide.

# TCP: The Service Model

- For any TCP service to be obtained, a connection must be explicitly established between a socket on the sending machine and a socket on the receiving machine.
  - Connections are identified by the socket identifiers at both ends, that is, (socket1, socket2)
  - A socket number (address) consisting of the IP address of the host and a 16-bit number local to that host, called a port.
  - Port numbers below 1024 are called well-known ports and are reserved for standard services. (see the next slide.)

# TCP: The Service Model

- Some well-known ports:
  23 for Telnet, 69 for TFTP, 79 for Finger, 119 for NNTP



| Port | Protocol | Use |
|---|---|---|
| 20, 21 | FTP | File transfer |
| 22 | SSH | Remote login, replacement for Telnet |
| 25 | SMTP | Email |
| 80 | HTTP | World Wide Web |
| 110 | POP-3 | Remote email access |
| 143 | IMAP | Remote email access |
| 443 | HTTPS | Secure Web (HTTP over SSL/TLS) |
| 543 | RTSP | Media player control |
| 631 | IPP | Printer sharing |

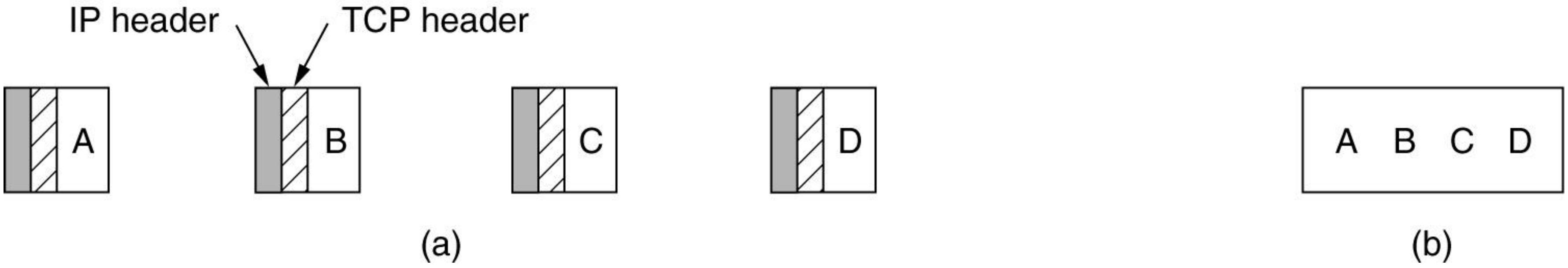
# TCP: The Service Model

- To have many daemons standby

- To have one master daemon inetd or xinetd standby
  - The master daemon attaches itself to multiple ports and wait for the first incoming connection
  - When one incoming connection request arrives, inetd or xinetd forks off a new process and executes the appropriate daemon on it, letting that daemon handle the request.

# TCP: The Service Model

- All TCP connections are full duplex and point-point.
- A TCP connection is a byte stream, not a message queue. Message boundaries are not preserved end to end.



(a) Four 512-byte segments sent as separate IP datagrams.

(b) The 2048 bytes of data delivered to the application in a single READ CALL.

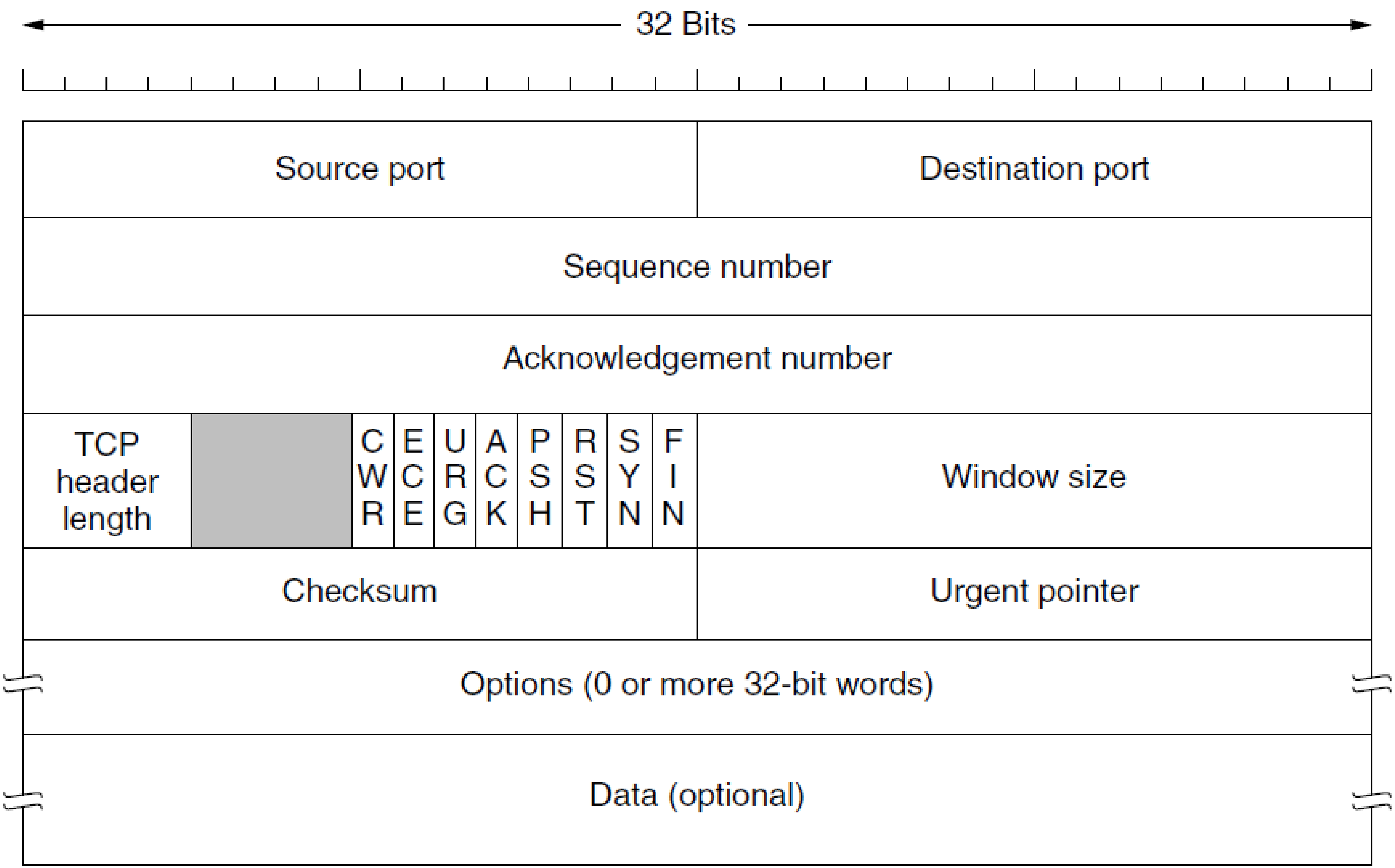
# TCP: The Service Model

- When an application passes data to TCP, TCP may send it immediately or buffer it at its own discretion.
  - To force data out, applications can use the PUSH flag, which tells TCP not to delay the transmission.
- TCP supports urgent data (now rarely used).
  - When the urgent data are received at the destination, the receiving application is interrupted (e.g., given a signal in UNIX terms) so it can stop whatever it was doing and read the data stream to find the urgent data.
    - The end of the urgent data is marked so the application knows when it is over.
    - The start of the urgent data is not marked. It is up to the application to figure that out.

# TCP: The Overview

- Every byte on a TCP connection has its own 32-bit sequence number.
- The sending and receiving TCP entities exchange data in the form of segments.
  - Each segment, including the TCP header, must fit in the 65515 (=65535-20) byte IP payload. (*Total length* in IP pkt is 16 bits long)
  - Each network has a maximum transfer unit or MTU (1500 for the Ethernet).
- The TCP entities use the *sliding window* protocol

# TCP: The Header



32 Bits

Source port

Destination port

Sequence number

Acknowledgement number

TCP header length

CWR

ECE

URG

ACK

PSH

RST

SYN

FIN

Window size

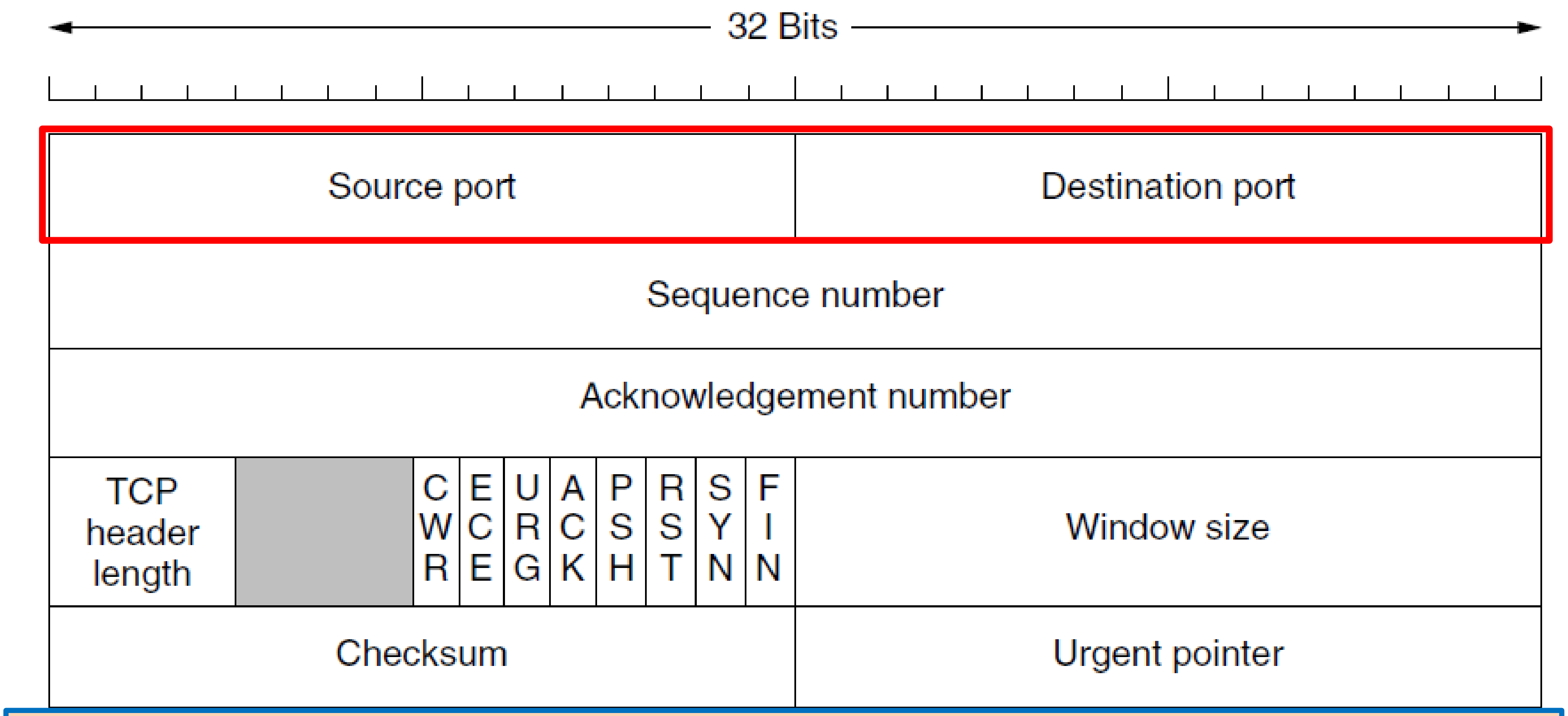
Checksum

Urgent pointer

Options (0 or more 32-bit words)

Data (optional)

