Networks: TCP and UDP

Transport Layer

Dr Mirco Musolesi

Dr Ian Batten

UNIVERSITY OF BIRMINGHAM

传输服务及端口

Transport services and protocols

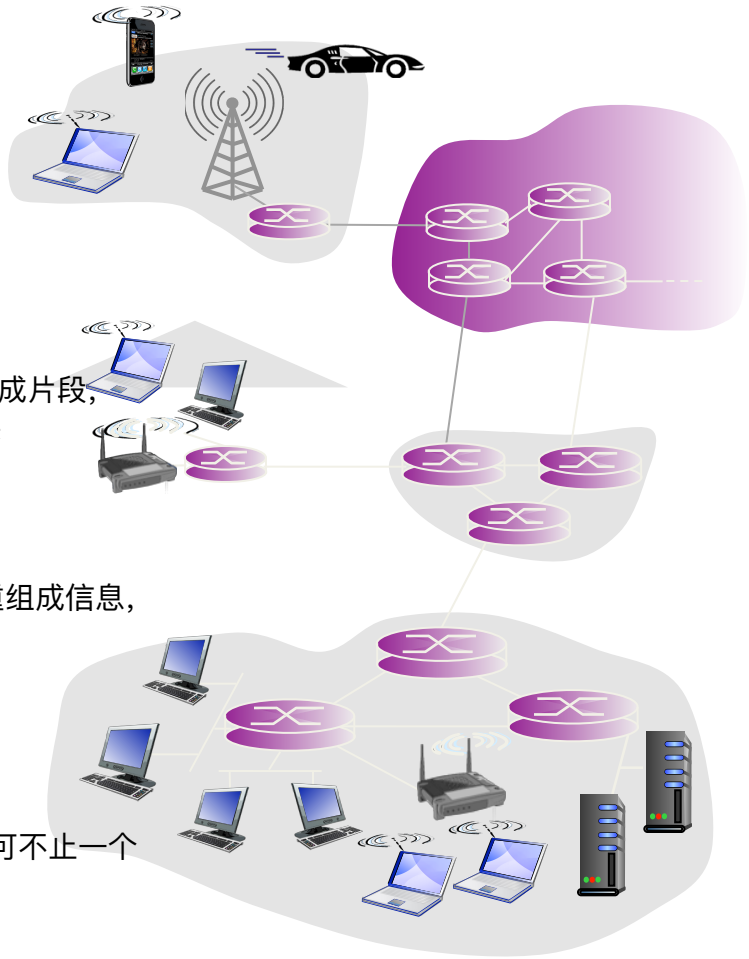
在不同主机上运行的应用程序进程之间提供逻辑通信

- provide *logical communication* between app processes running on different hosts
- transport protocols run in end systems 在终端系统中运行的传输协议：
  - send side: breaks app messages into *segments*, passes to network layer
  - rcv side: reassembles segments into messages, passes to app layer
- more than one transport protocol available to apps
  - Internet: TCP and UDP

1.发送方将信息分成片段，以此来通过网络层

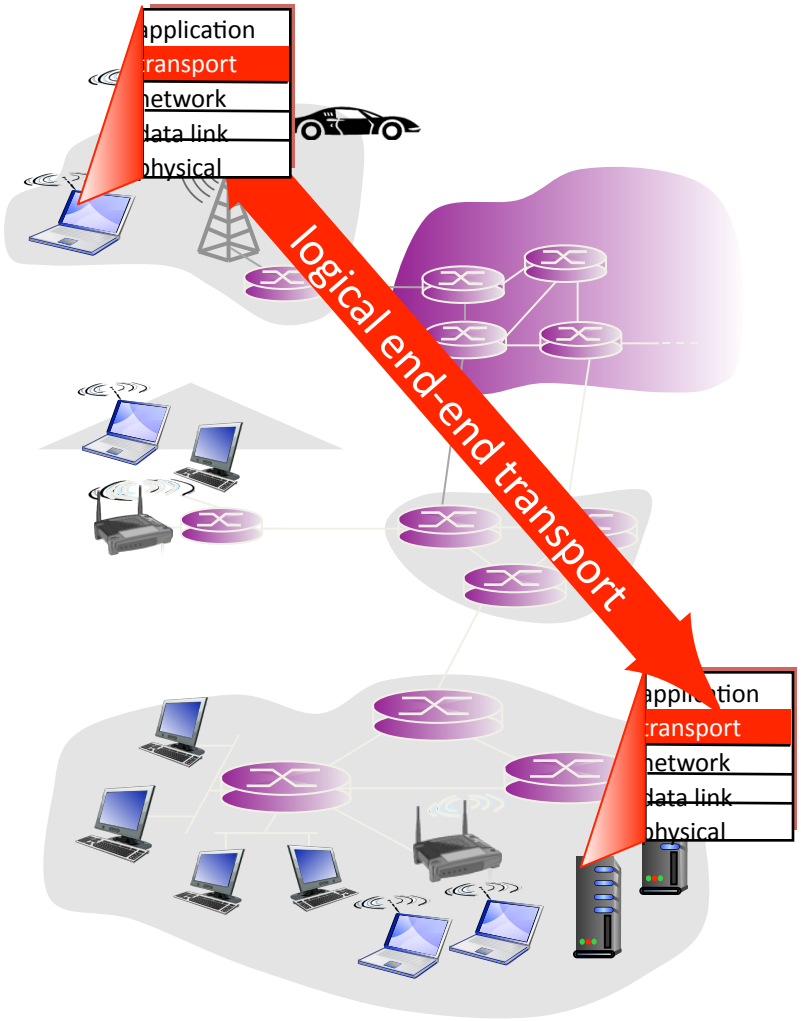
2.接收方把片段重组成信息，以此来通过app层

app的传输端口可不止一个



Transport services and protocols

- provide *logical communication* between app processes running on different hosts
- transport protocols run in end systems
  - send side: breaks app messages into *segments*, passes to network layer
  - rcv side: reassembles segments into messages, passes to app layer
- more than one transport protocol available to apps
  - Internet: TCP and UDP



Transport vs. network layer

网络层:主机之间的逻辑通信

- ***network layer***: logical communication between **hosts**

传输层:进程之间的逻辑通信 -用以依赖、增强网络层服务

- ***transport layer***: logical communication between **processes**
  - relies on, enhances, network layer services

网络传输层协议

Internet transport-layer protocols

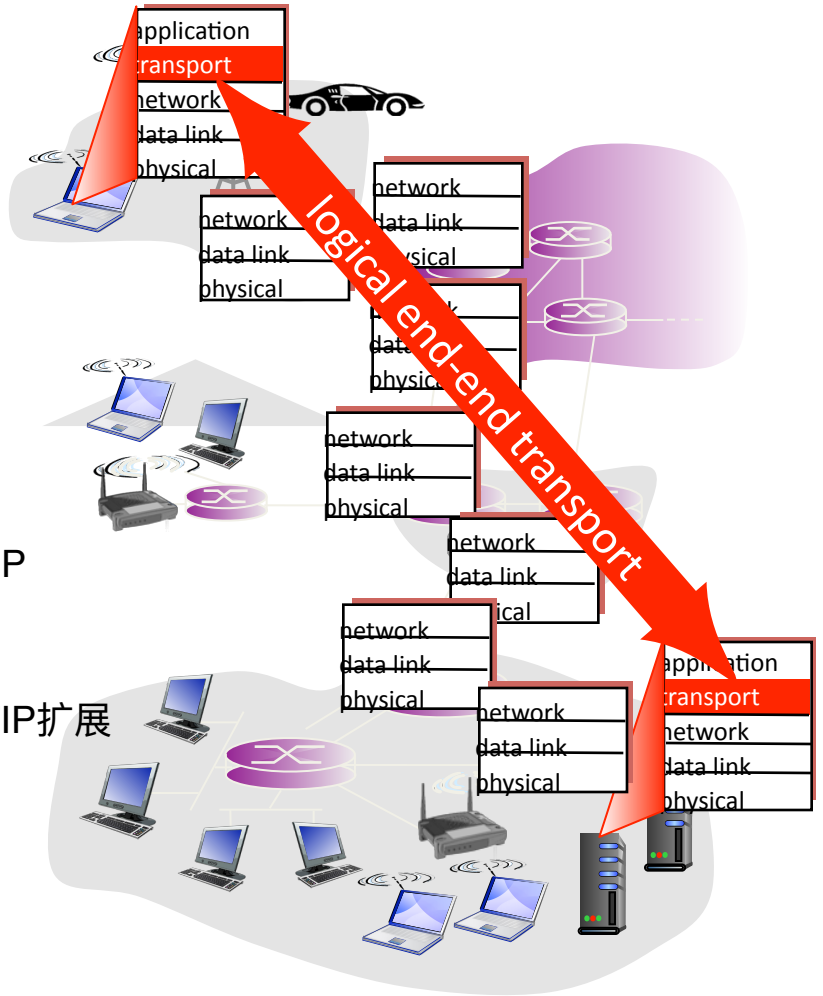
可靠的顺序交付(TCP)

- reliable, in-order delivery (TCP)
  - congestion control
  - flow control
  - connection setup

拥塞控制
流量控制
连接设定

- unreliable, unordered delivery: UDP 不可靠的,无序的交付:UDP
  - no-frills extension of "best-effort" IP 朴实无华的"最佳努力"IP扩展
- services not available:
  - delay guarantees
  - bandwidth guarantees

不可用的服务：延迟保证，带宽保证



多路复用/多路分解

套接字：IP地址+TCP/UDP+ Port

Multiplexing/demultiplexing

接收主机的多路复用：将接收到的文段传送到正确的套接字

Demultiplexing at rcv host:

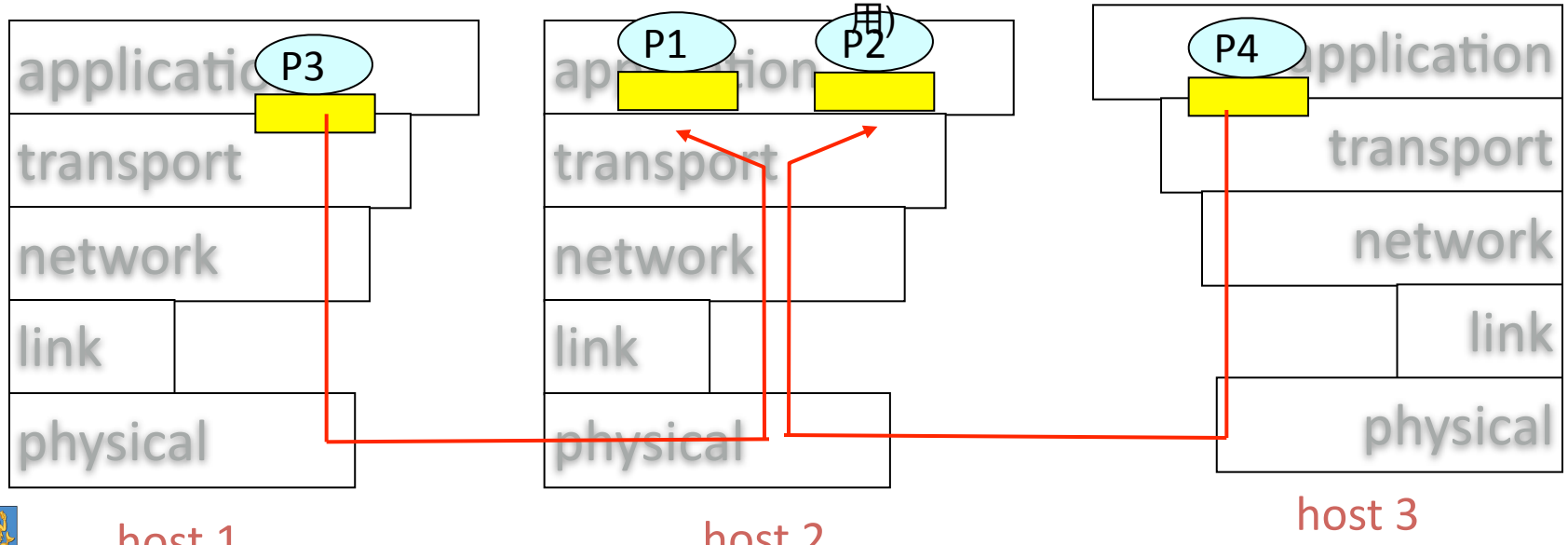
delivering received segments to correct socket

Multiplexing at send host:

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

= socket = process

发送主机的多路复用：从多个套接字收集数据，再用标头来封装数据(稍后用于解复用)



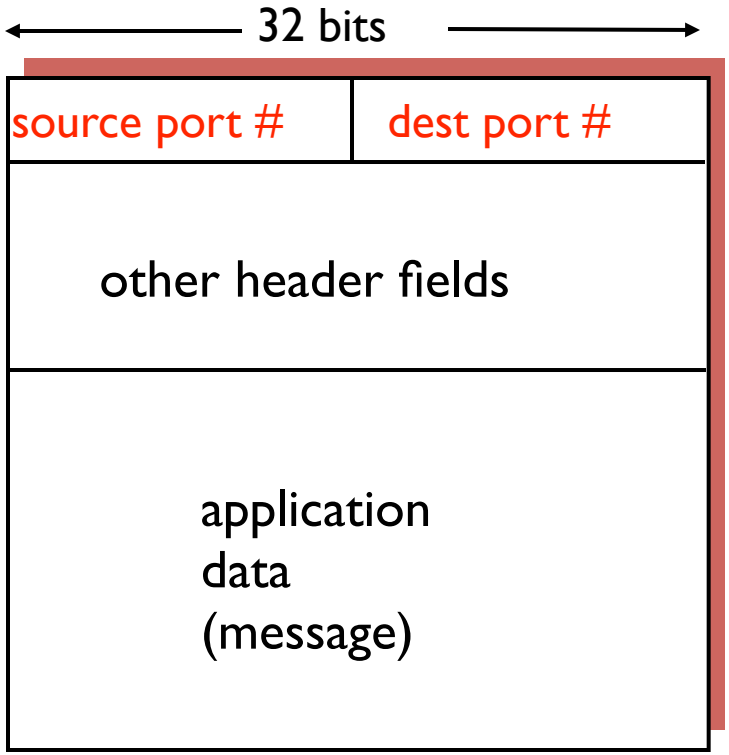
多路分解是咋个工作的

How demultiplexing works

主机接收IP数据报：每个数据报都有源IP地址、目标IP地址.
每个数据报携带一个传输层段. 每个段都有源、目的端口号

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

主机使用IP地址和端口号引导段到适当的套接字



TCP/UDP segment format

面向连接的多路分配器

Connection-oriented demux

TCP套接字由4元组识别: 源IP地址，源端口号，目标IP地址，目标端口号

- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- demux: receiver uses all four values to direct segment to appropriate socket