# TCP: The Header



32 Bits

Source port | Destination port

Sequence number

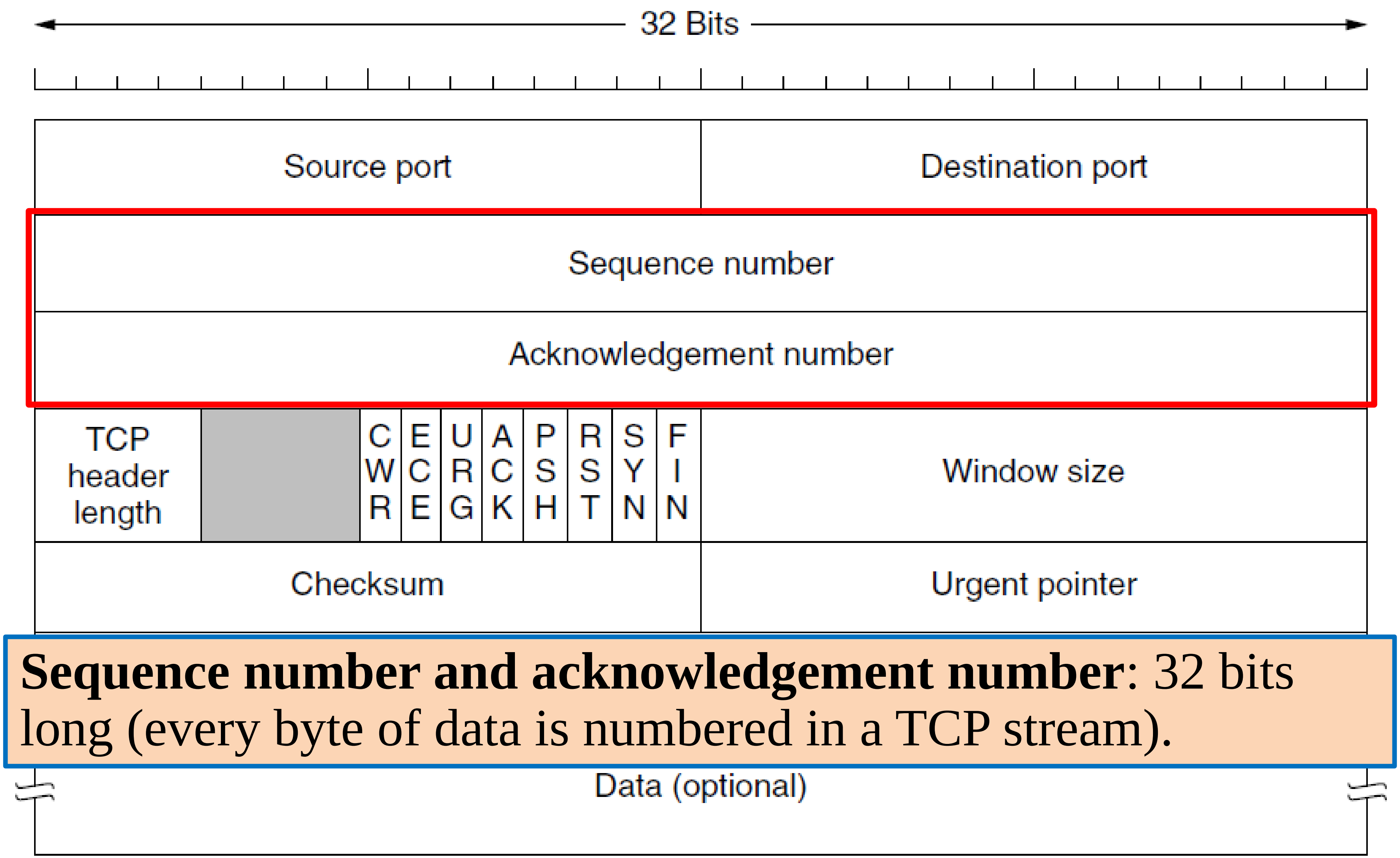
Acknowledgement number

TCP header length | | CWR | ECE | URG | ACK | PSH | RST | SYN | FIN | Window size

Checksum | Urgent pointer

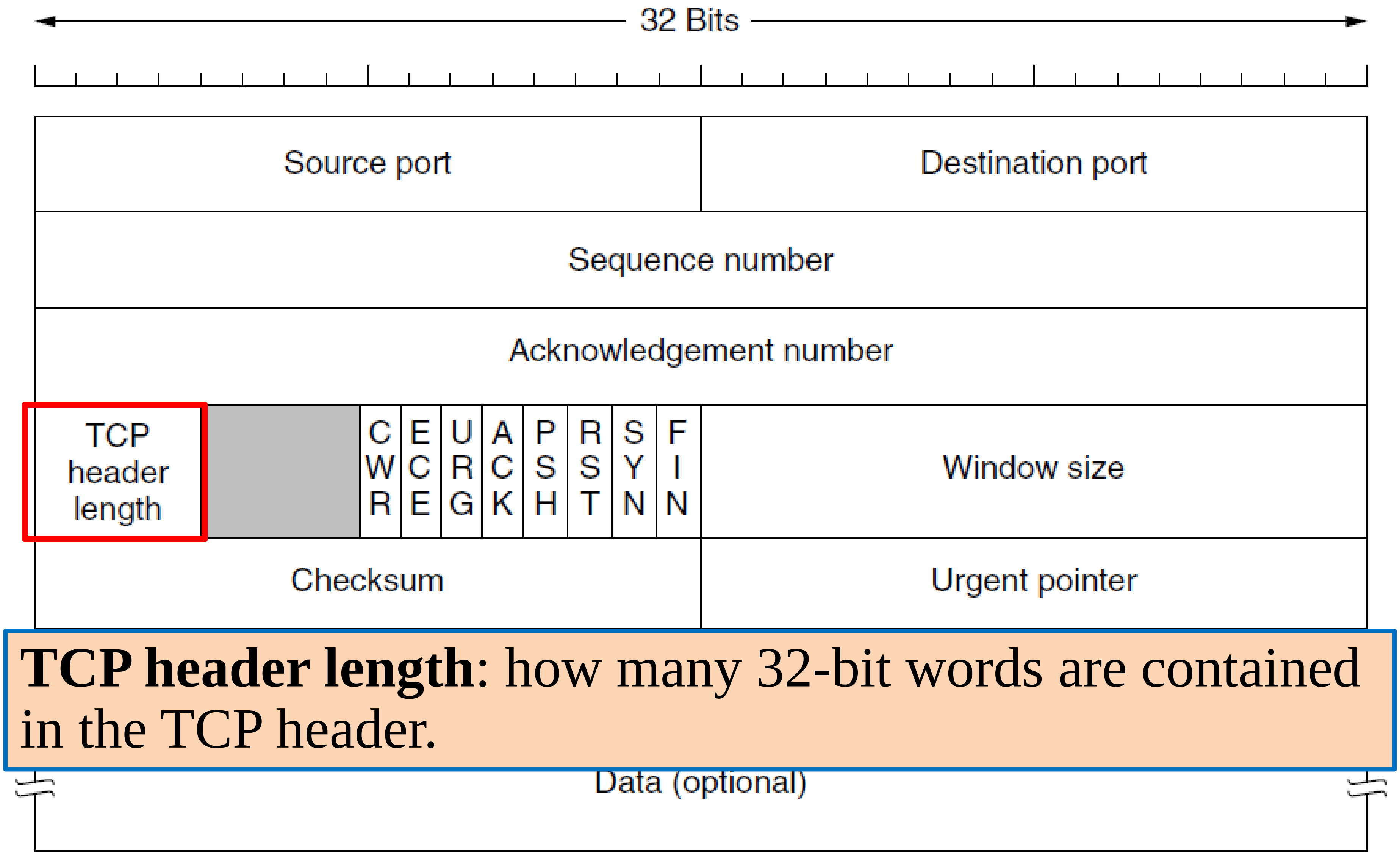
**Source port and destination port**: to identify the local end points of the connection. A port plus its host’s IP address forms a 48-bit unique end point. The source and destination end points together identify the connection.

# TCP: The Header



**Sequence number and acknowledgement number**: 32 bits long (every byte of data is numbered in a TCP stream).

# TCP: The Header



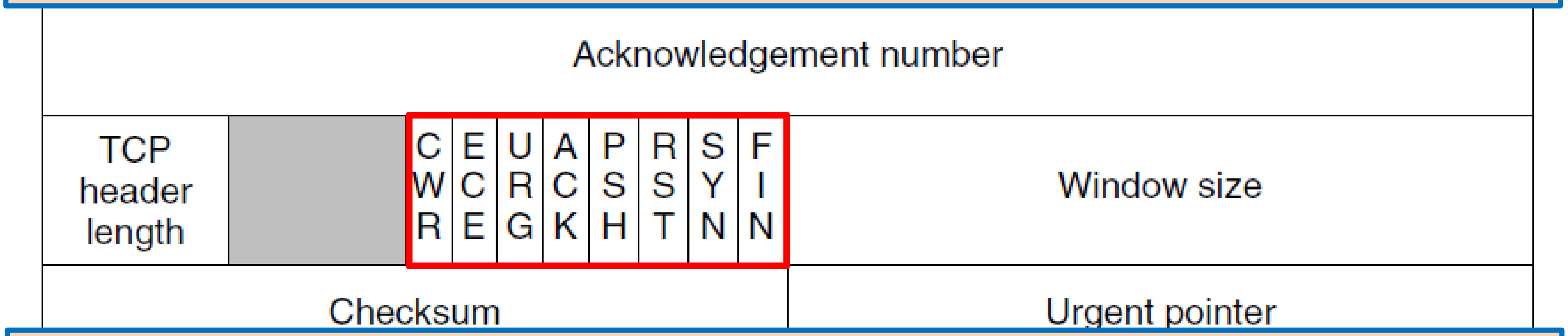
32 Bits

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Source port | | | | | | | | | Destination port |
| Sequence number | | | | | | | | | |
| Acknowledgement number | | | | | | | | | |
| TCP header length | | CWR | ECE | URG | ACK | PSH | RST | SYN | FIN | Window size |
| Checksum | | | | | | | | | Urgent pointer |
| Data (optional) | | | | | | | | | |

**TCP header length**: how many 32-bit words are contained in the TCP header.

# TCP: The Header

CWR: Congestion Window Reduced from a sender
ECE: ECN (Explicit Congestion Notification)-Echo to a sender
URG bit: the Urgent pointer is in use or not.
ACK bit: the Acknowledgement number is valid or not.
PSH bit: PUSHed data or not.