多路分配：接收者使用这四个值来把文段定位到正确的套接字

服务器可以同时支持多个TCP套接字

- server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple

每个套接字由四个自己的四元组验证

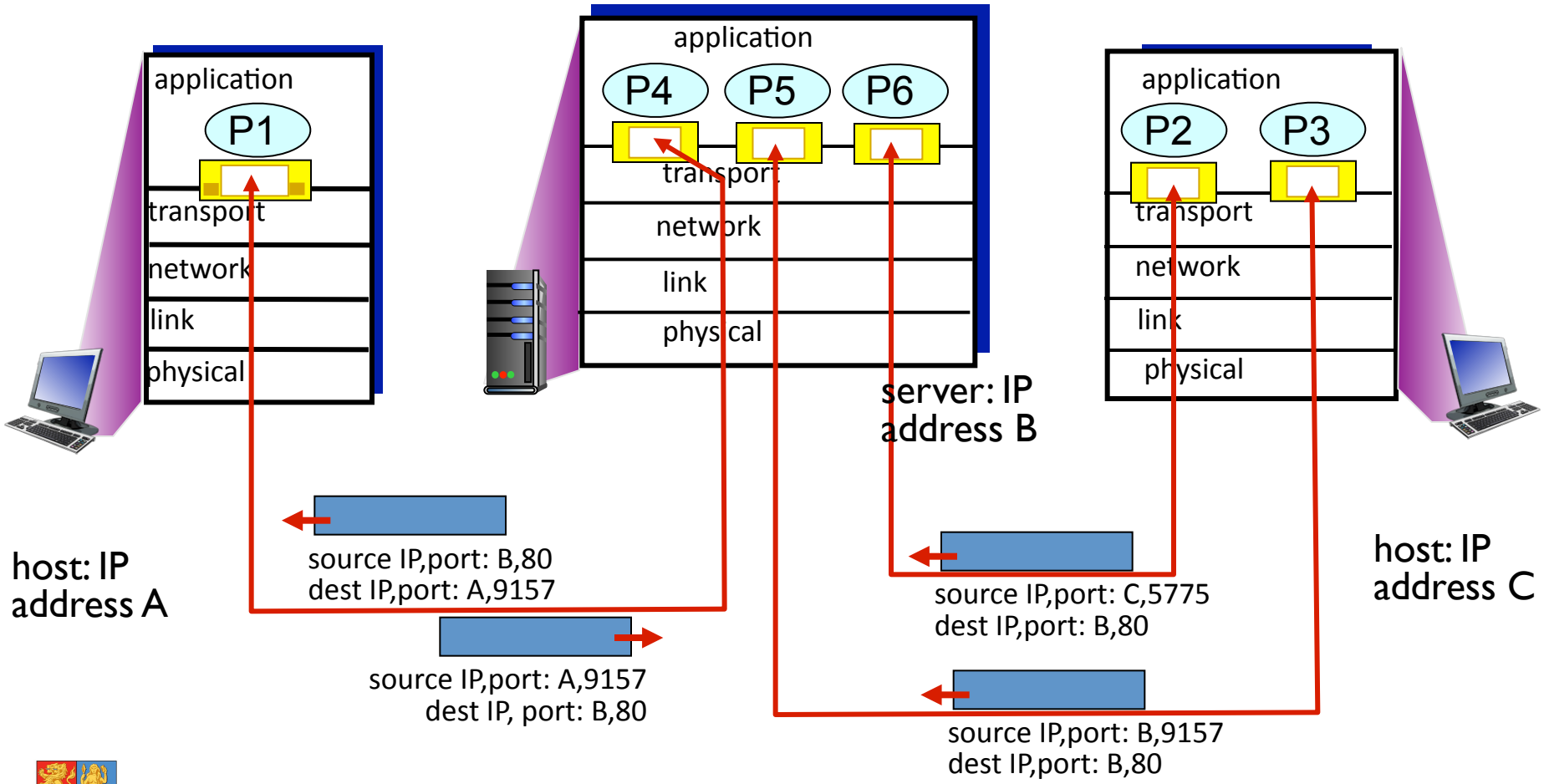
对每个不同的客户端来说，服务器都有不同的套接字

- web servers have different sockets for each connecting client
  - non-persistent HTTP will have different socket for each request

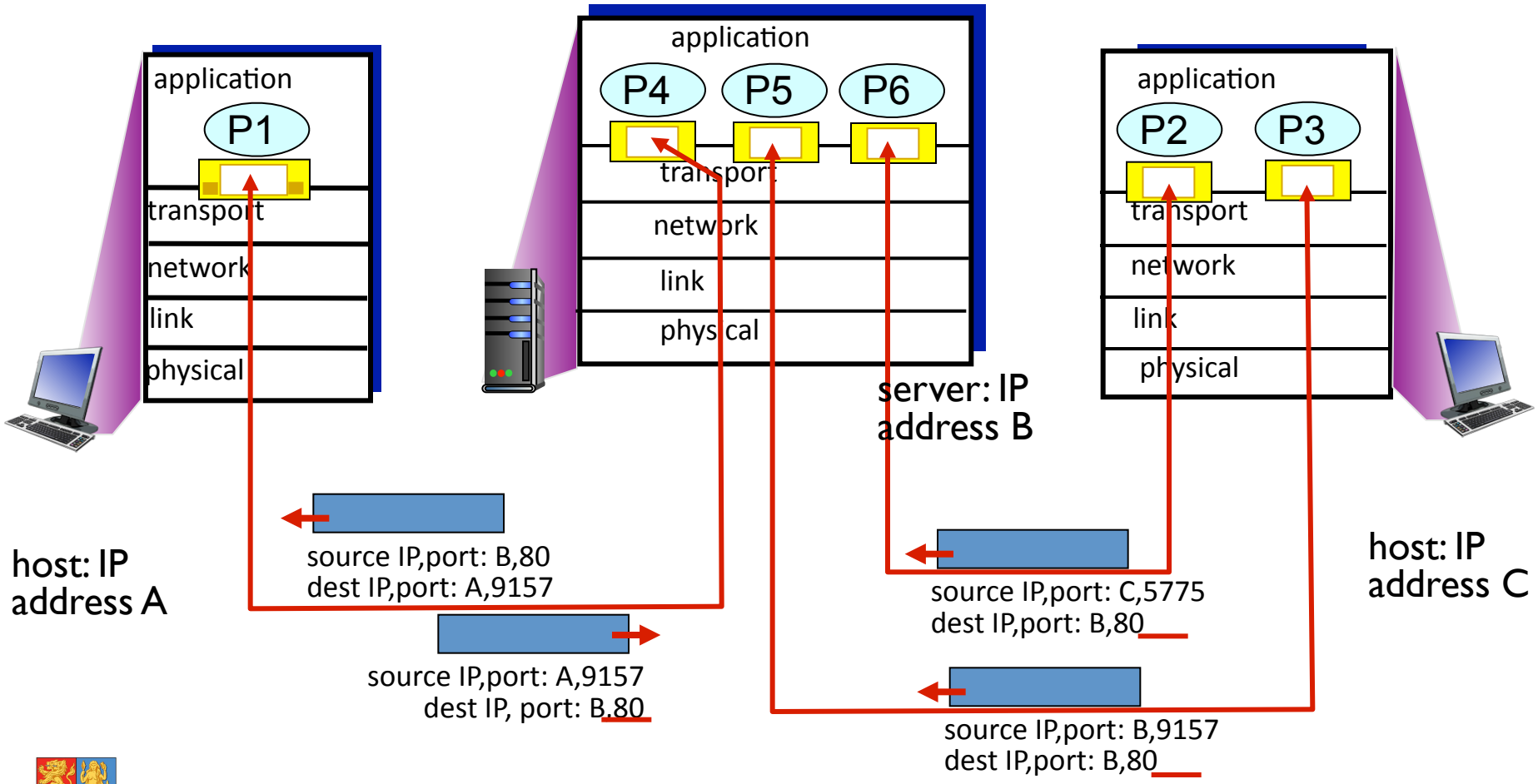
对于每个请求，非持久HTTP将具有不同的套接字

面向连接多路分配器的例子

Connection-oriented demux: example

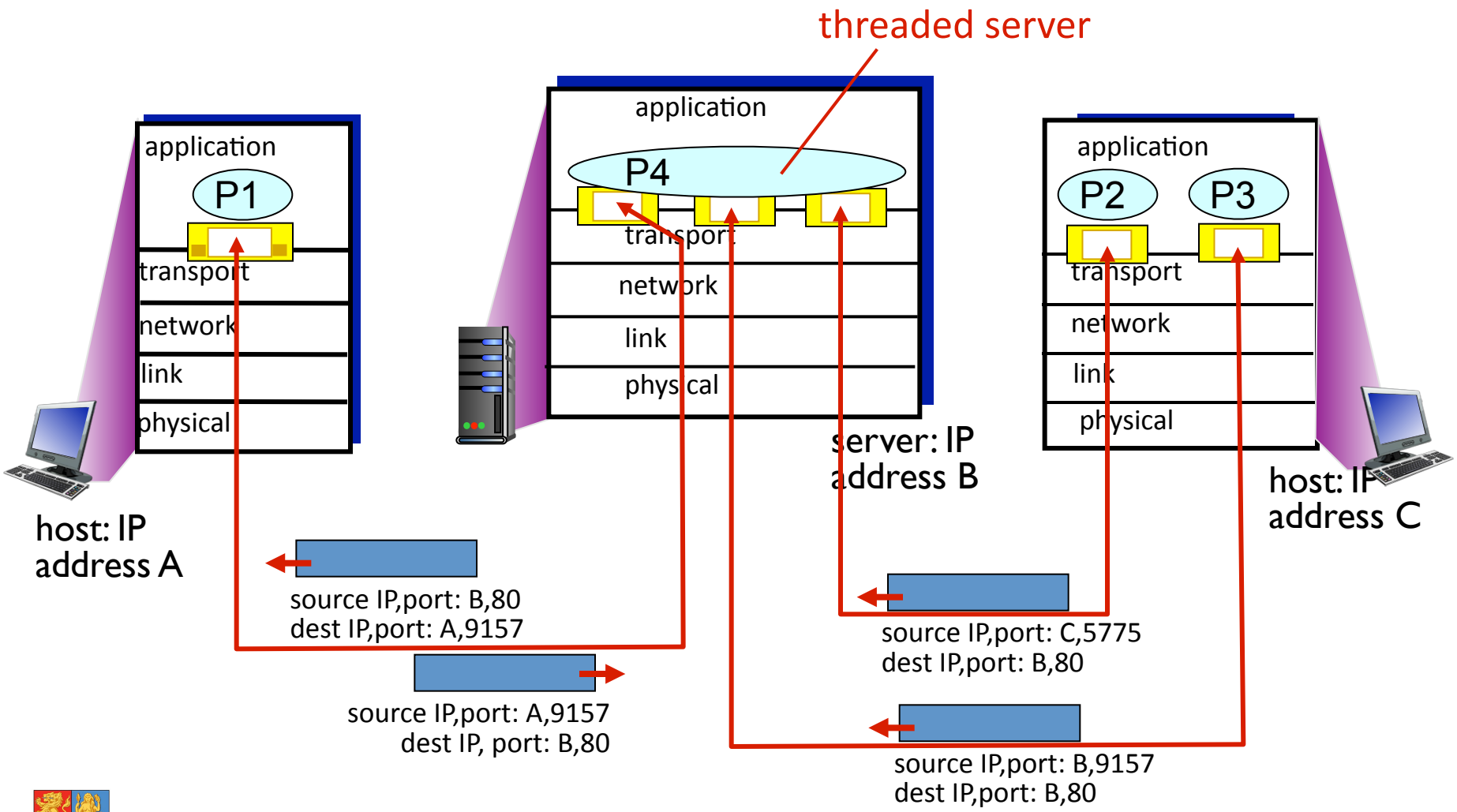


Connection-oriented demux: example

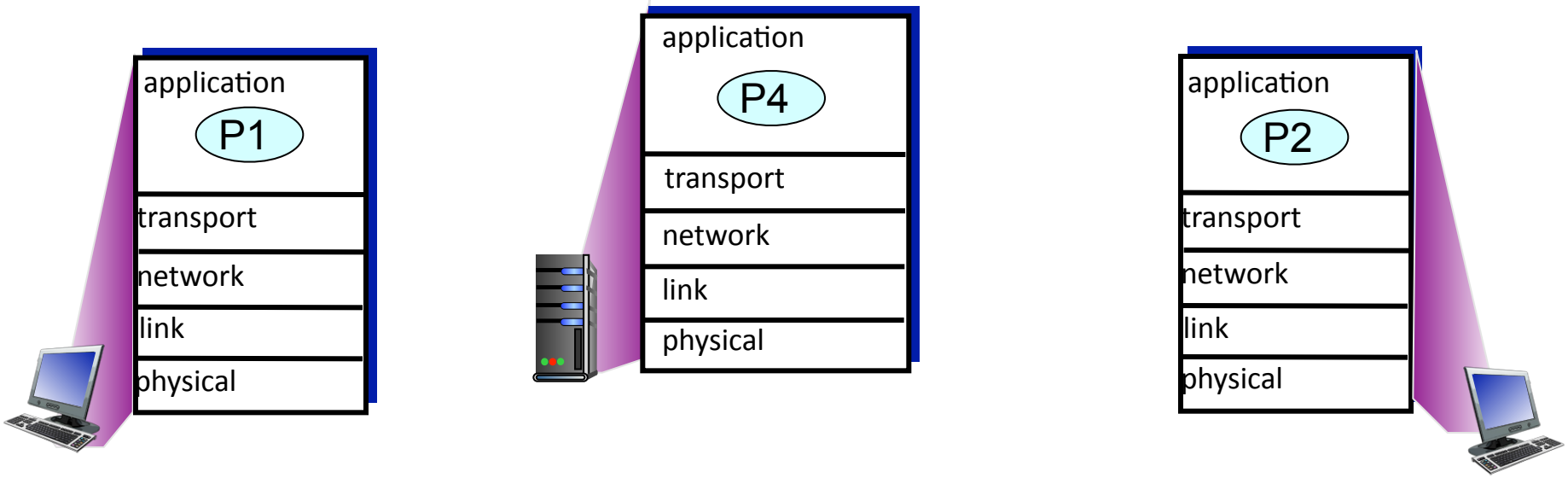


three segments, all destined to IP address: B,
dest port: 80 are demultiplexed to *different* sockets

Connection-oriented demux: example



Connectionless demux: example

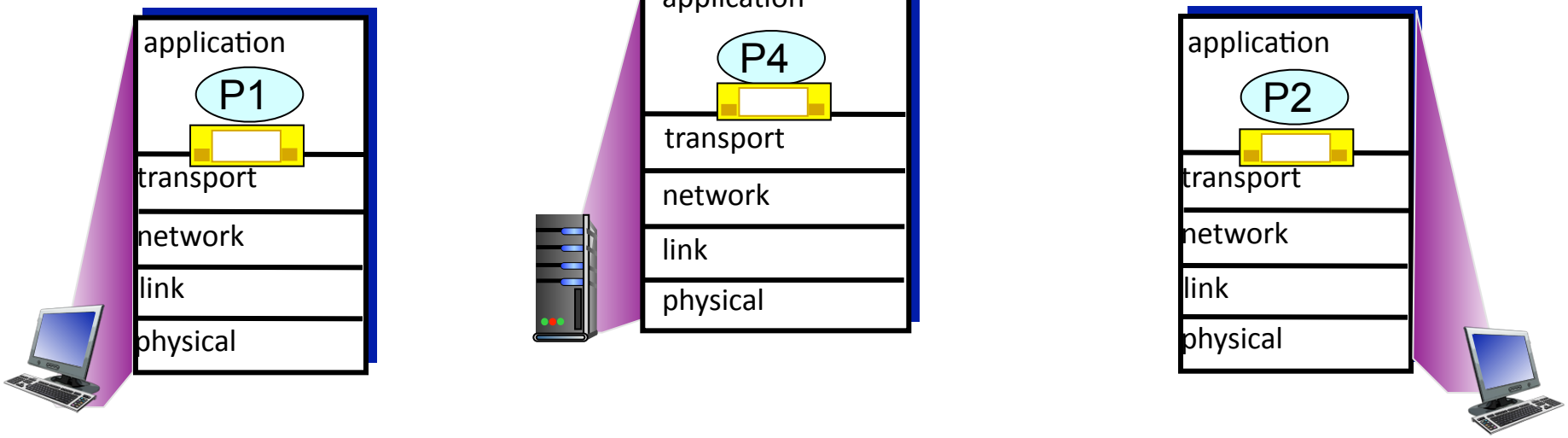


Connectionless demux: example

```
DatagramSocket
 mySocket2 = new
 DatagramSocket
 (9157);
```

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

```
DatagramSocket
 mySocket1 = new
 DatagramSocket
 (5775);
```

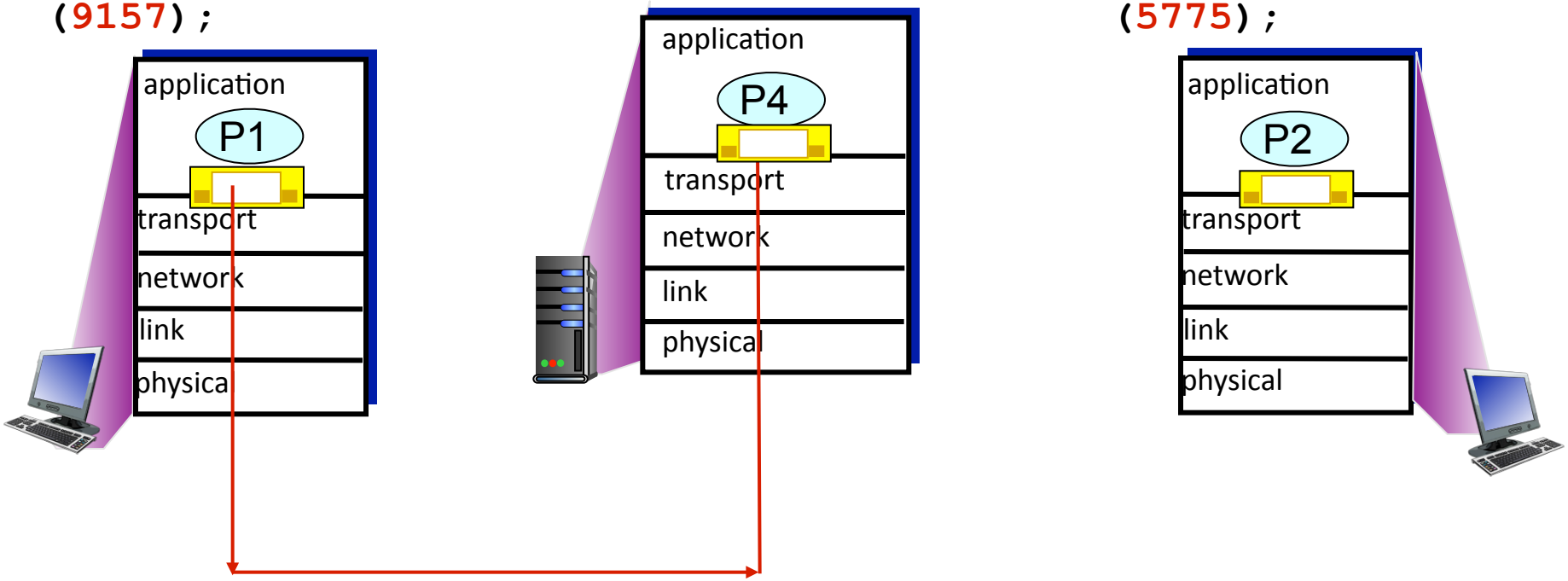


Connectionless demux: example

```
DatagramSocket
 mySocket2 = new
 DatagramSocket
 (9157);
```

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

```
DatagramSocket
 mySocket1 = new
 DatagramSocket
 (5775);
```

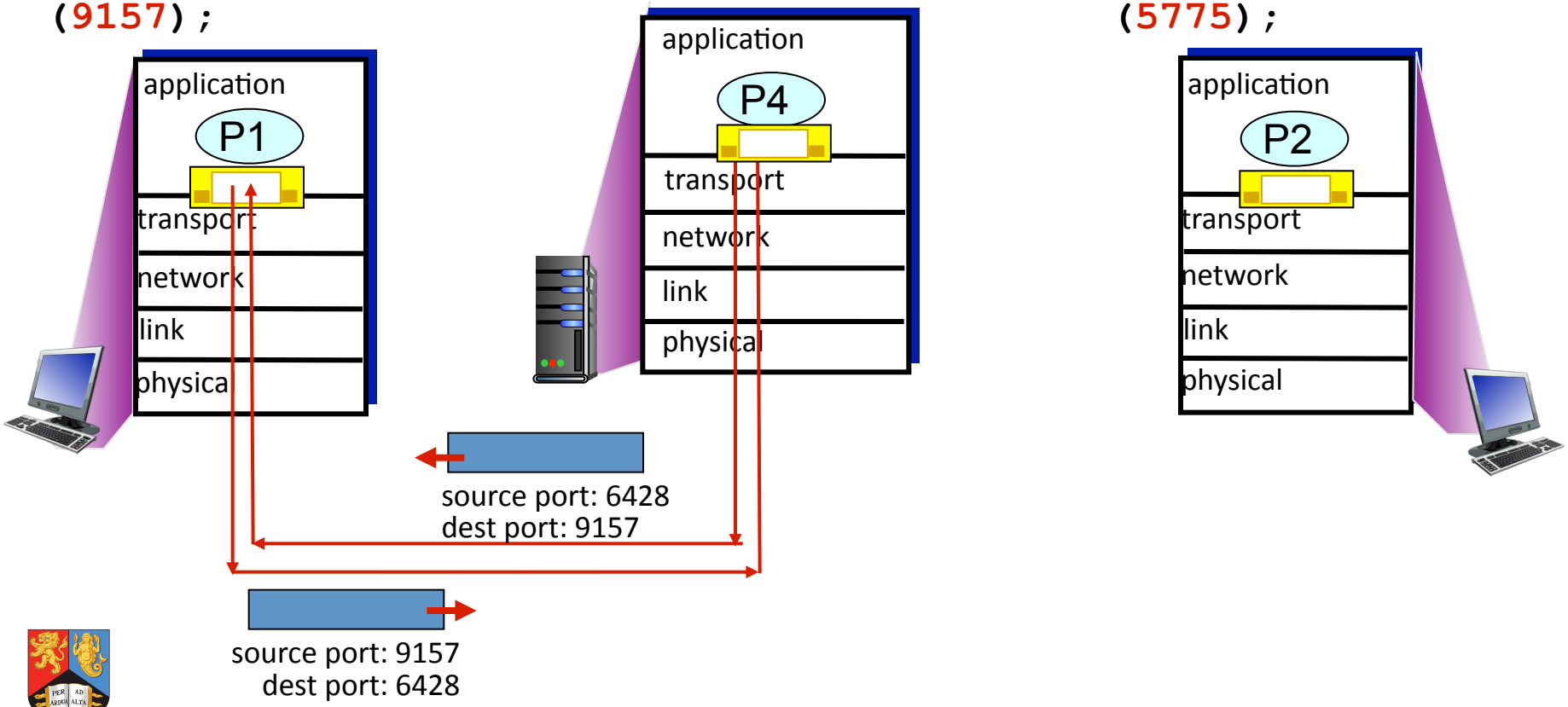


Connectionless demux: example

```
DatagramSocket
 mySocket2 = new
 DatagramSocket
 (9157);
```

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

```
DatagramSocket
 mySocket1 = new
 DatagramSocket
 (5775);
```

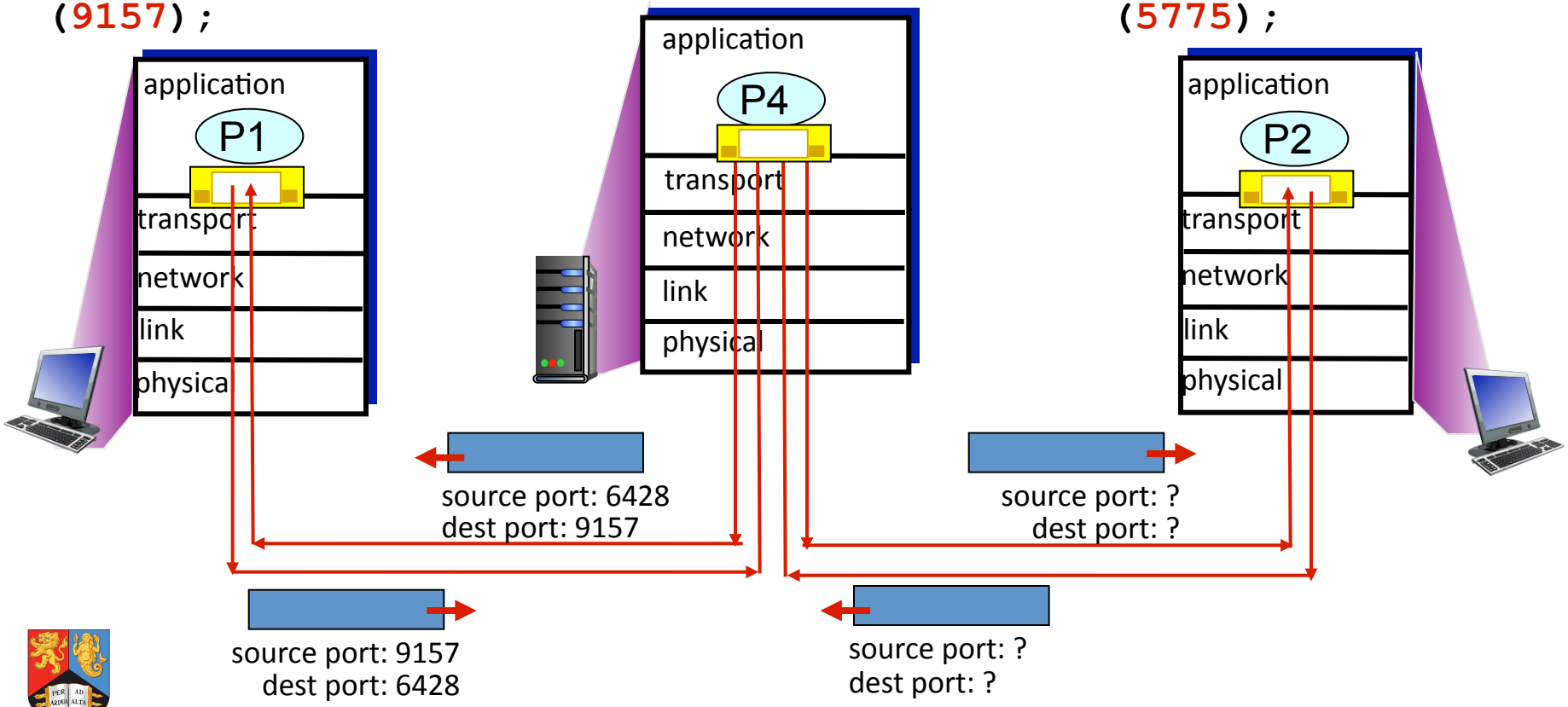


Connectionless demux: example

```
DatagramSocket
 mySocket2 = new
 DatagramSocket
 (9157);
```

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

```
DatagramSocket
 mySocket1 = new
 DatagramSocket
 (5775);
```



无连接的多路分解

Connectionless demultiplexing

创建套接字与端口号

- Create sockets with port numbers:

```
DatagramSocket mySocket1 = new
   DatagramSocket(12534);
```

- UDP socket identified by two-tuple:

(dest IP address, dest port number)

UPD 套接字由二元组验证

主机接收UDP段时:

- When host receives UDP segment:
  - checks destination port number in segment
  - directs UDP segment to socket with that port number
- IP datagrams with different source IP addresses and/or source port numbers can be directed to same socket

具有不同源IP地址和/或源端口号的IP数据报可以定向到同一个套接字

Connectionless demultiplexing

- *recall:* created socket has host-local port #:

```
DatagramSocket mySocket1
= new DatagramSocket(12534);
```

when creating datagram to send into UDP socket, you must specify

- destination IP address
- destination port #

- when host receives UDP segment:
  - checks destination port # in segment
  - directs UDP segment to socket with that port #

IP datagrams with *same dest. port #,* but different source IP addresses and/or source port numbers will be directed to *same socket* at dest

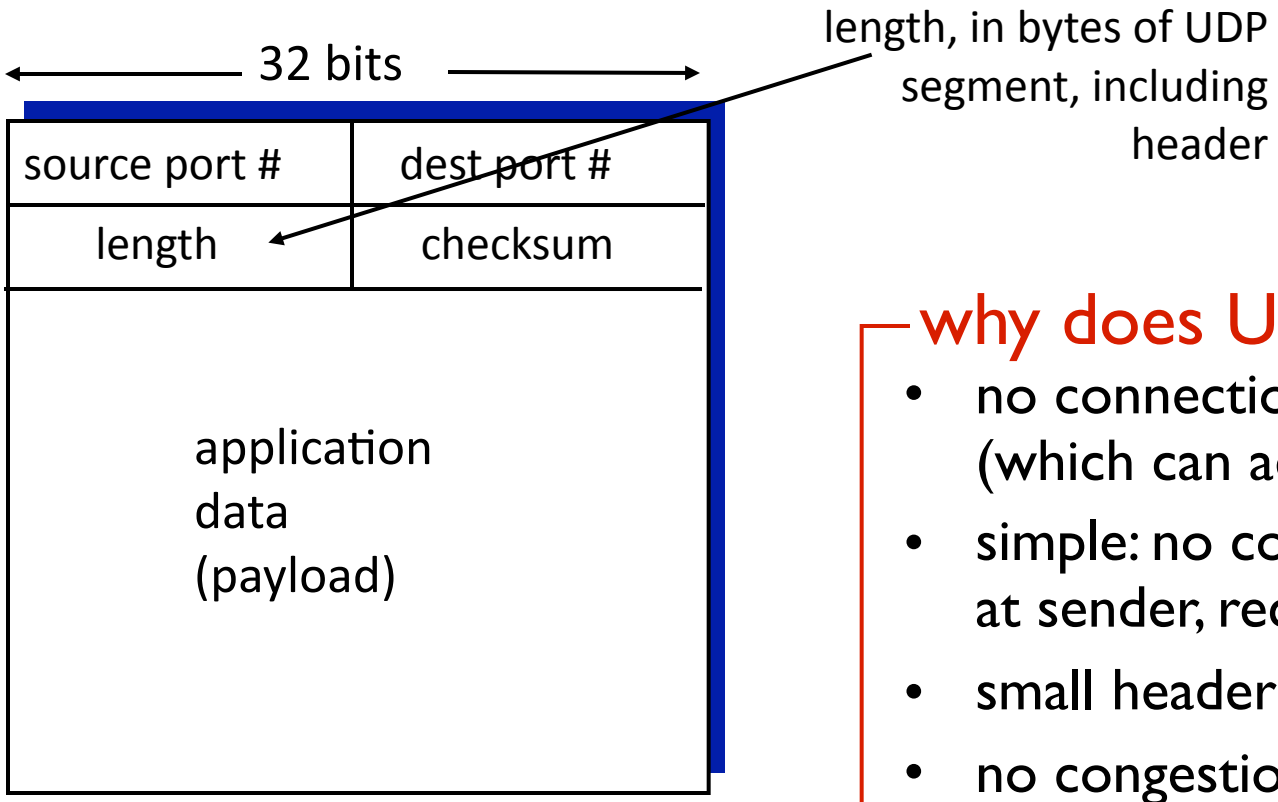
UDP: User Datagram Protocol [RFC 768]

UDP: send data to another end
no handshaking, very light

- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out-of-order to app
- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

- UDP use:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
  - SNMP
- reliable transfer over UDP:
  - add reliability at application layer
  - application-specific error recovery!