Acknowledgement number

TCP header length

CWR

ECE

URG

ACK

PSH

RST

SYN

FIN

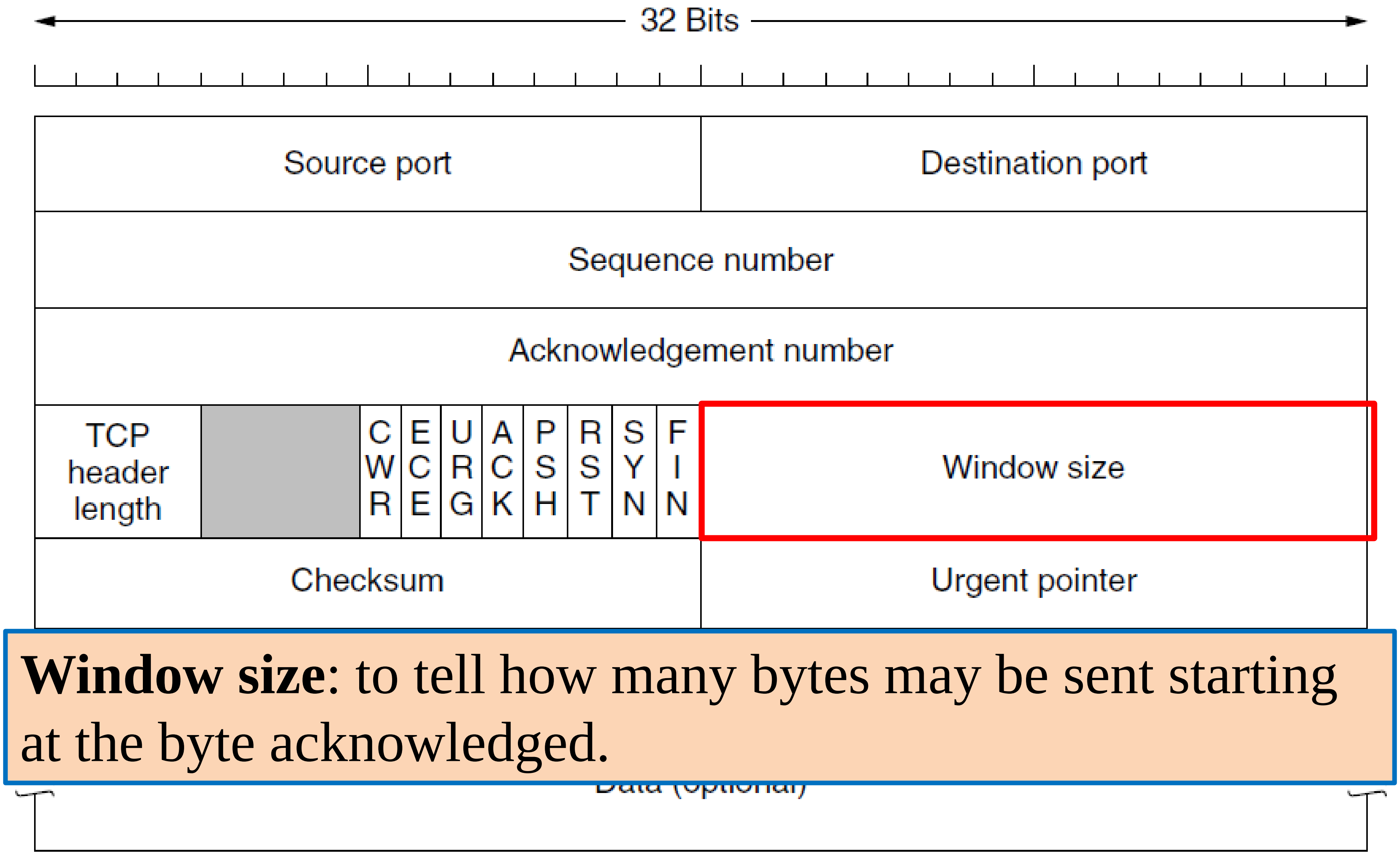
Window size

Checksum

Urgent pointer

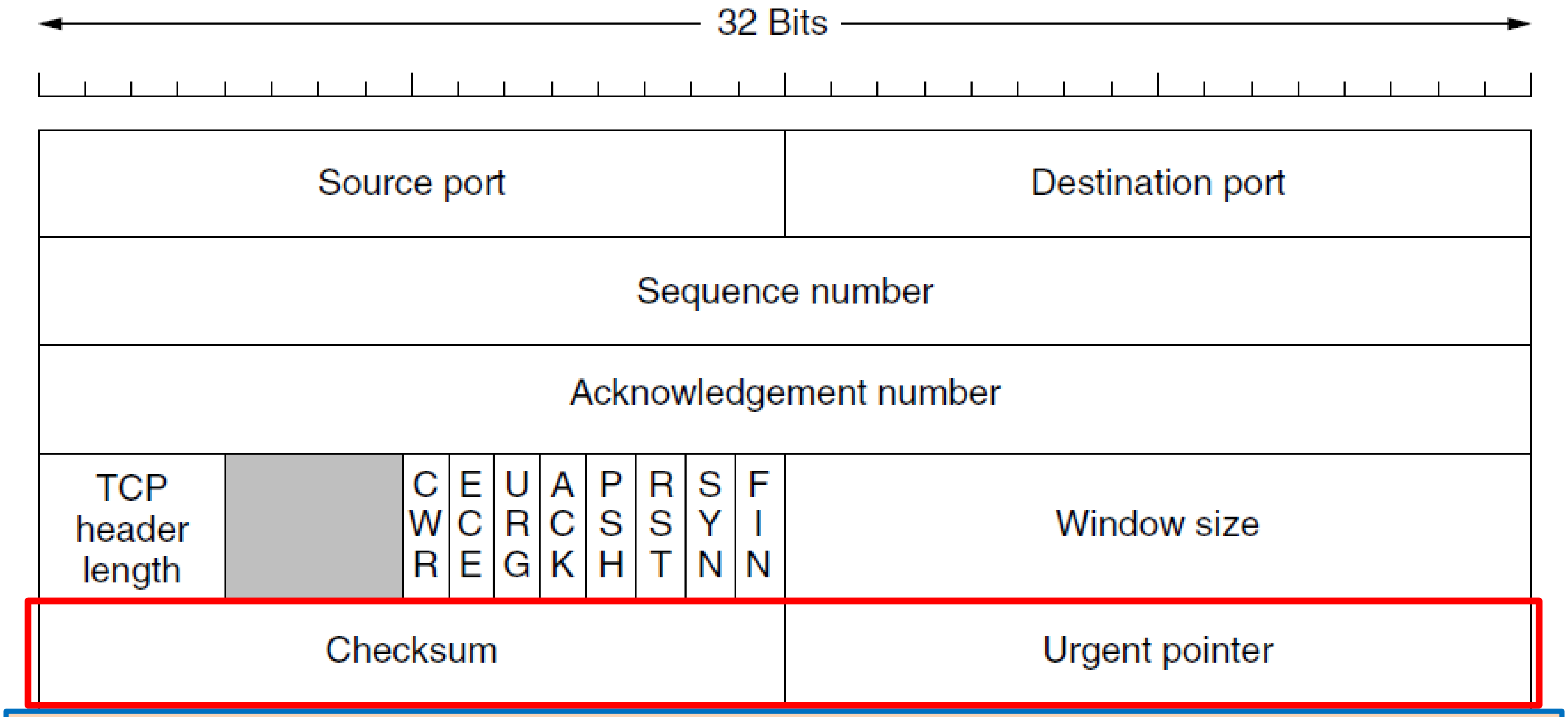
RST bit: to reset a connection that has become confused due to a host crash or some other reason.
SYN bit: to used to establish connections. SYN for CONNECTION REQUEST,  SYN+ACK for CONNECTION ACCEPTED.
FIN bit: used to release a connection.

# TCP: The Header



**Window size**: to tell how many bytes may be sent starting at the byte acknowledged.

# TCP: The Header



32 Bits

Source port

Destination port

Sequence number

Acknowledgement number

TCP header length

CWR

ECE

URG

ACK

PSH

RST

SYN

FIN

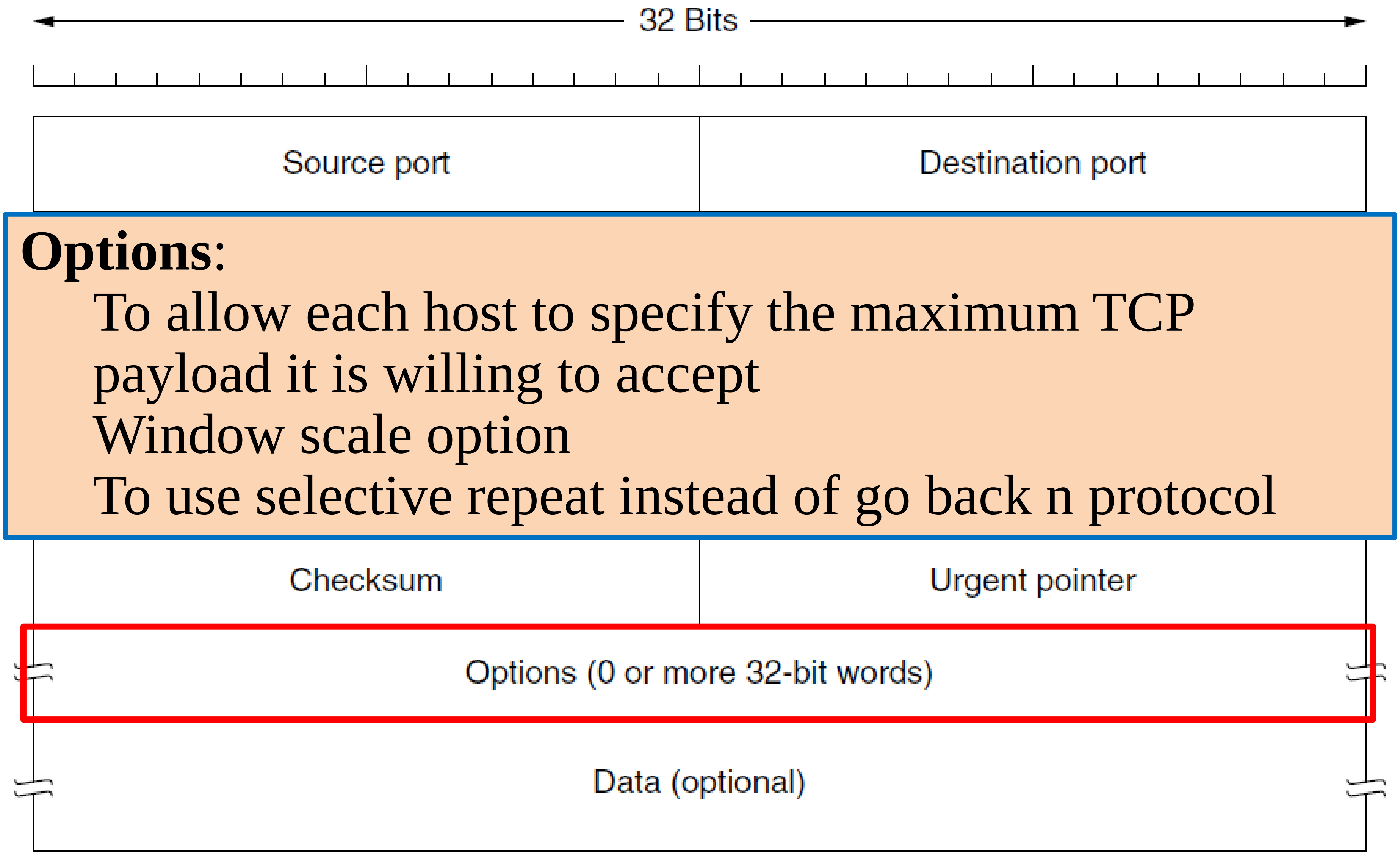
Window size

Checksum

Urgent pointer

**Checksum**: provided for extra reliability.
**Urgent pointer**: used to indicate a byte offset from the current sequence number at which urgent data are to be found.

# TCP: The Header



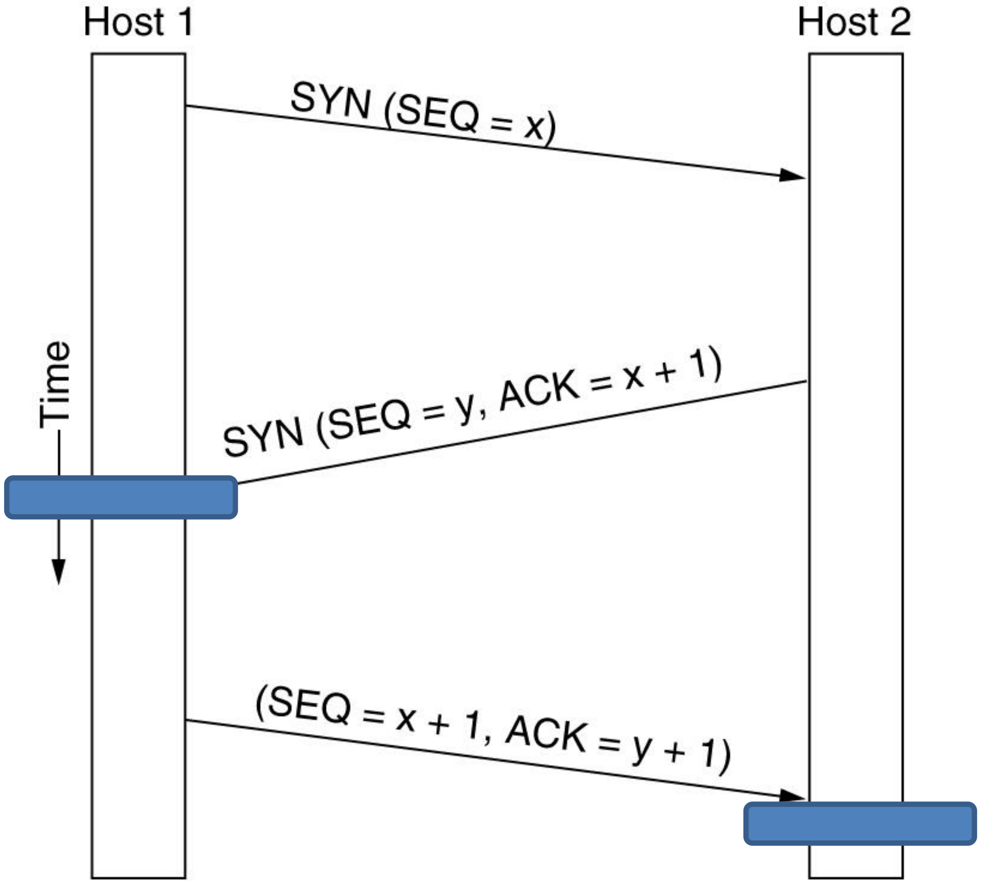
**Options**:

To allow each host to specify the maximum TCP payload it is willing to accept

Window scale option

To use selective repeat instead of go back n protocol

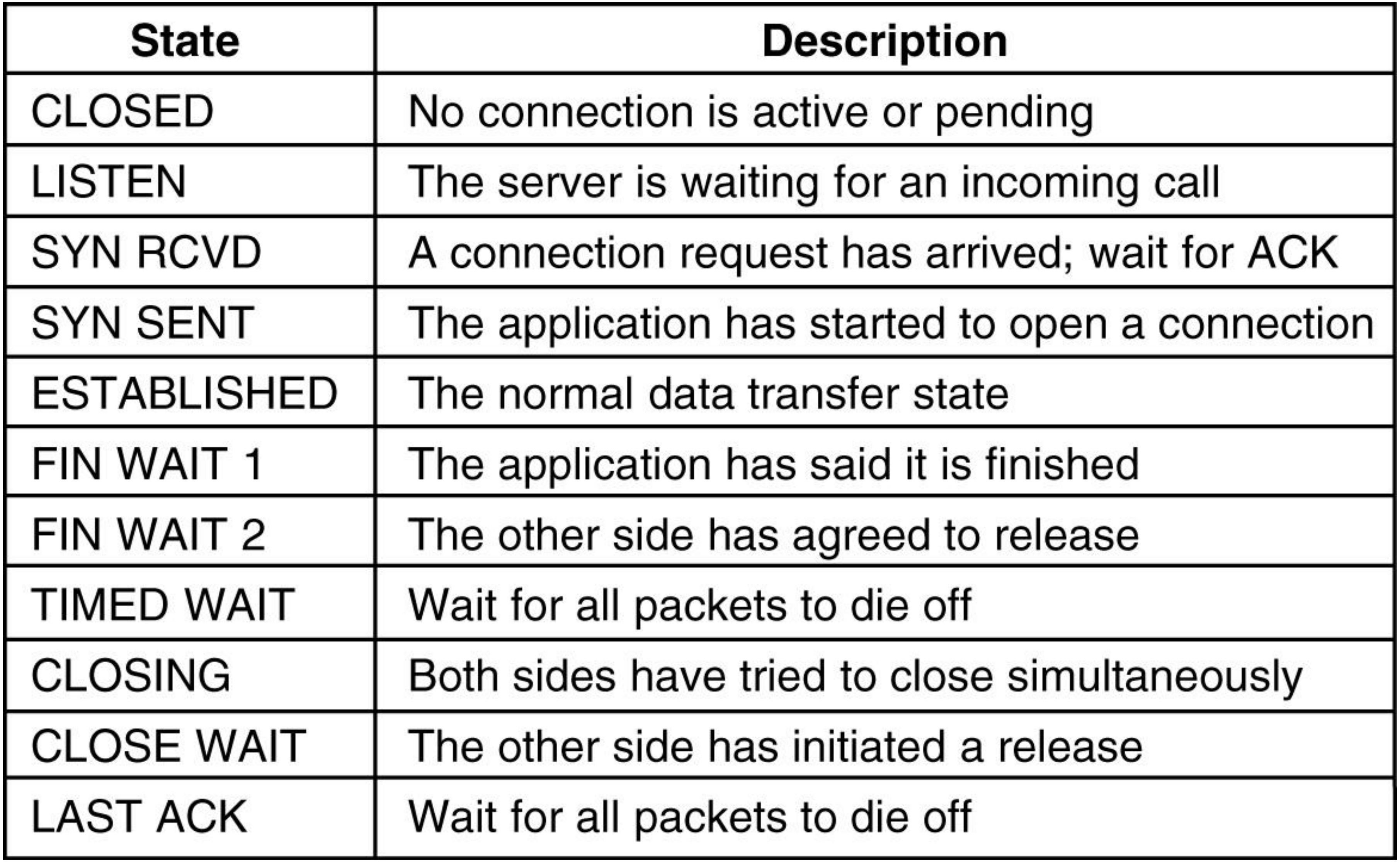
# TCP: Connection Establishment



# TCP: Connection Release

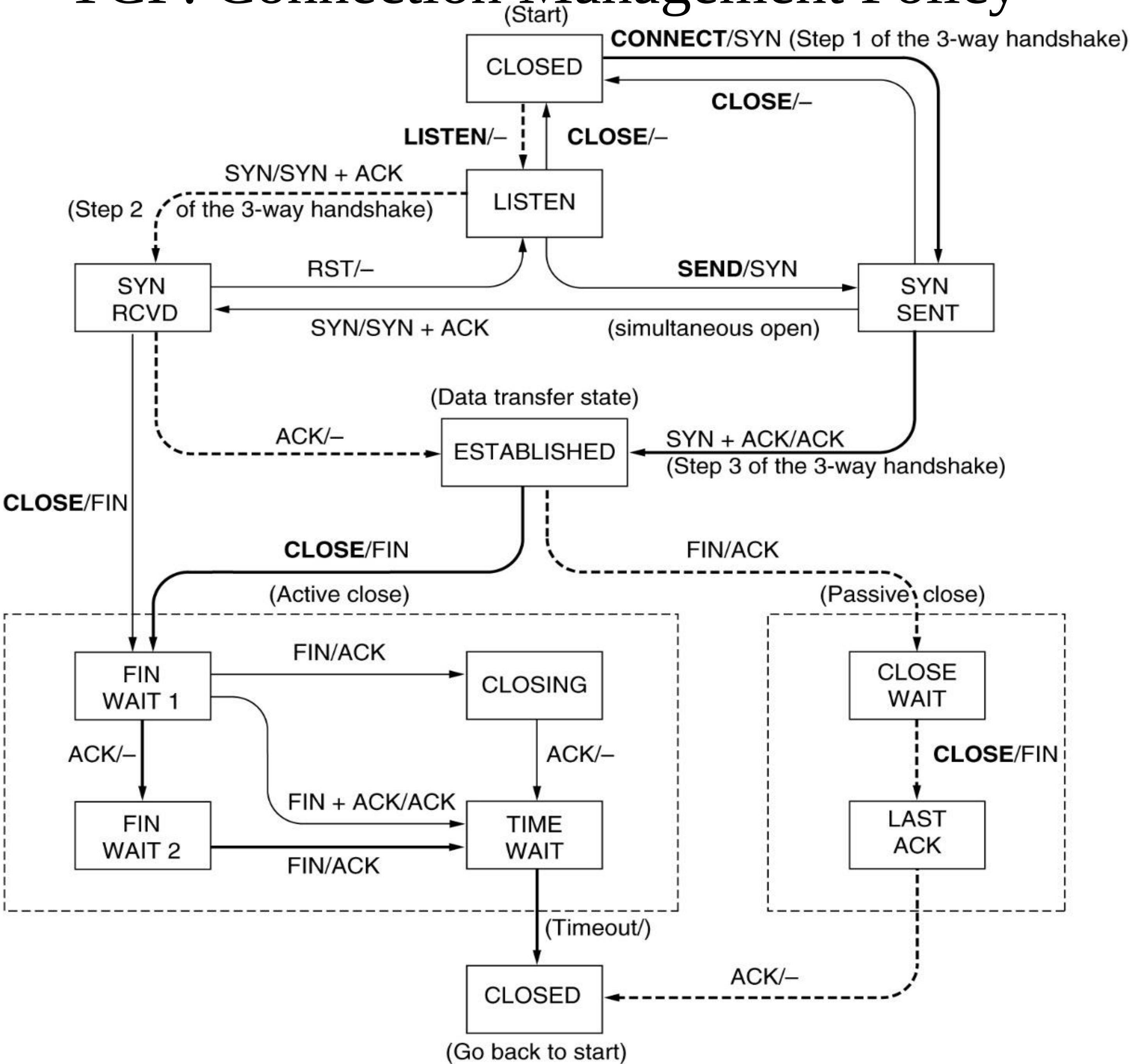
- TCP connections are full duplex and can be treated as a pair of simplex connections.
- Each simplex connection is released independently of its sibling.
  - To release a connection, a party can send a TCP segment with the FIN bit set, which means that it has no more data to transmit.
  - When the FIN is acknowledged, that direction is shut down for the new data.
  - When both directions have been shut down, the connection is released.
- To avoid the two-army problem, timers are used.

# TCP: Connection Management Policy

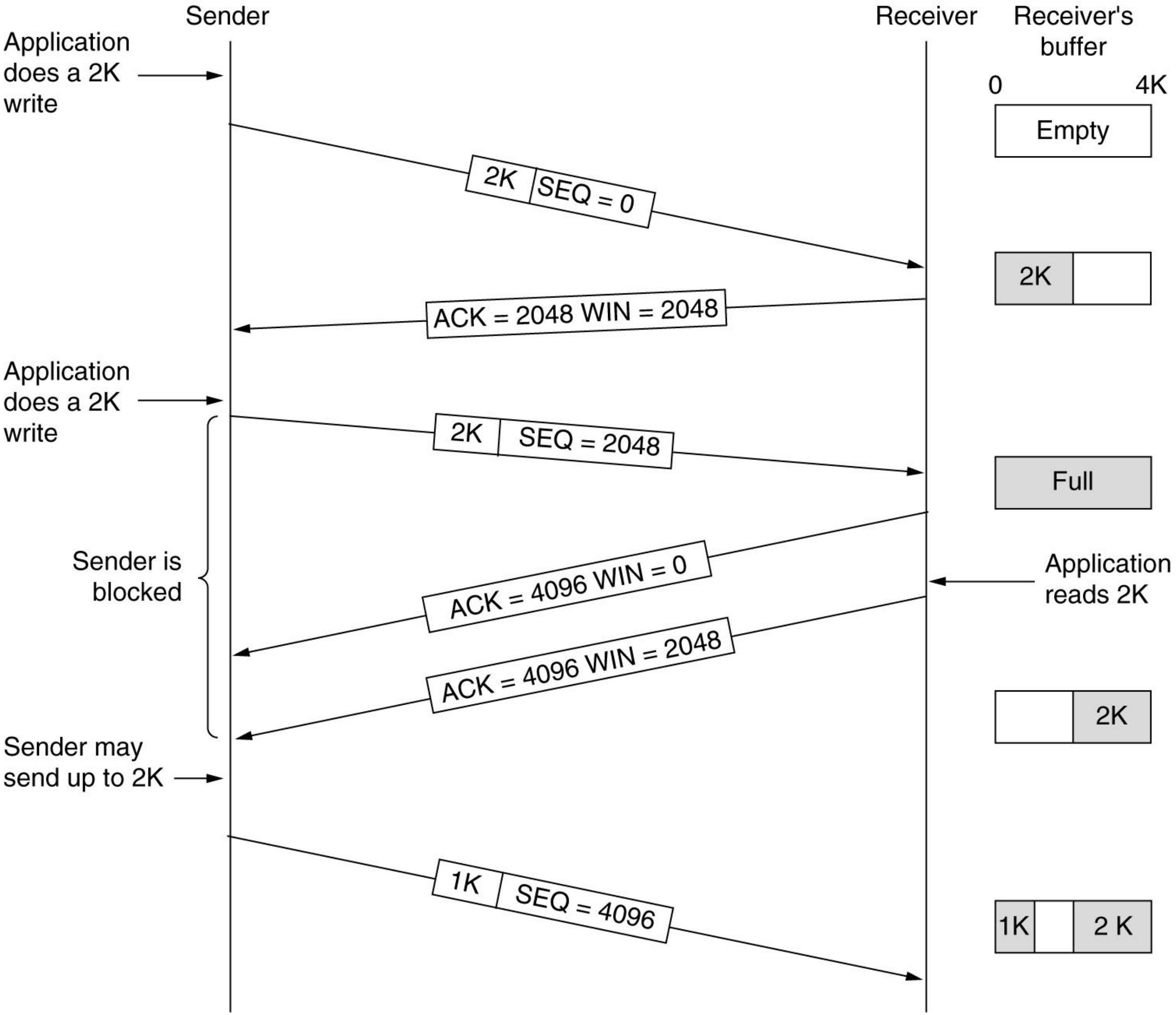


| State | Description |
|---|---|
| CLOSED | No connection is active or pending |
| LISTEN | The server is waiting for an incoming call |
| SYN RCVD | A connection request has arrived; wait for ACK |
| SYN SENT | The application has started to open a connection |
| ESTABLISHED | The normal data transfer state |
| FIN WAIT 1 | The application has said it is finished |
| FIN WAIT 2 | The other side has agreed to release |
| TIMED WAIT | Wait for all packets to die off |
| CLOSING | Both sides have tried to close simultaneously |
| CLOSE WAIT | The other side has initiated a release |
| LAST ACK | Wait for all packets to die off |

# TCP: Connection Management Policy



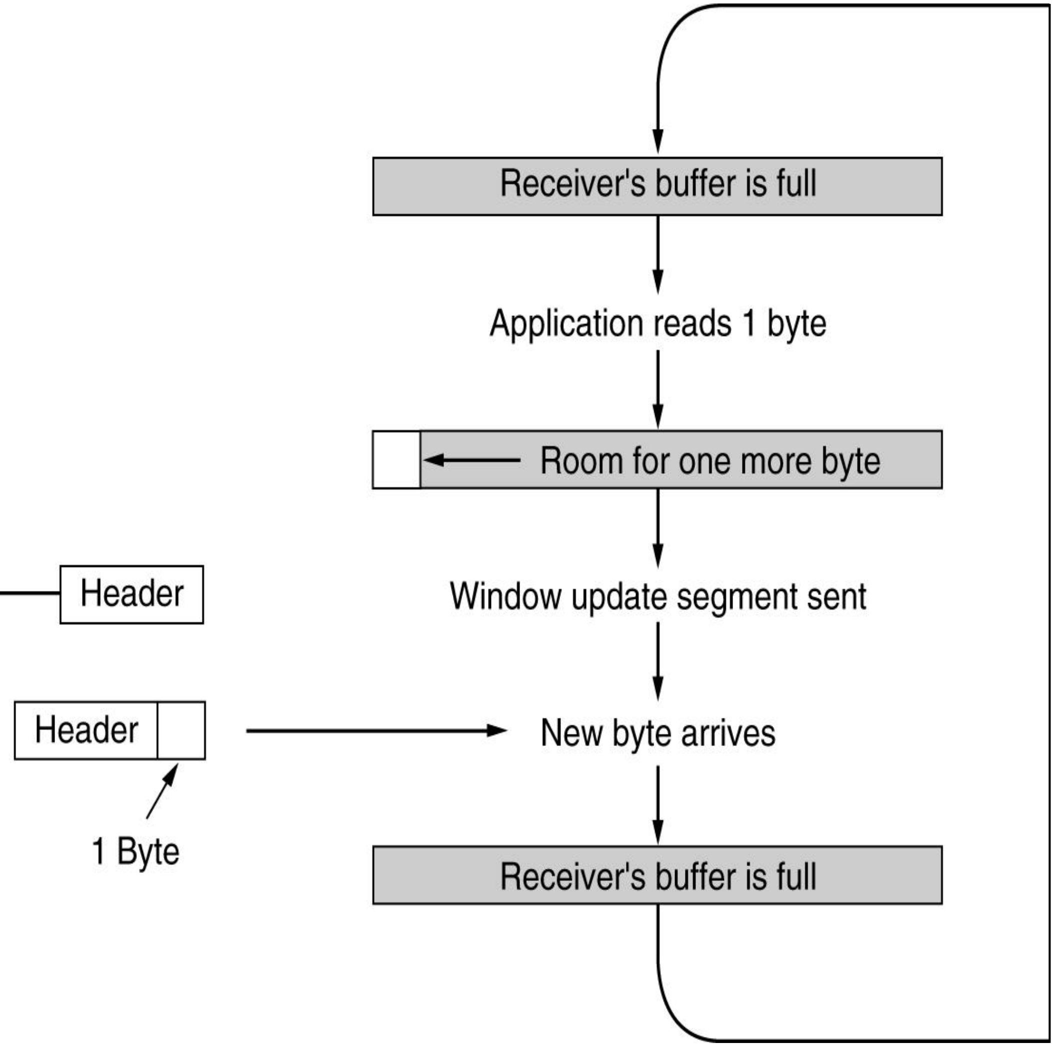
# TCP: Sliding Window



# TCP: Sliding Window

- The *low utilization problem* in TCP
  - Consider a telnet connection to an interactive editor that reacts on every keystroke.
  - One useful in 20+20+1 byte packet.
- Sender side approach: **Nagle's algorithm**
  - When data come into the sender one byte at a time, just send the first byte and buffer all the rest until the outstanding byte is acknowledged. Then send all the buffered characters in one TCP segment and start buffering again until they are acknowledged. (To disable it by TCP_NODELAY option)

# TCP: Transmission Policy

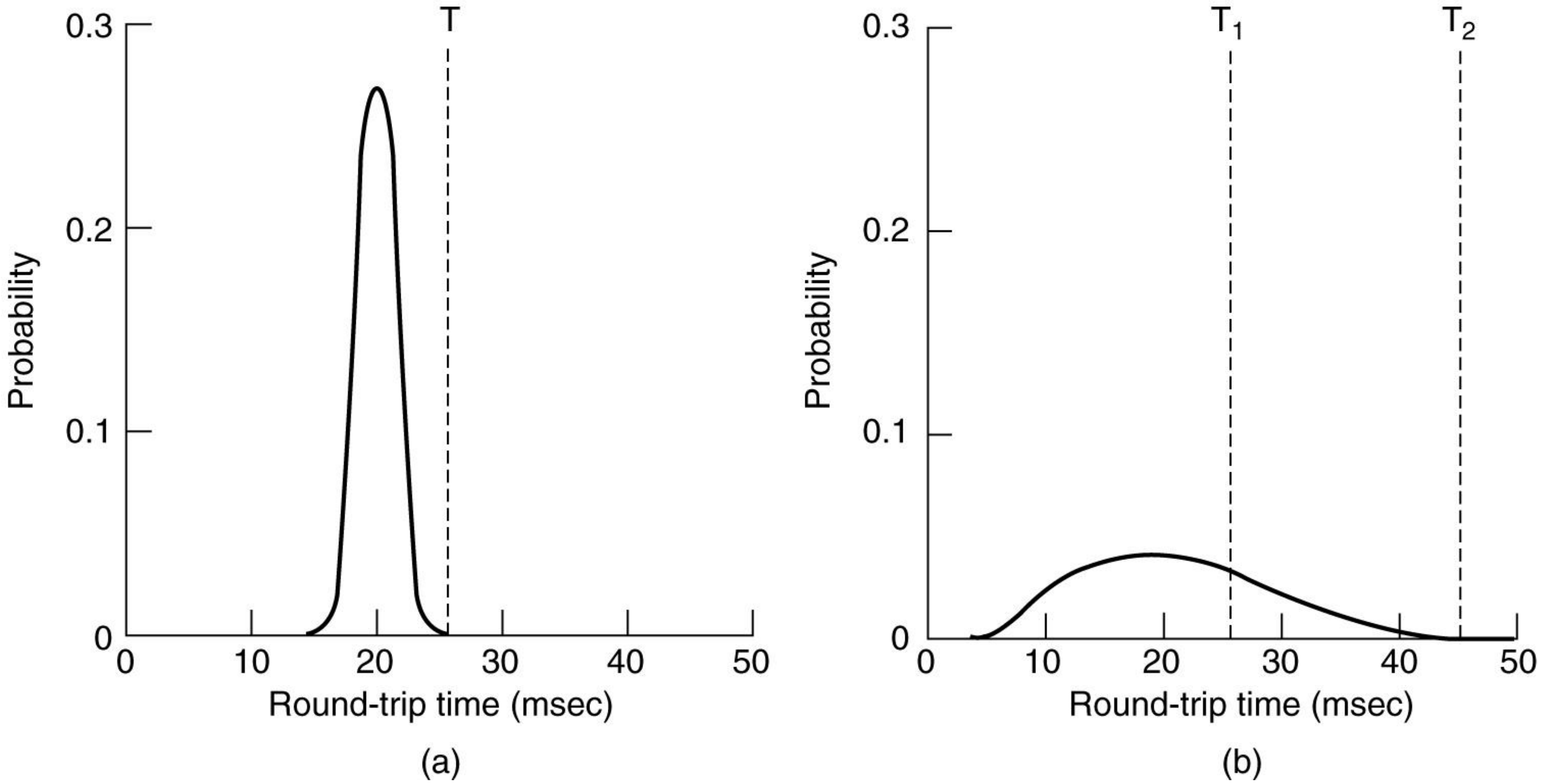


- The *low utilization problem* in TCP can also be caused by the receiver side
- Receiver side approach: **Clark's solution** for silly window syndrome
  - To prevent the receiver from sending a window update for 1 byte.

# TCP: Timer Management

(a) Probability density of ACK arrival times in the data link layer.

(b) Probability density of  ACK arrival times for TCP.



# TCP: Timer Management

- **Retransmission timer**
  - SRTT (Smoothed Round-Trip Time) ($\alpha$=7/8)

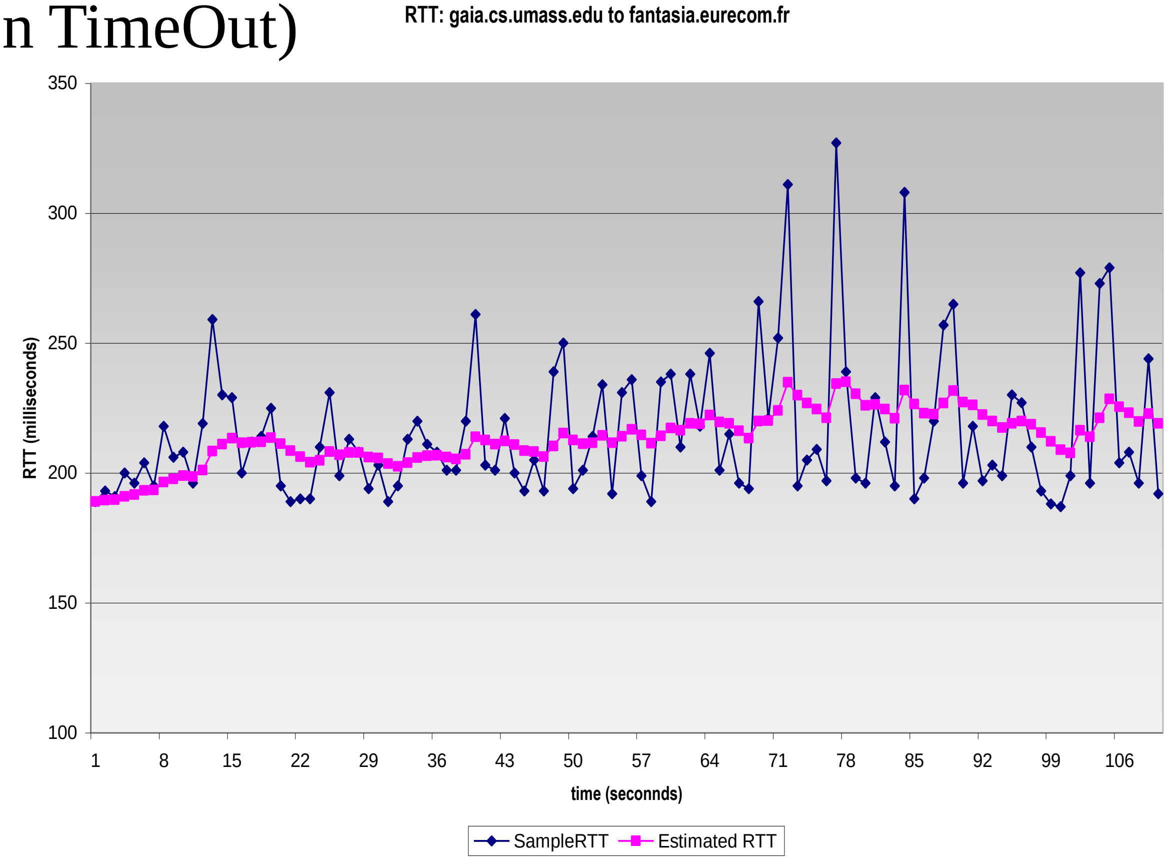
$$\mathbf{SRTT = \alpha\, SRTT + (1 - \alpha)\, R}$$