UDP: segment header



UDP segment format

why does UDP exist?

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

UDP checksum

Goal: detect "errors" (e.g., flipped bits) in transmitted segment

sender:

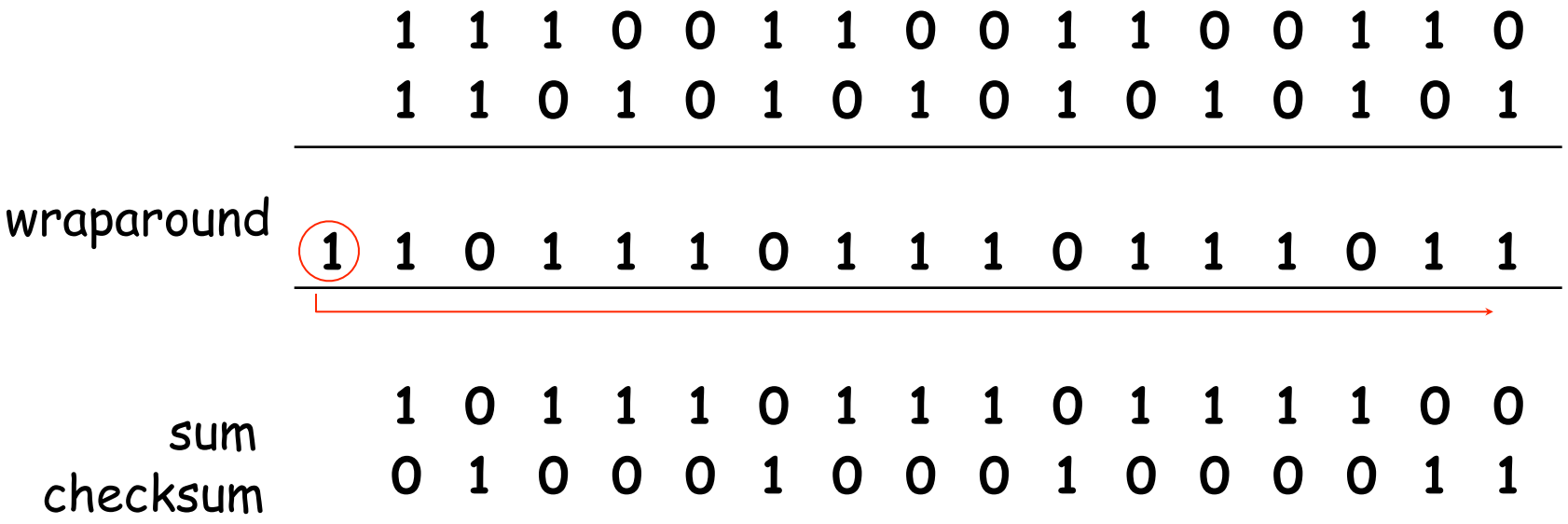
- treat segment contents, including header fields, as sequence of 16-bit integers, with checksum field set to zero
- "**Checksum is the 16-bit one's complement of the one's complement sum of a pseudo header of information from the IP header, the UDP header, and the data, padded with zero octets at the end (if necessary) to make a multiple of two octets**." (RFC 768 – that's all it says!)
- sender puts checksum value into UDP checksum field

receiver:

- compute checksum of received segment
- check if computed checksum equals checksum field value:

Internet checksum: example

example: add two 16-bit integers



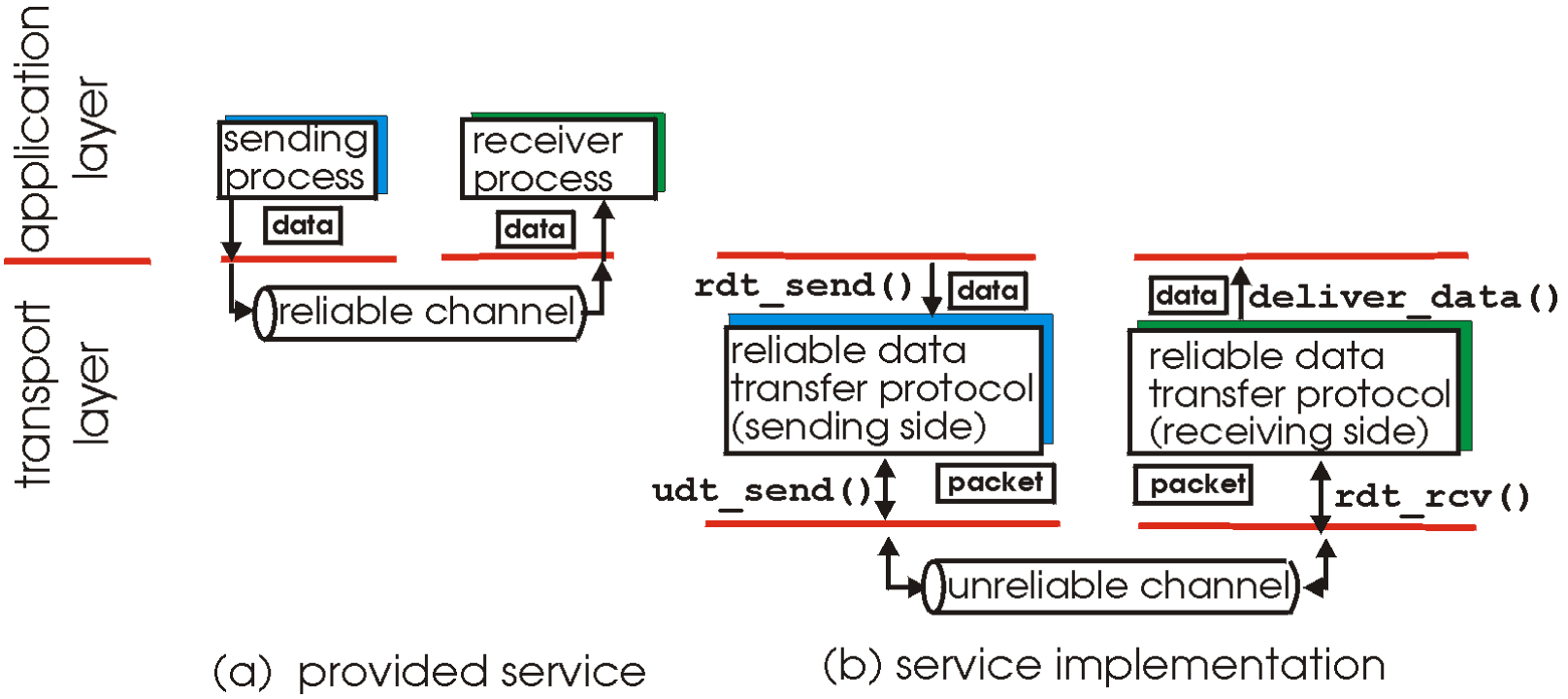
Note: when adding numbers, a carryout from the most significant bit needs to be added to the result

Implementation Issues

- The design is very 16-bit: IEN 45 in 1978 is all about pdp11s, and the non-16-bit machines are actually 36 bit (pdp10s).
  - 32 bit add (or 64 bit add) is expensive on such boxes
- You can get the effect of the end wraparound by summing 16 bit quantities in a 32 bit word and then adding the top and bottom halves together repeatedly until the top half is zero.
- RFC1071 has the gory details: it took until 1988 to publish
  - And is itself full of ancient history, like Cray assember
- Allegedly, at the first interworking "bake off" **no** implementations actually interworked…

Principles of reliable data transfer

- important in application, transport, link layers

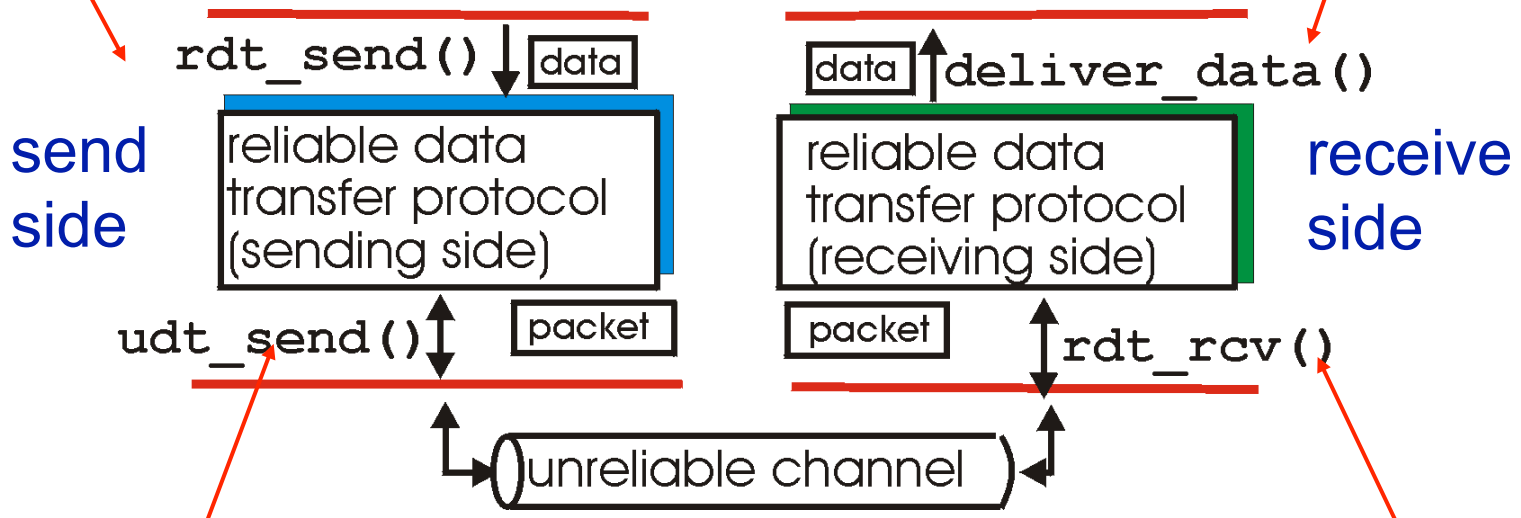


- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Reliable data transfer: getting started

`rdt_send()`: called from above, (e.g., by application). Passed data to deliver to receiver upper layer

`deliver_data()`: called by **rdt** to deliver data to upper



`udt_send()`: called by rdt, to transfer packet over unreliable channel to receiver

`rdt_rcv()`: called when packet arrives on rcv-side of channel

In a perfect world…

- Channels are perfectly reliable:
  - No errors (i.e., flipped bits)
  - No packet loss
  - Packets received by the receiver in order
- If a channel is perfectly reliable it is sufficient to send and receive data: you don't need any additional mechanisms such as acknowledgements
- But the world is not perfect and errors happen…
- …and you need to recover from errors!

ACK机制(确认)

Acknowledgement mechanisms

首先，我们假设这个频道可能会引入错误(如算坏的包)，但没有丢包情况

- Let us assume first that the channel might introduce errors (i.e., corrupted packets) but packets are not lost
- Underlying channel may flip bits in packet 底层通道可能会翻转包中的bits
  - Checksum is used to detect bit in errors Checksum用于检测错误的bit
- The key problem is how to recover from errors
- Two mechanisms 如何从错误中恢复才是关键：两个机制
  - *acknowledgements (ACKs):* receiver explicitly tells sender that the packet has received correctly

    接收者显式地告诉发送者包已经正确接收

  OR

  - *negative acknowledgements (NAKs):* receiver explicitly tells sender that packet had errors 接收者显式地告诉发送者包有错误
    - Sender retransmits packet on receipt of NAK

      发送方在收到NAK后重新发送包

如果ACK / NAK被损坏会发生什么?

What happens if ACKs/NACKs get corrupted?

- You can have errors in packet (data) transmission but also in ACK/NACK transmission 这是有可能发生的
- And so what happens if ACKs/NACKs get corrupted?
  - Essentially, the sender does not know what happens at the receiver 基本上，发送方不知道接收方收到了什么
- One solution is to simply retransmit, **but** you can't simply do that, since you might have **duplicates** at the receiver 其中一个解决方案是重新发送，但你不能单纯这么做，因为接收者可能收两次
- Problem: how to handle duplicates

  如何应对重复收到呢?