  - RTTVAR(Round-Trip Time VARiation) ($\beta$=3/4)

$$\mathbf{RTTVAR = \beta\, RTTVAR + (1 - \beta)\, |\, SRTT - R\, |}$$

  - RTO (Retransmission TimeOut)

$$\mathbf{RTO = SRTT + 4\, RTTVAR}$$



# TCP timers

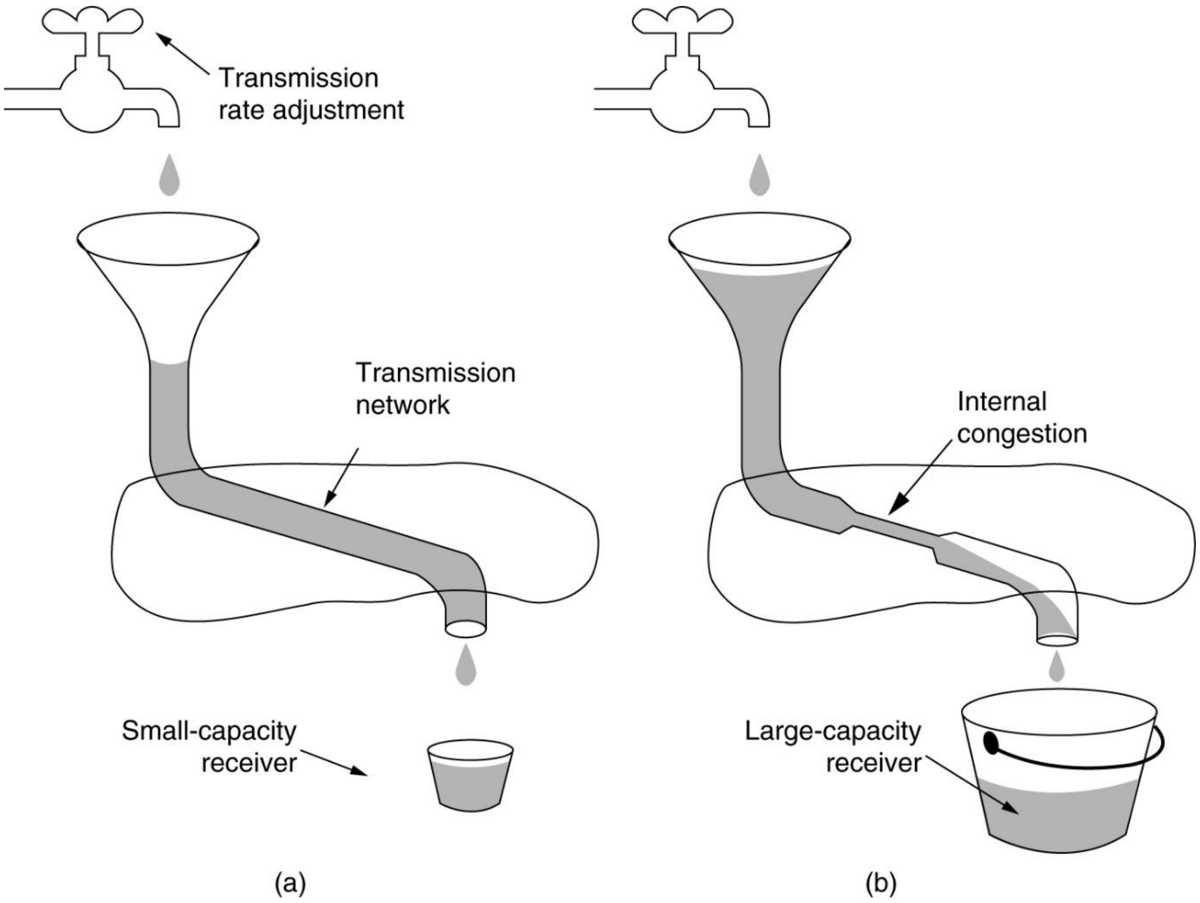
- **Retransmission timer:** described above
- **Persistence timer**
  send probe to prevent the deadlock:

| | | | | |
|---|---|---|---|---|
| 15 | ← | <ack = 6, buf = 0> | ← | A is still blocked |
| 16 | • • • | <ack = 6, buf = 4> | ← | Potential deadlock |

- **Keepalive timer**
  to check whether the other side is still there.
- Other timers such as the one used in the TIMED WAIT state.

# TCP: Congestion Control

- Congestion window
- Receiver window

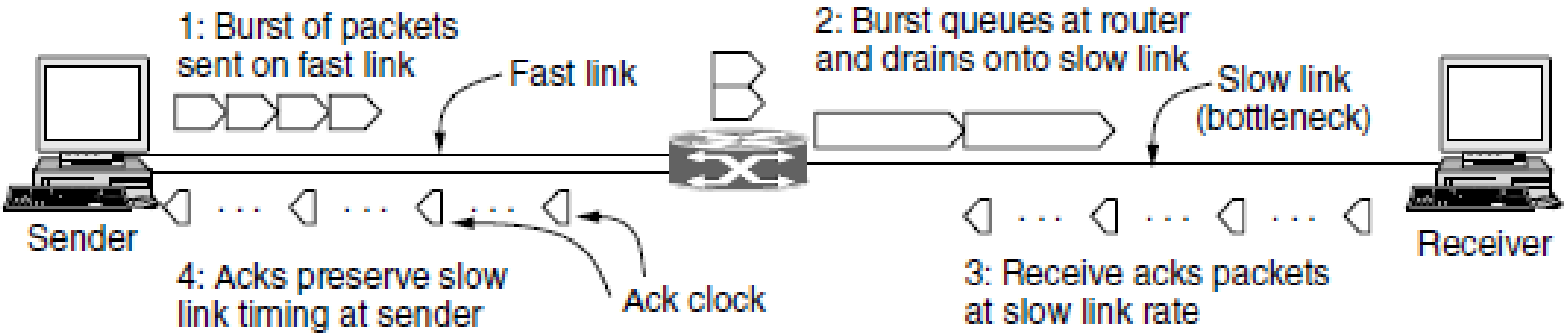


# TCP: Congestion Control

- Congestion Control Algorithm by Van Jacobson (1988)
  - To approximate an **AIMD** congestion window
  - To represent congestion signal by packet loss
  - To measure packet loss by a retransmission timer
  - To split data into segments (Ack Clock)
  - To use the optimal congestion window

# TCP: Congestion Control

## Ack clock

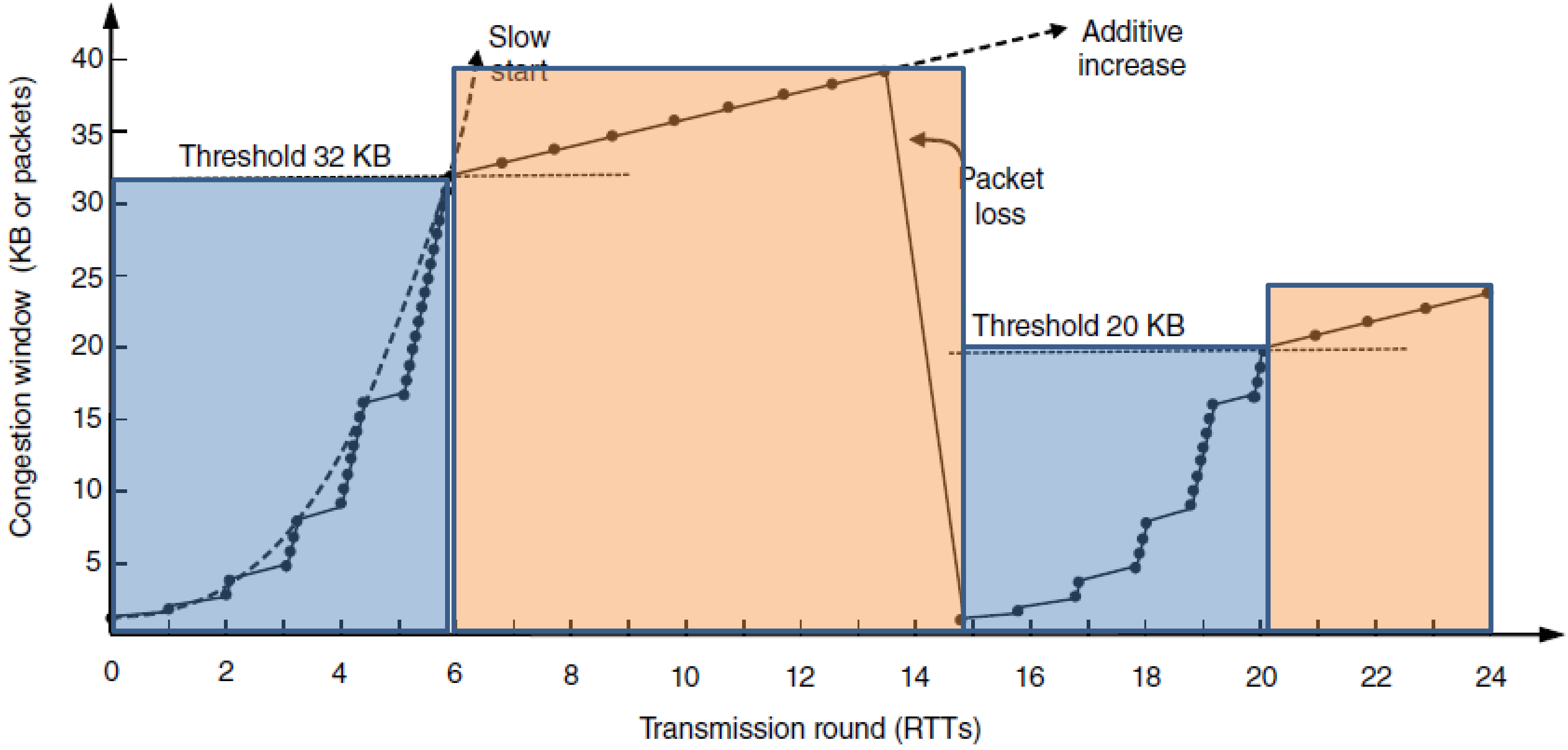

1: Burst of packets sent on fast link
Fast link
2: Burst queues at router and drains onto slow link
Slow link (bottleneck)
Sender
Receiver
4: Acks preserve slow link timing at sender
Ack clock
3: Receive acks packets at slow link rate

A burst of packets from a sender and the returning ack clock.

# TCP: Congestion Control

Slow start followed by additive increase in **TCP Tahoe**.

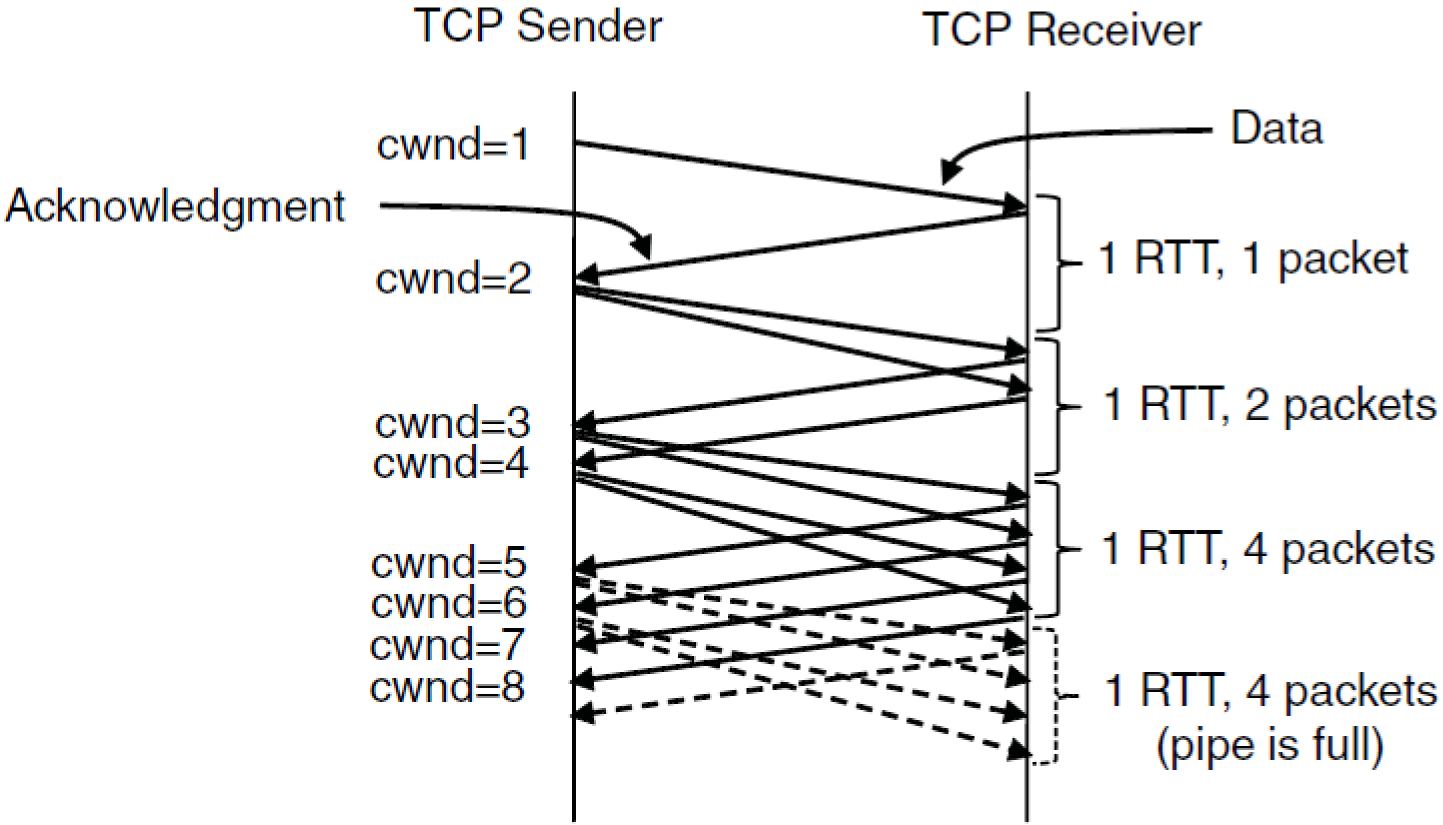
- **`cwnd`**
- **`Threshold`**
- slow-start (SS)
- congestion-avoidance



# TCP: Congestion Control

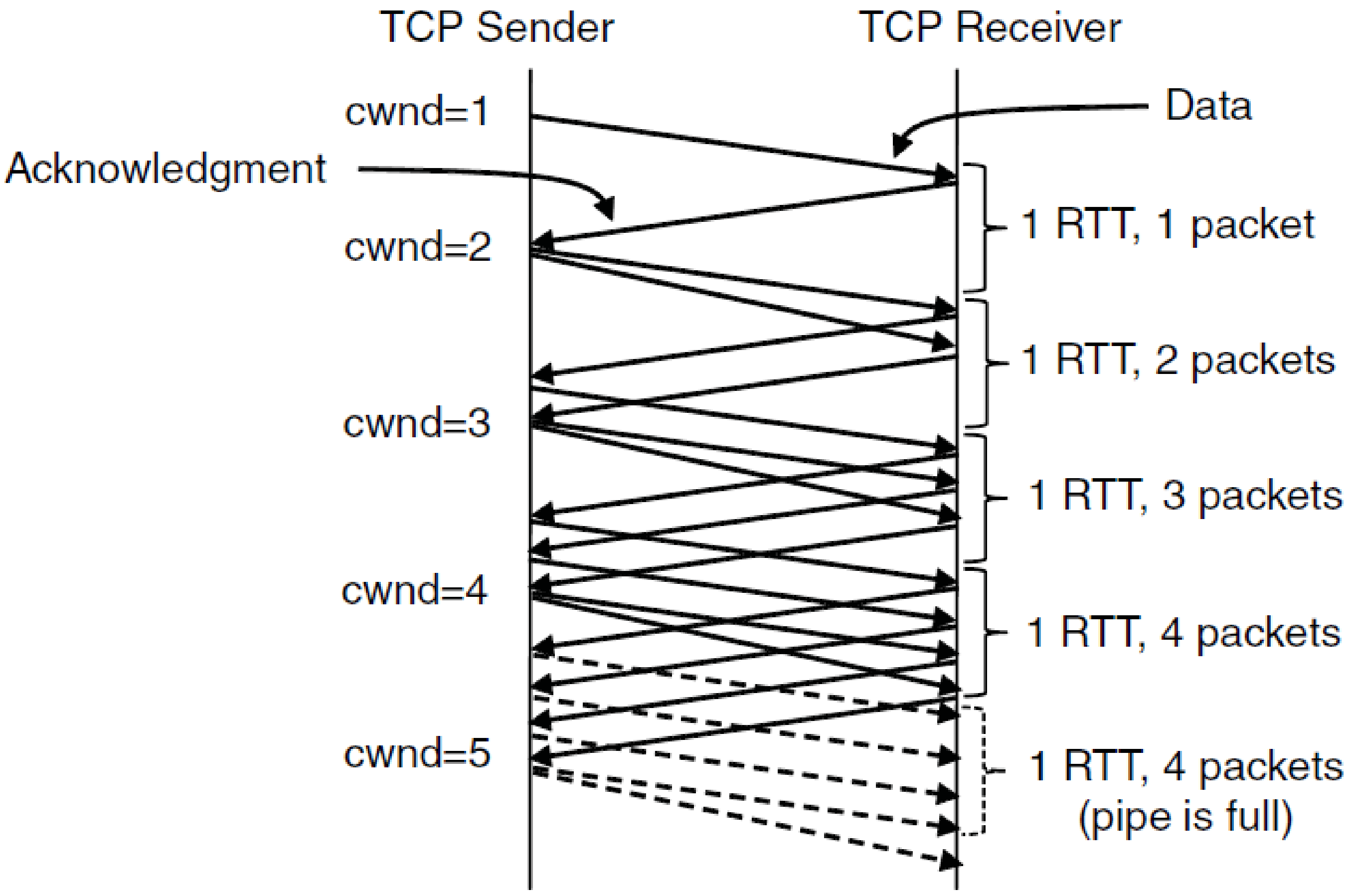
The **AIMD** rule will take a very long time to reach a good operating point on fast networks if the congestion window is started from a small size. => **slow start is not slow**

Slow start from an initial congestion window of 1 segment



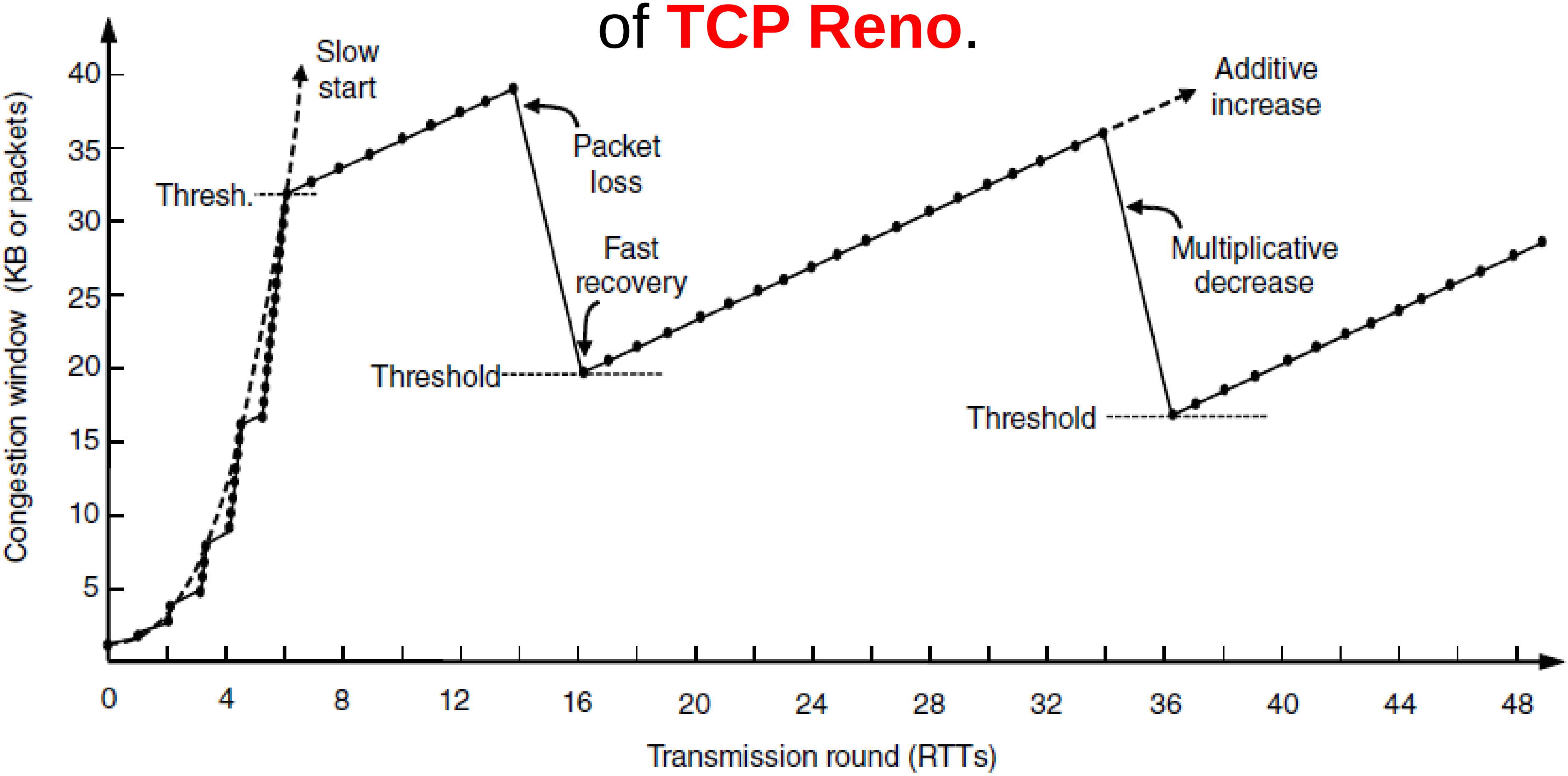
# TCP: Congestion Control

**CA:** Additive increase from an initial congestion window of 1 segment.



# TCP: Congestion Control

**Fast recovery** and the sawtooth pattern of **TCP Reno**.



# TCP/Reno: Why different actions on detected loss?

- After 3 dup ACKs:
  - **`cwnd`** is cut in half
  - window then grows linearly
- But after timeout event:
  - **`cwnd`** instead set to 1 MSS;
  - window then grows exponentially to a **threshold**, then grows linearly
  - Threshold =1/2 of **`cwnd`** before loss event

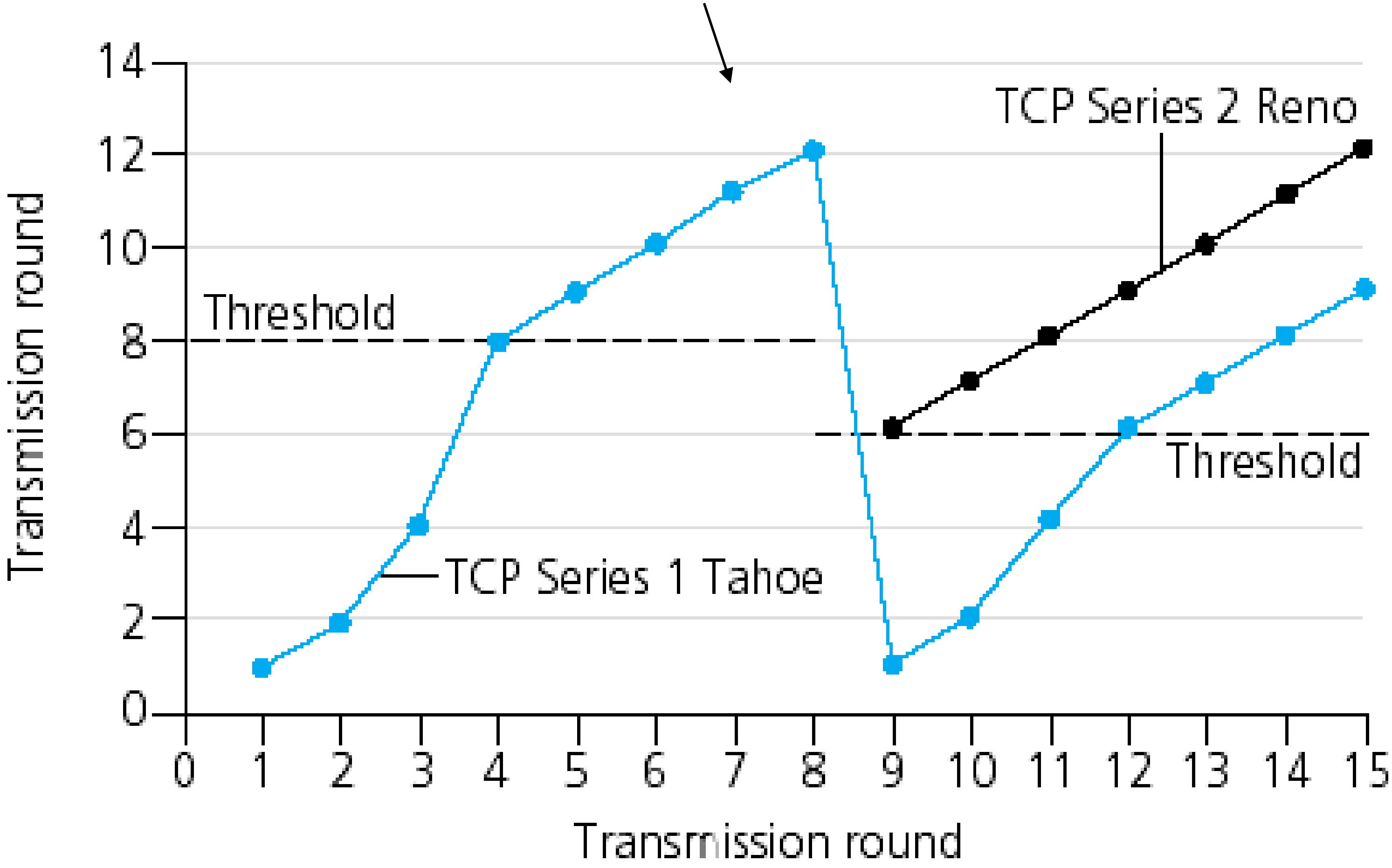
Philosophy:

❑ 3 dup ACKs indicates network capable of delivering some segments

❑ timeout indicates a "more alarming" congestion scenario

# Example

Triple duplicated ACK



# Summary: TCP (Reno) Congestion Control

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

# How to change sending rate?

- **cwnd** is measured in bytes (or number of pkts)
- Send rate? Implicitly indicated by **cwnd**:

  Rate = **cwnd**/RTT
- # ACKs per RTT = cwnd/MSS
- Linear increase: +MSS per RTT

  → +MSS*(MSS/cwnd) per ACK
- Exponential increase: double Rate

  → +cwnd per RTT

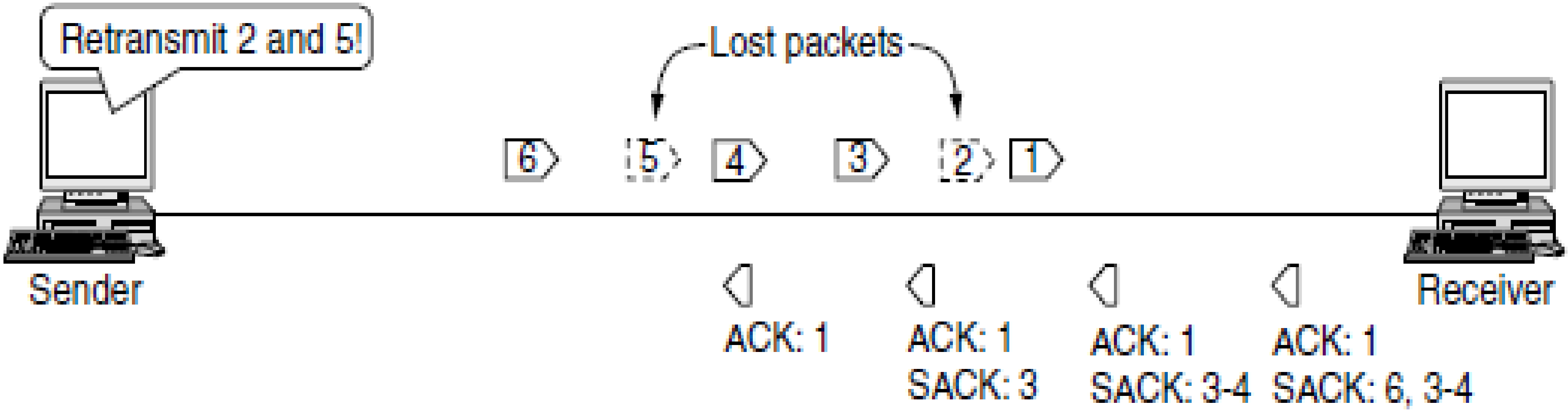
  → +MSS per ACK

# TCP sender congestion control

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

# TCP: Congestion Control

*Selective* ackowledgement



# TCP throughput

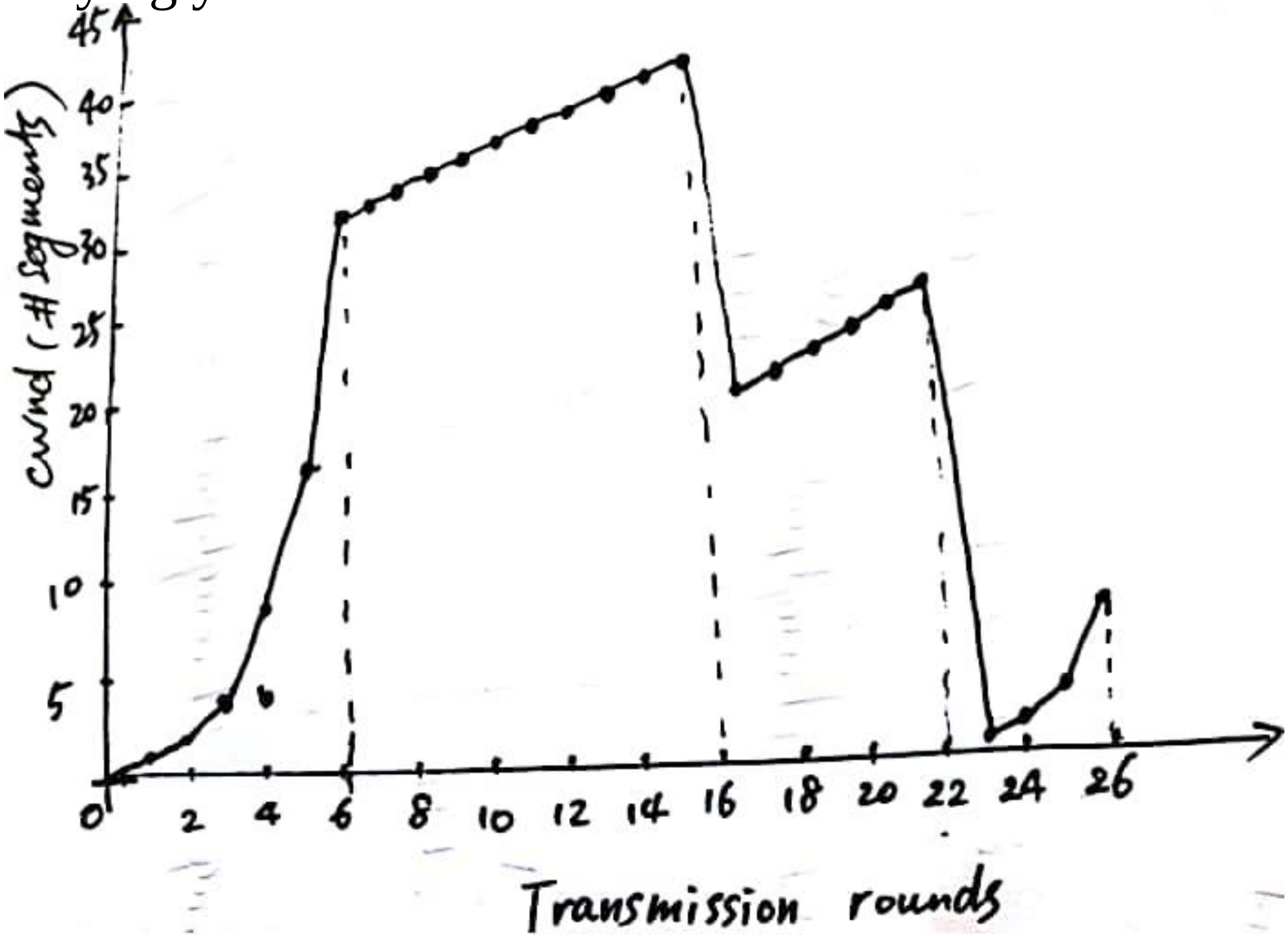
- What’s the average throughput of TCP as a function of window size and RTT?
  - Ignore slow start
- Let W be the window size when loss occurs.
- When window is W, throughput is W/RTT
- Just after loss, window drops to W/2, throughput to W/2RTT.
- Average throughout: **0.75 W/RTT**

# TCP: The Future

- SCTP (Stream Control Transmission Protocol ) (RFC 4960) is message-oriented like UDP and ensures reliable, in-sequence transport of messages.
- SST (Structured Stream Transport) is an experimental transport protocol designed to address the needs of modern applications that need to juggle many asynchronous communication activities in parallel, such as downloading different parts of a web page simultaneously and playing multiple audio and video streams at once.
- FAST TCP (Wei et al. 2006)
- Transactional TCP
- …

# Question

- Consider the following figure. Assuming **TCP Reno** is the protocol experiencing the behavior, answer the following questions. In all cases, you should provide a short discussion justifying your answer.



- **(1).** Identify the intervals of time when TCP slow start is operating.
- **(2).** Identify the intervals of time when TCP congestion avoidance is operating.
- **(3).** After the 16th transmission round, is segment loss detected by a triple duplicate ACK or by a timeout?
- **(4).** After the 22nd transmission round, is segment loss detected by a triple duplicate ACK or by a timeout?
- **(5).** What is the initial value of **Threshold** at the first transmission round?
- **(6).** What is the value of **Threshold** at the 18th transmission round?
- **(7).** What is the value of **Threshold** at the 24th transmission round?
- **(8).** Assuming a packet loss is detected after the 26th round by the receipt of a triple duplicate ACK, what will be the values of the **cwnd** and **Threshold**?
- **(9).** Suppose **TCP Tahoe** is used (instead of TCP Reno), and assume that triple duplicate ACKs are received at the 16th round. What are the **Threshold** and **cwnd** at the 19th round?

# Homework(1)

- **1.** In both parts of Fig. 6-6, there is a comment that the value of SERVERPORT must be the same in both client and server. Why is this so important?

- **2.** Imagine that a two-way handshake rather than a three-way handshake were used to set up connections. In other words, the third message was not required. Are deadlocks now possible? Give an example or show that none exist.

- **3.** Why does UDP exist? Would it not have been enough to just let user processes send raw IP packets?

# Homework(2)

- **4.** A client sends a 128-byte request to a server located 100 km away over a 1-gigabit optical fiber. What is the efficiency of the line during the remote procedure call?

- **5.** Datagram fragmentation and reassembly are handled by IP and are invisible to TCP. Does this mean that TCP does not have to worry about data arriving in the wrong order?

- **6.** The maximum payload of a TCP segment is 65,495 bytes. Why was such a strange number chosen?

# Homework(3)

- **7.** If the TCP round-trip time, RTT, is currently 30 msec and the following acknowledgements come in after 26, 32, and 24 msec, respectively, what is the new RTT estimate using the Jacobson algorithm? Use $\alpha=0.9$.

- **8.** To get around the problem of sequence numbers wrapping around while old packets still exist, one could use 64-bit sequence numbers. However, theoretically, an optical fiber can run at 75 Tbps. What maximum packet lifetime is required to make sure that future 75-Tbps networks do not have wrap around problems even with 64-bit sequence numbers? Assume that each byte has its own sequence number, as TCP does.

- **9.** Consider that only a single TCP (Reno) connection uses one 10Mbps link which does not buffer any data. Suppose that this link is the only congested link between the sending and receiving hosts. Assume that the TCP sender has a huge file to send to the receiver, and the receiver's receive buffer is much larger than the congestion window. We also make the following assumptions: each TCP segment size is 1,500 bytes; the two-way propagation delay of this connection is 150 msec; and this TCP connection is always in congestion avoidance phase, that is, ignore slow start.
  - **a.** What is the maximum window size (in segments) that this TCP connection can achieve?
  - **b.** What is the average window size (in segments) and average throughput (in bps) of this TCP connection?
  - **c.** How long would it take for this TCP connection to reach its maximum window again after recovering from a packet loss?