注意：Checksum(校验和)是很弱的

Note the checksums are weak

- The IP checksum is weak, and won't reliably detect multiple bit flips, depending on where they happen. IP校验和很弱，不能可靠地检测到多比特翻转(这取决于它们发生在哪里)
- Usually a channel that unreliable won't pass traffic successfully, and the TCP protocol itself will collapse 通常一个不可靠的通道不会成功地通过traffic，TCP协议本身也会崩溃
- But If you are moving data whose contents matter bit-for-bit, you need to use a stronger checksum over the whole thing (SHA256 is good)

但如果你是在转移一些在意顺序的数据，你需要用更强大的校验方式来应对整个文件 （SHA256）

重复事件的处理

Handling Duplicates

为了处理重复的包，每个包都应该有一个序列号作为唯一标识

- In order to handle duplicates, every packet should have a **sequence number** that identifies that packet uniquely
- The receiver discards (i.e., does not deliver the packet to the layer above) the duplicate packets

接受方将会丢弃重复的包（即不会将包发送到上面的层 app层之类的）

And now let's consider the general case:

channels **do** lose packets…

搞清楚了这个 现在就可以开始讨论通道的丢包了

And a channel can also lose packets…

底层通道是可能会丢包的（数据本身以及回应的ACK都有可能丢）

- Underlying channel can also lose packets (data, ACKs)
  - checksum, sequence numbers , ACKs, retransmissions are of help, but not enough… 校验和，序列号，ACK，重发，是有帮助的 但不够
- Sender waits "reasonable" amount of time for ACK

服务器会在一个合理时间内等待ACK

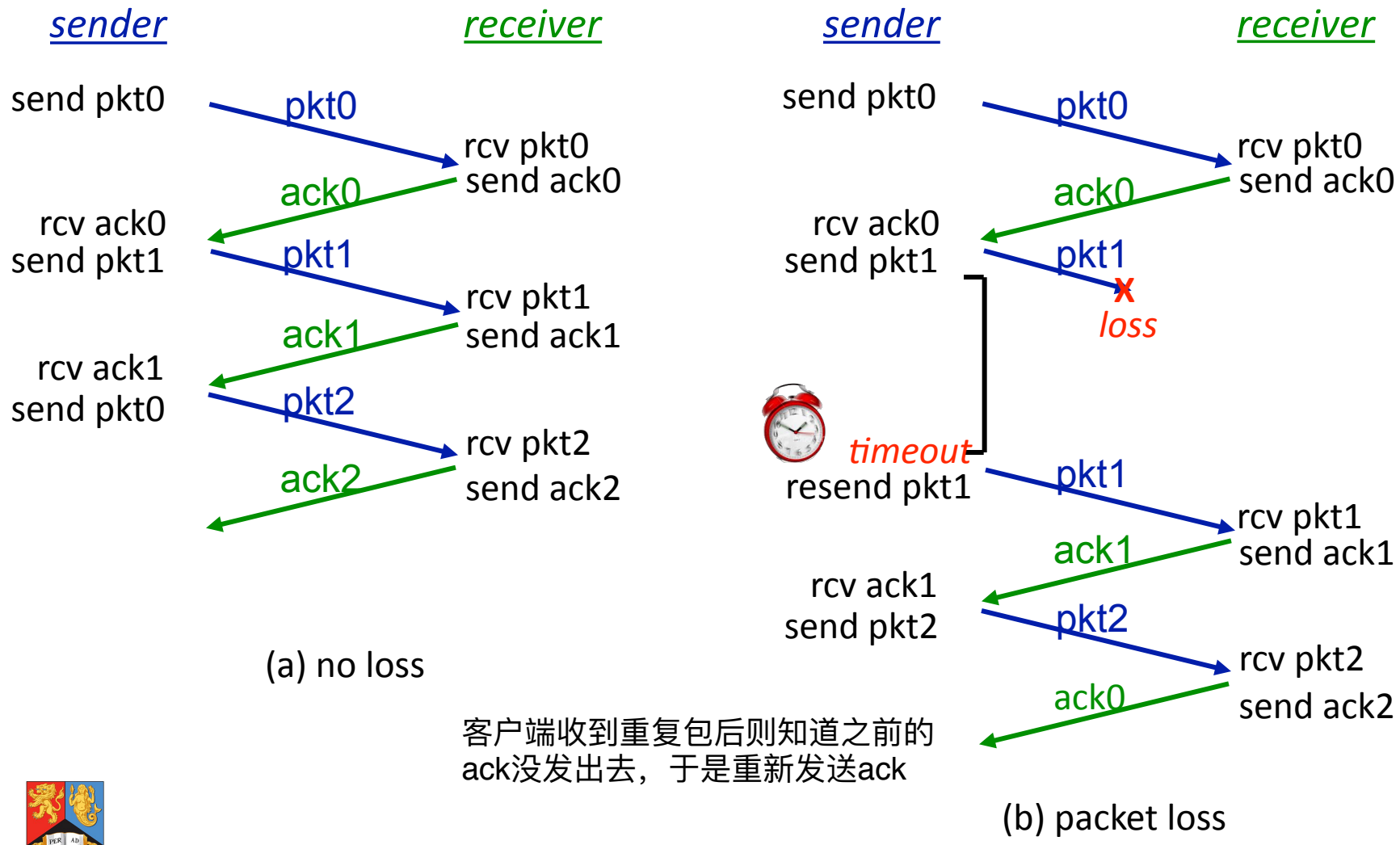
  - retransmits if no ACK received in this time 如果一直没收到则会重发
  - if pkt (or ACK) just delayed (not lost): 如果包或ACK没有丢只是延迟了
    - retransmission will be duplicate, but seq. #'s already handles this
    - receiver must specify seq # of packets being ACKed

重发依旧会继续，重复则会发生 不过序列号足以解决这个；接收方需声明哪些序列号的包被ACK了

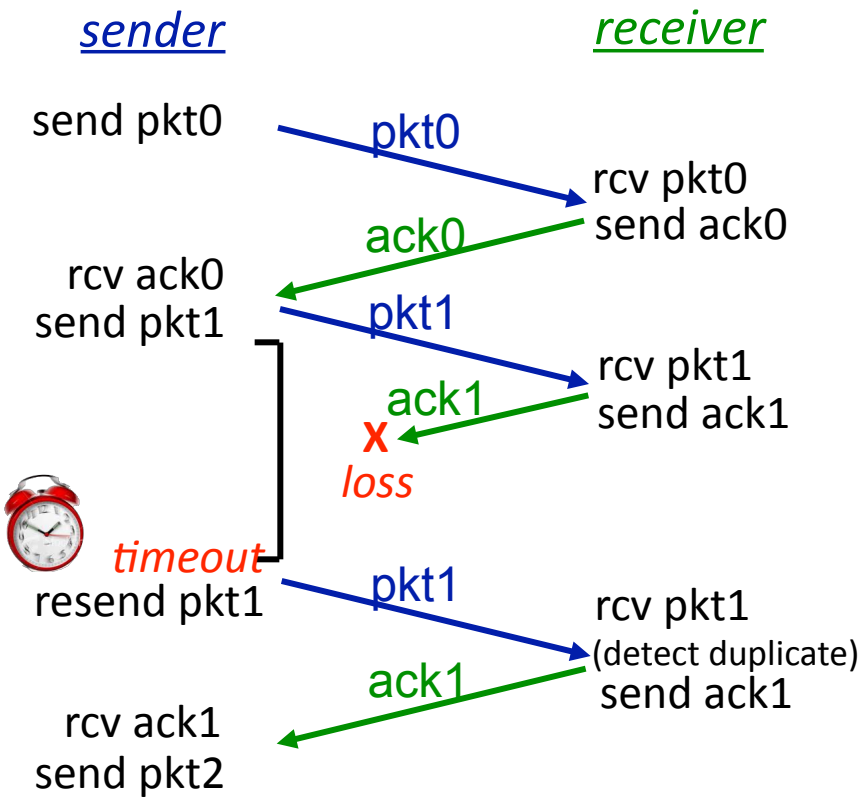
- This mechanism requires countdown timer

该机制需要一个倒数的计时器

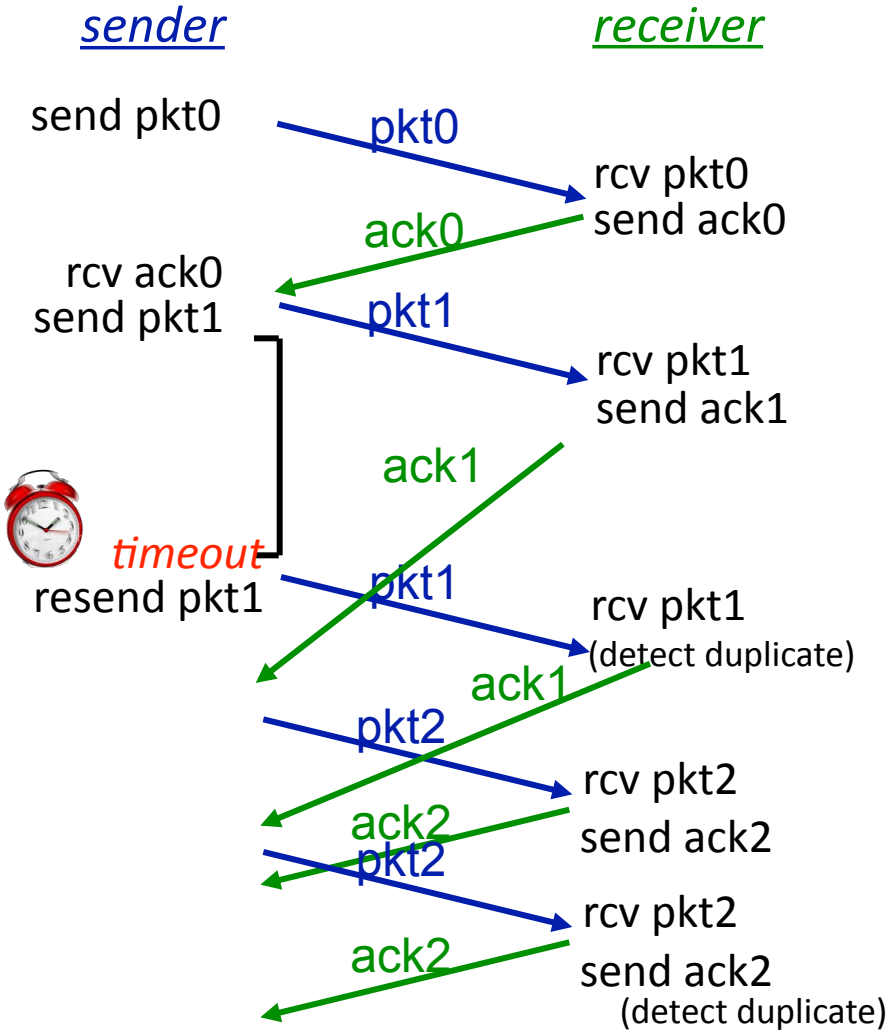
ACKs and time-outs in action



ACKs and time-outs in action



(c) ACK loss



过早超时 / 重复ACK

(d) premature timeout/ delayed ACK

但延迟会毁掉这个方法

Latency kills this method

假设我们只允许一次有一个包处于未完成状态

- Suppose we only allow one packet outstanding at a time
- Speed of light is $3x10^8 ms^{-1}$.
- Plenty of paths are $3x10^6 m$ (short transatlantic path to east coast).
- So 20ms round trip time, assuming infinitely fast routers and computers.
- 50 packets per second @1500bytes/packet = 75KB/sec ~ **600Kb/sec** maximum.

横跨大西洋最短路径是3x10^6, 光速的1%

假设路由器和电脑速度无限快，则有20毫秒的往返时间。

每秒50个包，每个包1500字节 也就是75k/b 约等于600Kb/sec 也就是极限了

UNIVERSITY OF BIRMINGHAM

管线式协议

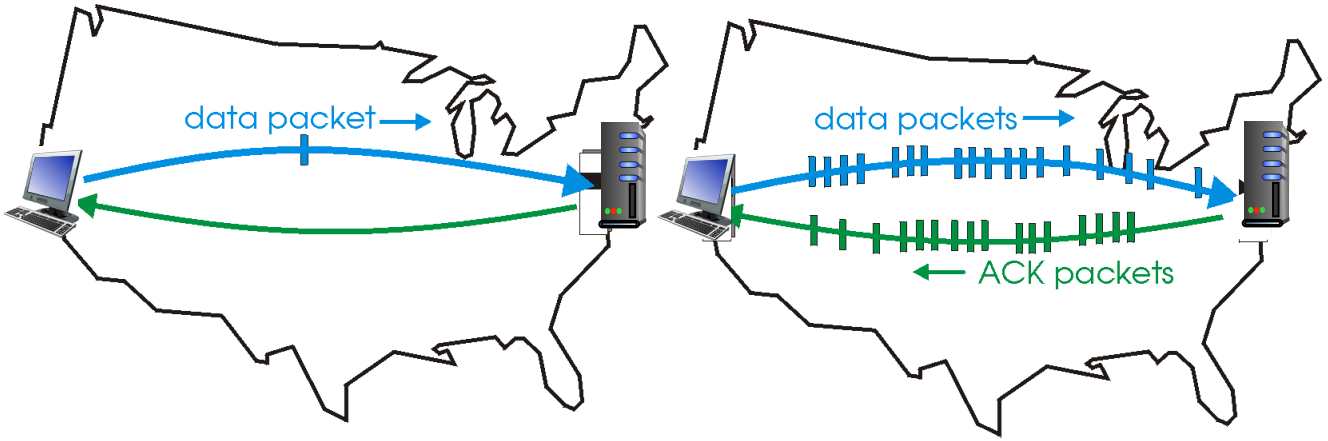
Pipelined protocols

该协议的发送方允许多个包处于运行中的状态，也就是“尚未被确认的包”

pipelining: sender allows multiple, “in-flight”, yet-to-be-acknowledged packets

- range of sequence numbers must be increased
- buffering at sender and/or receiver

1. 序列号的范围必须要增加
2. 接收方或发送方必须有缓冲



(a) a stop-and-wait protocol in operation

(b) a pipelined protocol in operation

- two generic forms of pipelined protocols: *go-Back-N, selective repeat*

管线式协议的两个通用形式：

管线式协议总结

Pipelined protocols: overview

形式1:Go-back-N

发送方在管道中最多可以有N个未ack的包

Go-back-N:

- sender can have up to N unacked packets in pipeline 接收方只发送累计的ack包
- receiver only sends *cumulative ack*
  - doesn't ack packet if there's a gap 如果有间隙就不响应包
- sender has timer for oldest unacked packet
  - when timer expires, retransmit *all* unacked packets

发送方(服务器)会为最早发送包计时，当计时结束后将重传所有未ack的包

形式2: 选择性重复

发送方在管道中最多可以有N个未ack的包

Selective Repeat:

- sender can have up to N unack'ed packets in pipeline
- rcvr sends *individual ack* for each packet

  接收方为每个包发送单独的ack
- sender maintains timer for each unacked packet
  - when timer expires, retransmit only that unacked packet

发送方依旧为未ack的包计时，当计时结束仅传输那个具体未ack的包

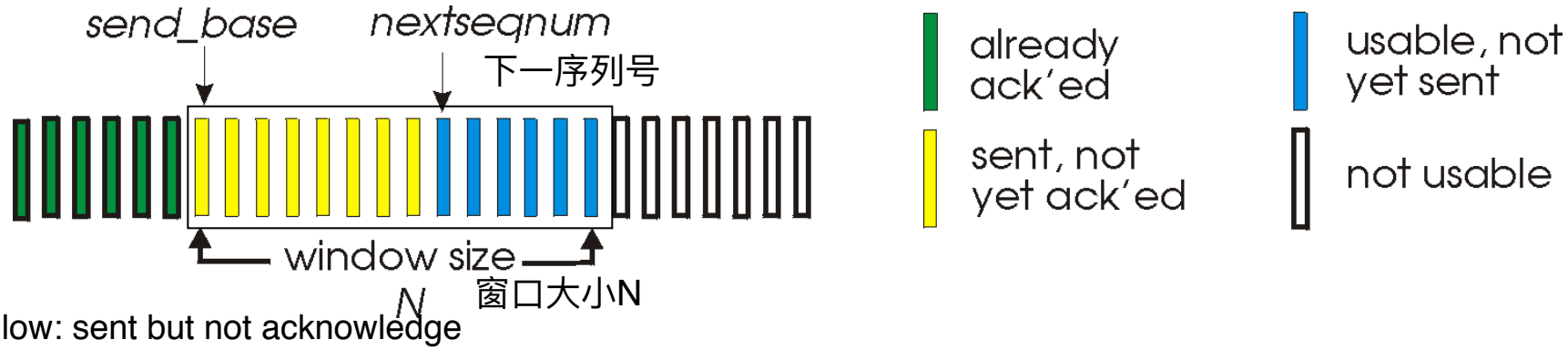
Go-Back-N策略：发送方（服务器）

Go-Back-N: sender

在packet header中有k位序列号

- k-bit sequence number in packet header
- "window" of up to N, consecutive unacked packets allowed

最多有n个"窗口期"，连续未应答的包是被允许的



yellow: sent but not acknowledge

- ACK(n): ACKs all pkts up to, including sequence number *n* - *"cumulative ACK"* ACK(n)意思为响应所有的包，包含序列号n的累计包
  - may receive duplicate ACKs (see receiver) 可能会收到重复的ACK响应(去看下一页)
- timer for oldest in-flight pkt 为最早发的包设计时
- *timeout(n):* retransmit packet n and all higher sequence number packets in window 超时(n):在窗口中重新发送包n和所有更高序列号的包

Go-Back-N形式的接收方

Go-Back-N: receiver

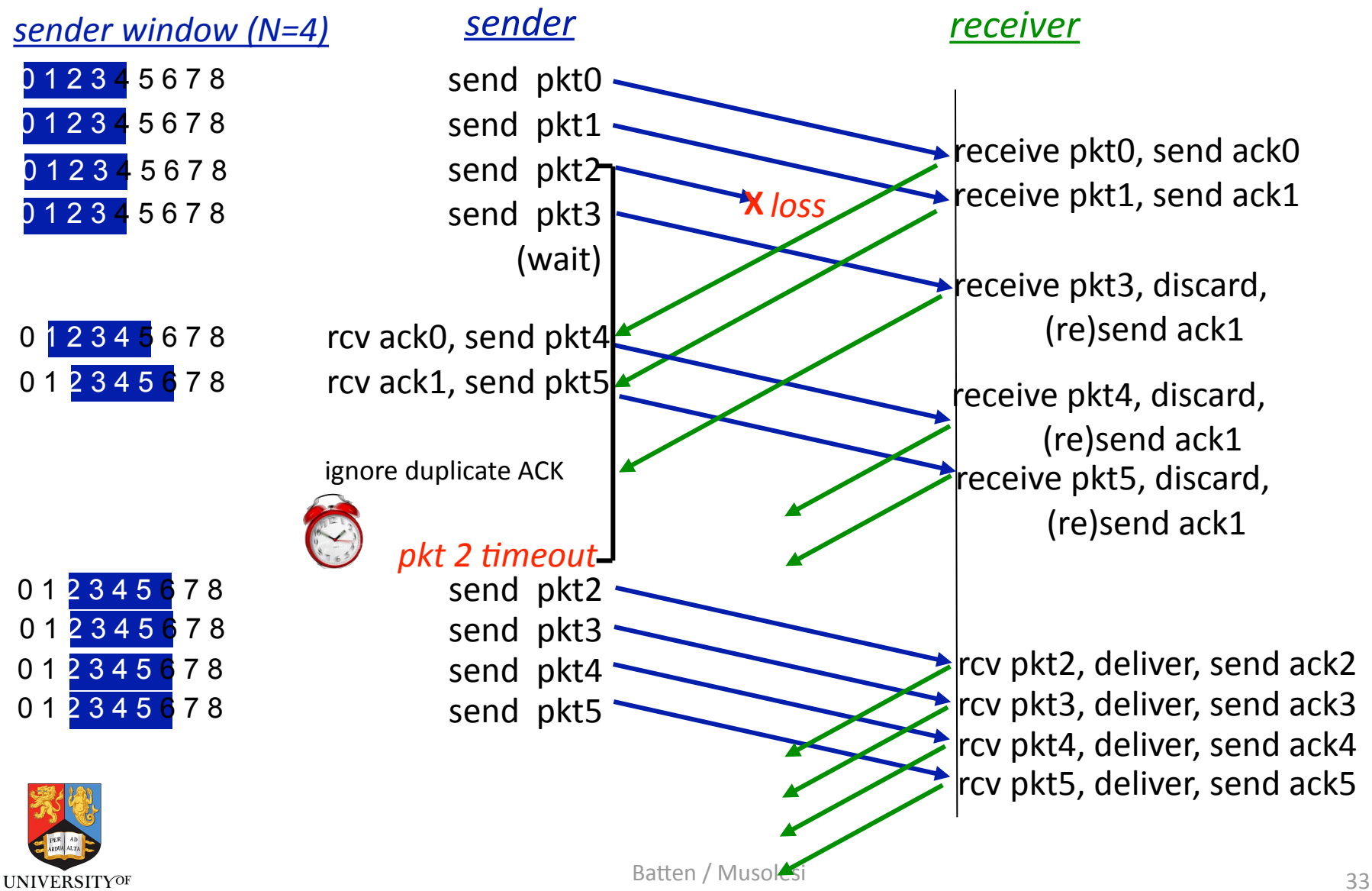
ACK-only: 始终对正确收到的包发送最序号(按顺序的)的ACK响应

ACK-only: always send ACK for correctly-received packet with highest *in-order* sequence number

- may generate duplicate ACKs
- need only remember **`expectedseqnum`**

- out-of-order packets:
  - discard (don't buffer): *no receiver buffering!*
  - re-ACK packets with highest in-order sequence number

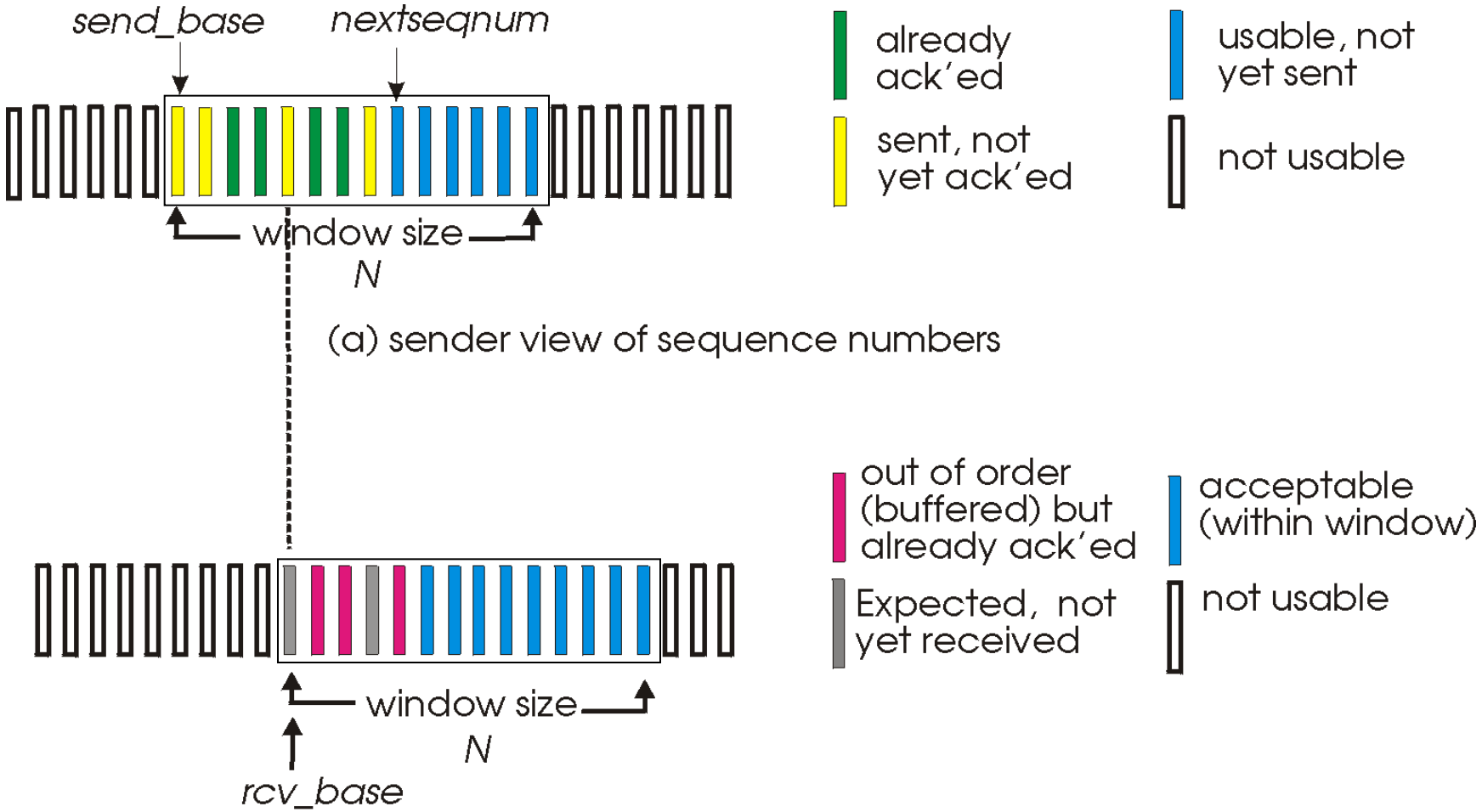
Go-Back-N in action



Selective repeat

- Receiver *individually* acknowledges all correctly received packets
  - Buffers packets, as needed, for eventual *in-order delivery* to upper layer
- Sender only resends packets for which ACK not received
  - sender timer for each unACKed packet
- sender window
  - *N* consecutive sequence numbers
  - limits sequence numbers of sent, unACKed pkts

Selective repeat: sender, receiver windows



Selective repeat

sender

data from above:

- if next available sequence number in window, send packet

timeout(n):

- resend pkt n, restart timer

ACK(n) in [sendbase,sendbase+N]:

- mark pkt n as received
- if n smallest unACKed pkt, advance window base to next unACKed seq number

receiver

pkt n in [rcvbase, rcvbase+N-1]

- send ACK(n)
- out-of-order: buffer
- in-order: deliver (also deliver buffered, in-order pkts), advance window to next not-yet-received pkt

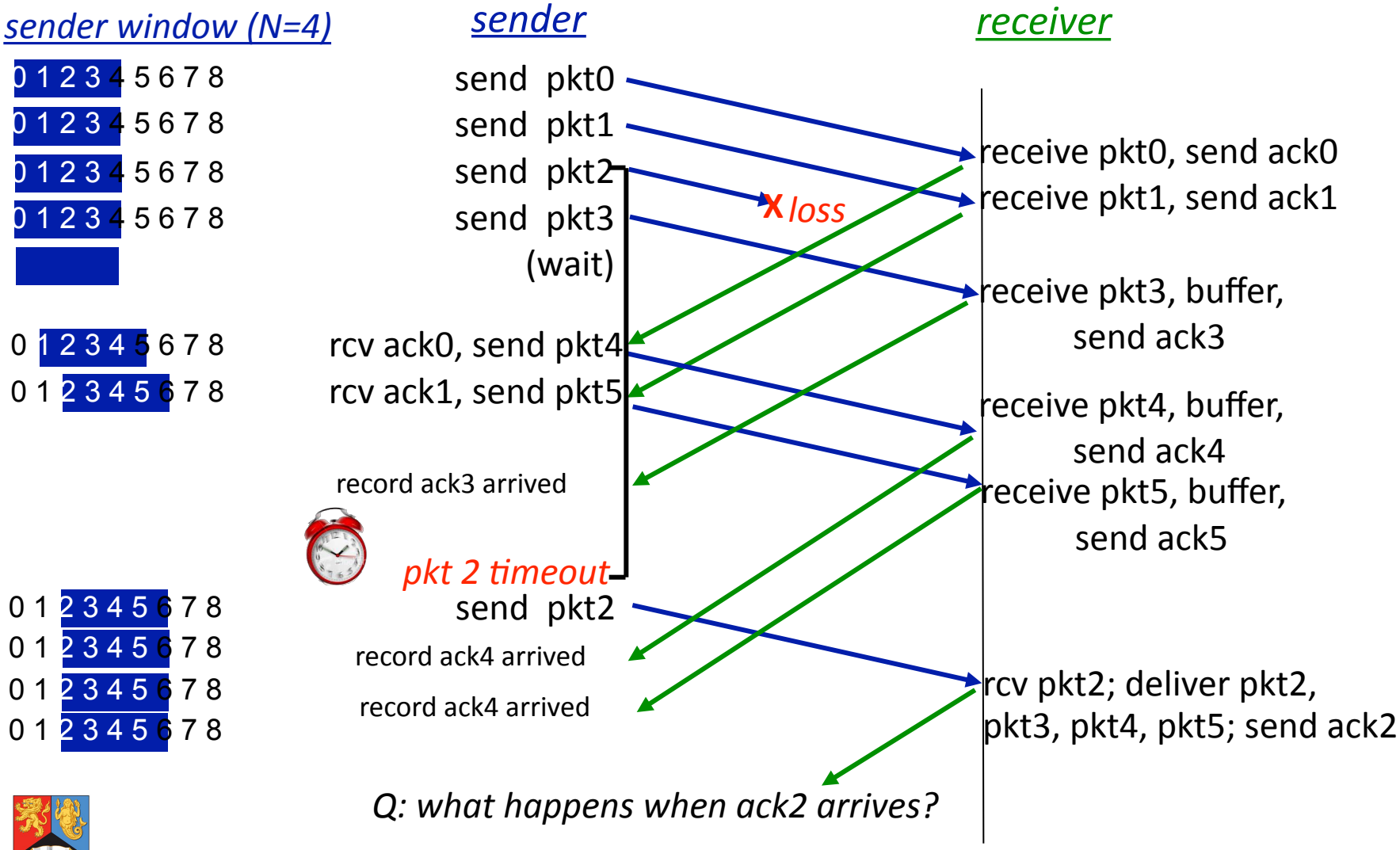
pkt n in [rcvbase-N,rcvbase-1]

- ACK(n)

otherwise:

- ignore

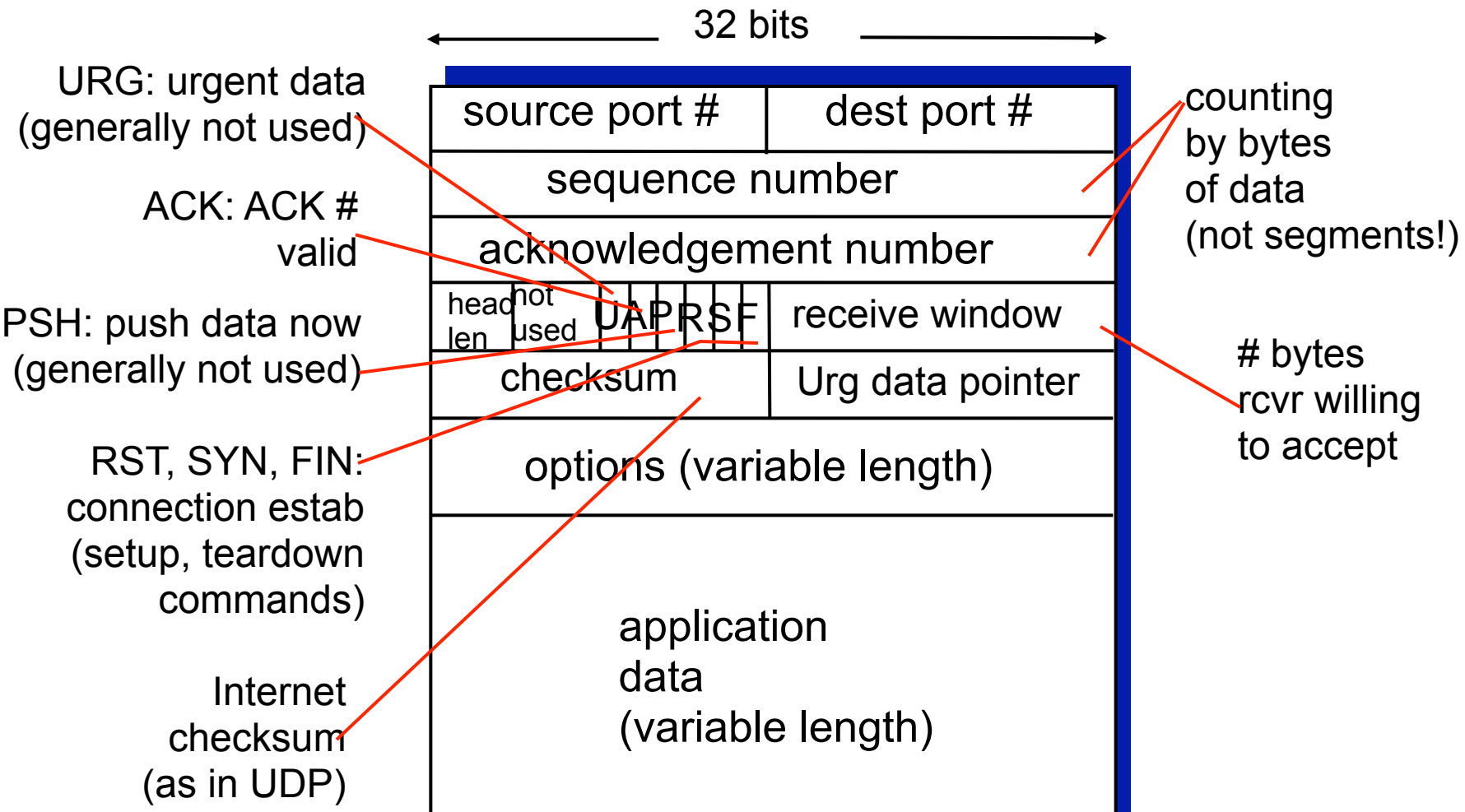
Selective repeat in action



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

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte stream:*
  - no "message boundaries"
- pipelined:
  - TCP congestion and flow control set window size
- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- connection-oriented:
  - handshaking (exchange of control msgs) inits sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

TCP segment structure



TCP seq. numbers, ACKs

sequence numbers:

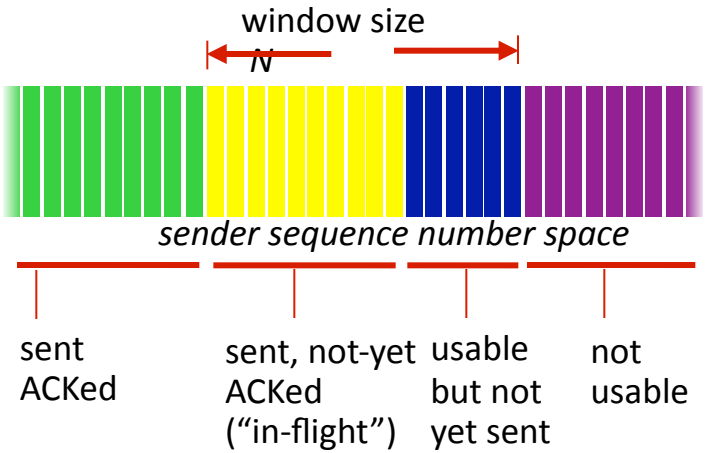
- byte stream "number" of first byte in segment's data

acknowledgements:

- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

- A: TCP spec doesn't say, - up to implementor



A

TCP seq. numbers, ACKs

sequence numbers:

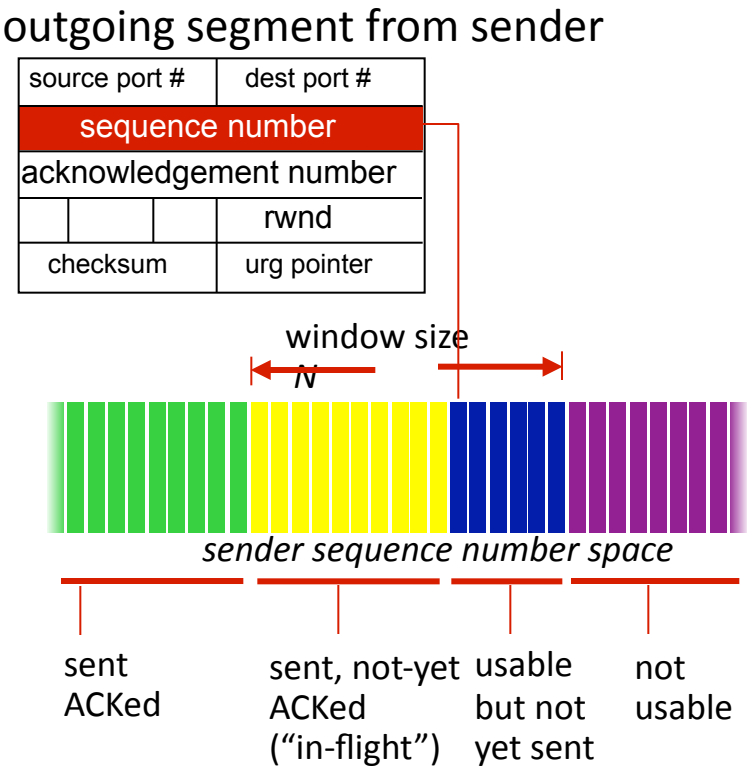
- byte stream "number" of first byte in segment's data

acknowledgements:

- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

- A: TCP spec doesn't say, - up to implementor



A

TCP seq. numbers, ACKs

sequence numbers:

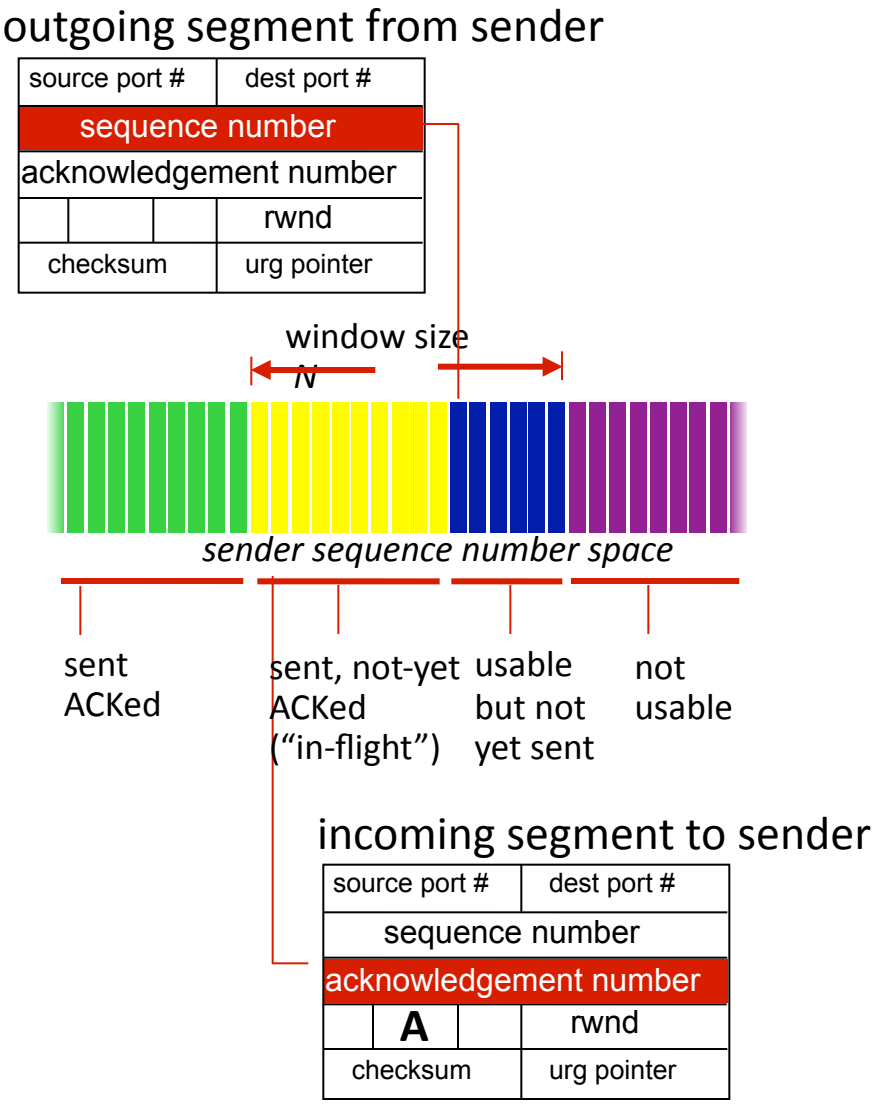
- byte stream "number" of first byte in segment's data

acknowledgements:

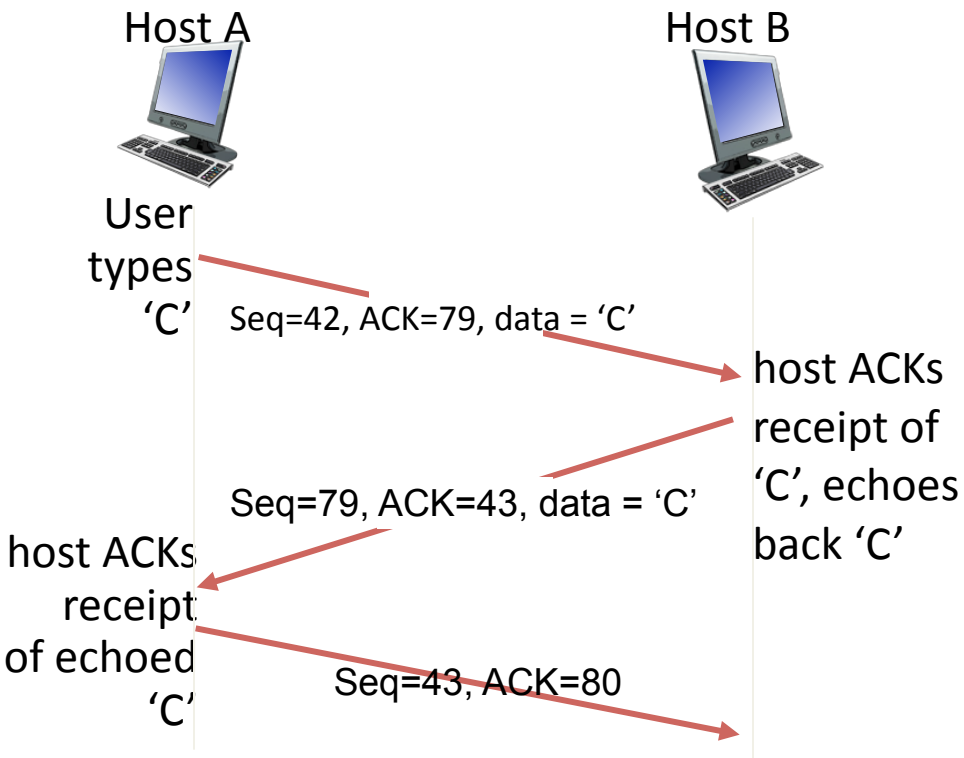
- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

- A: TCP spec doesn't say, - up to implementor



TCP seq. numbers, ACKs



simple telnet scenario

TCP round trip time, timeout

Q: how to set TCP timeout value?

- longer than RTT
  - but RTT varies
- *too short:* premature timeout, unnecessary retransmissions
- *too long:* slow reaction to segment loss

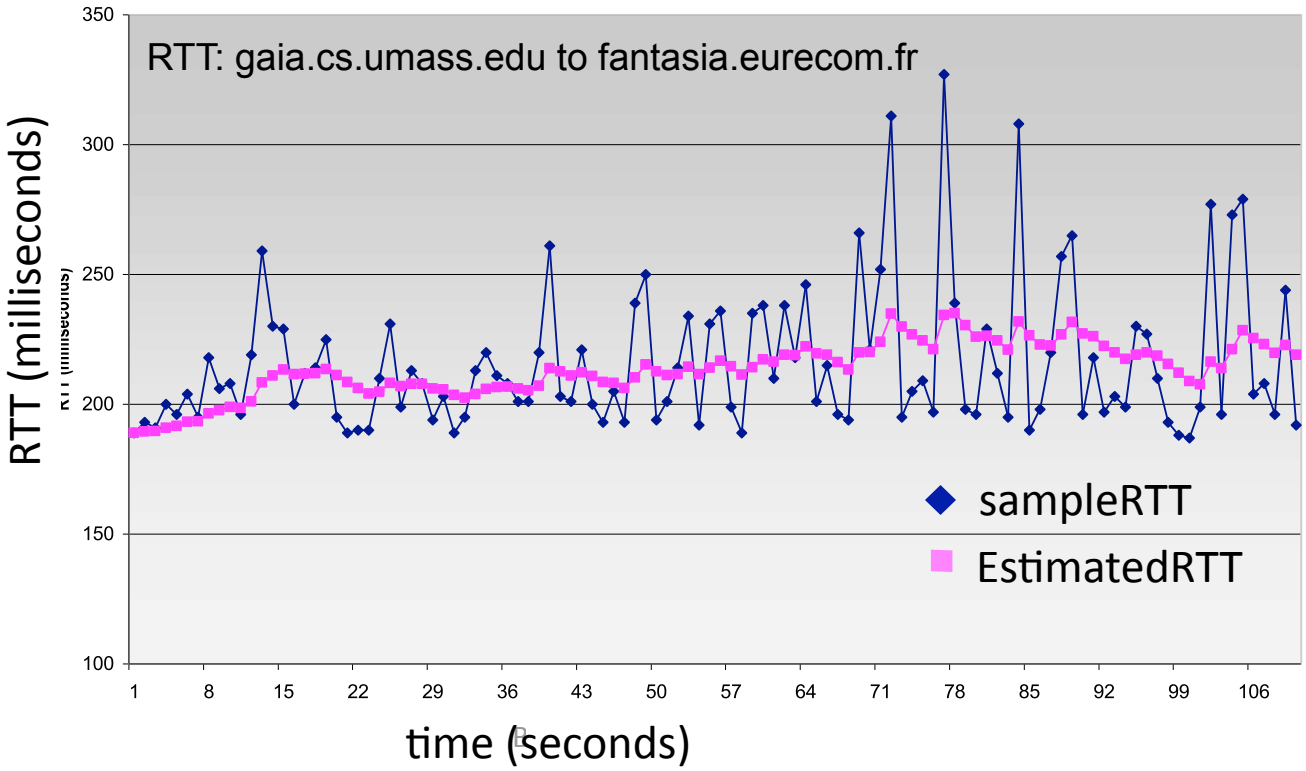
Q: how to estimate RTT?

- **`SampleRTT`**: measured time from segment transmission until ACK receipt
  - ignore retransmissions
- **`SampleRTT`** will vary, want estimated RTT "smoother"
  - average several *recent* measurements, not just current **`SampleRTT`**

TCP round trip time, timeout

EstimatedRTT = (1- α)*EstimatedRTT + α*SampleRTT

- exponential weighted moving average
- influence of past sample decreases exponentially fast
- typical value: α = 0.125



UNIVERSITY OF BIRMINGHAM

TCP round trip time, timeout

- timeout interval: **`EstimatedRTT`** plus "safety margin"
  - large variation in **`EstimatedRTT`** -> larger safety margin
- estimate SampleRTT deviation from EstimatedRTT:

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

(typically, $\beta = 0.25$)

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

estimated RTT

"safety margin"

TCP reliable data transfer

- TCP creates **reliable data service** on top of IP's unreliable service
  - pipelined segments
  - cumulative acks
  - single retransmission timer
- retransmissions triggered by:
  - timeout events
  - duplicate acks

let's consider simplified TCP sender:

- ignore duplicate acks
- ignore flow control, congestion control

TCP sender events:

Data received from app:

- create segment with seq number
- seq 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`

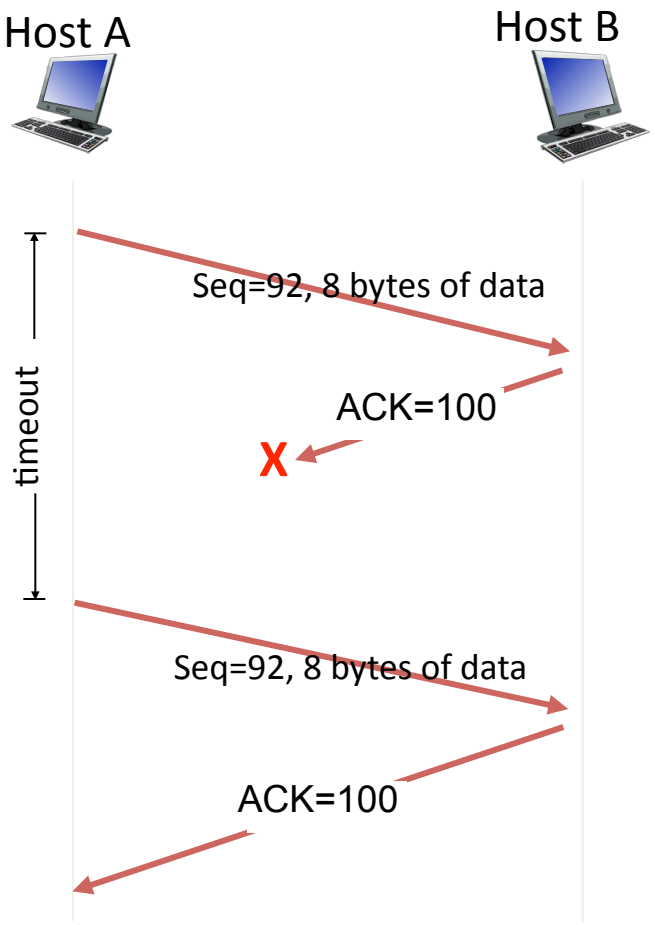
timeout:

- retransmit segment that caused timeout
- restart timer

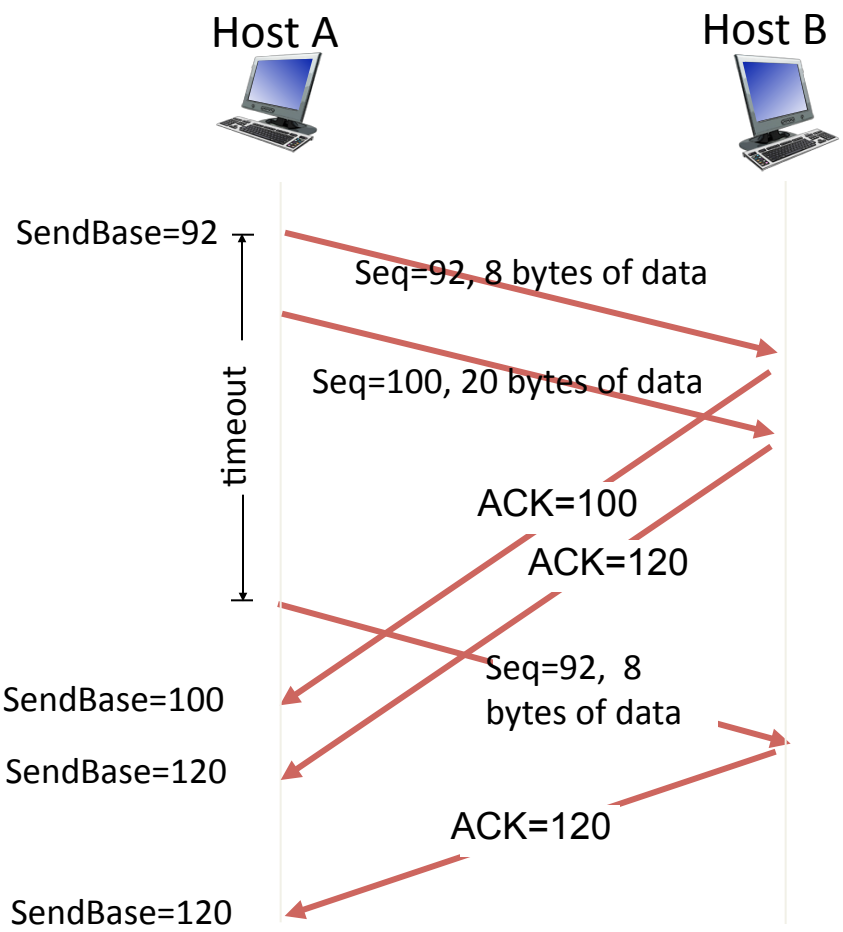
ack received:

- if ack acknowledges previously unacked segments
  - update what is known to be ACKed
  - start timer if there are still unacked segments

TCP: retransmission scenarios

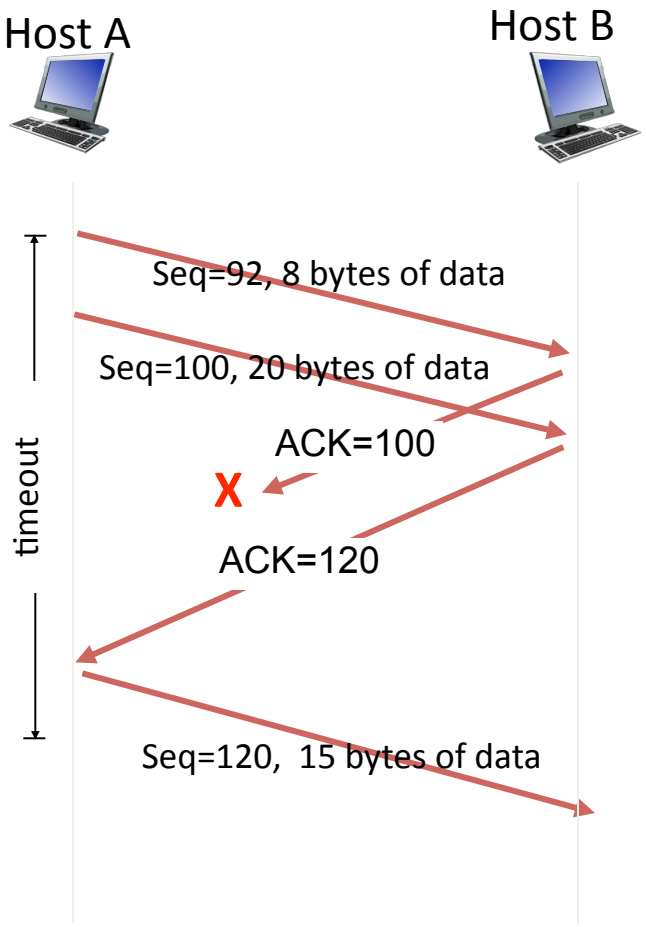


lost ACK scenario



premature timeout

TCP: retransmission scenarios

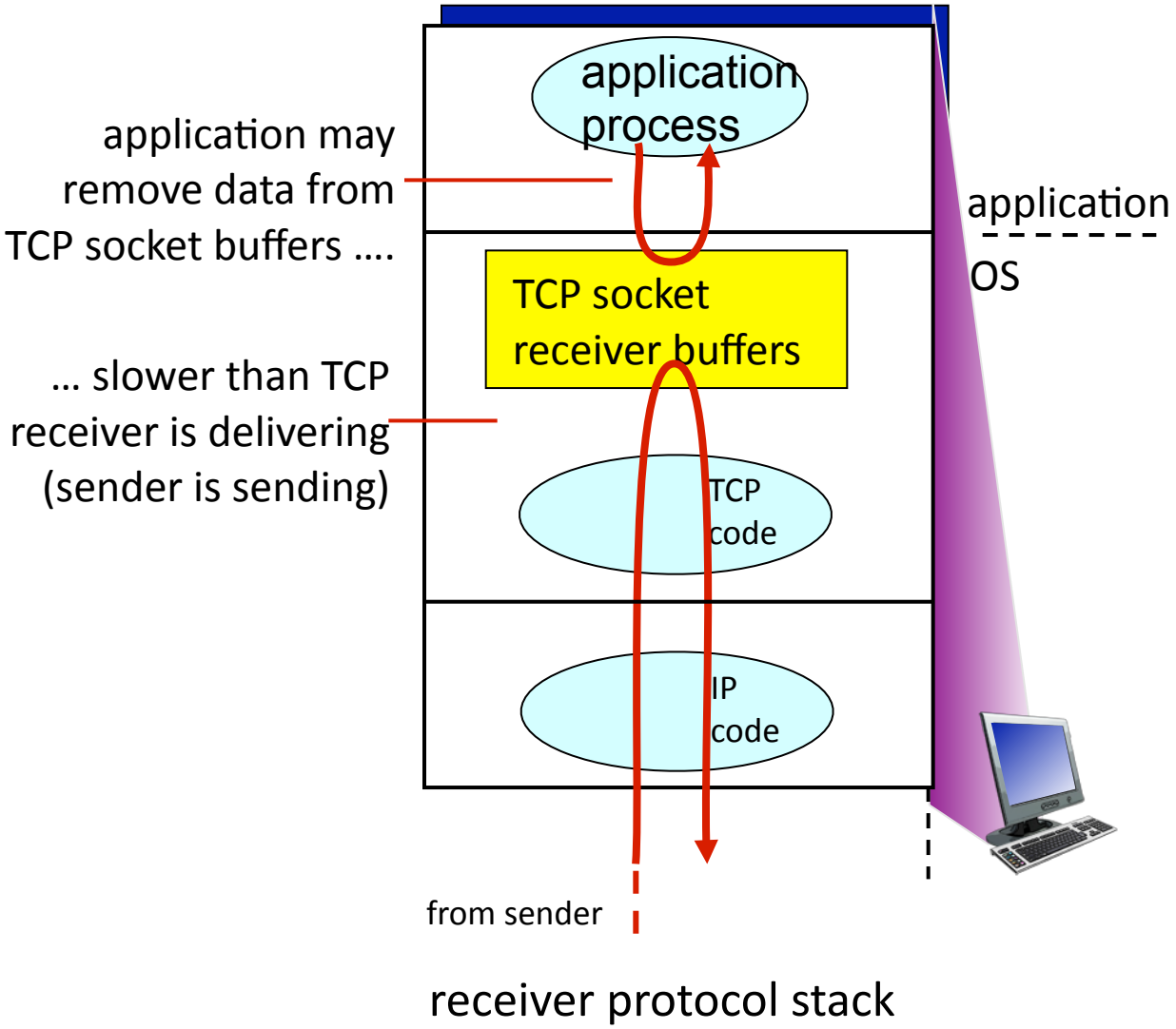


cumulative ACK

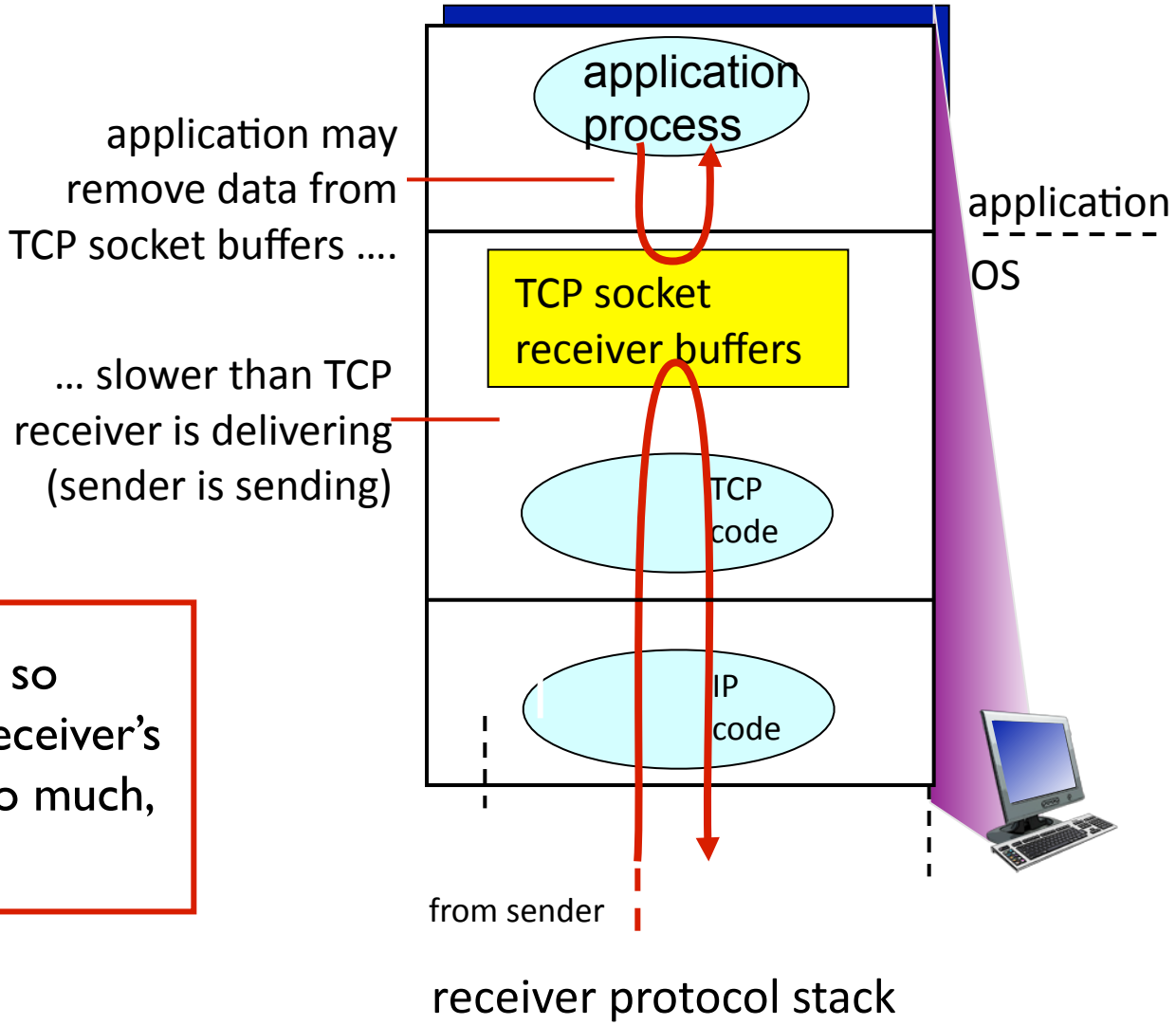
TCP ACK generation

event at receiver	*TCP receiver action*
arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
arrival of in-order segment with expected seq #. One other segment has ACK pending	immediately send single cumulative ACK, ACKing both in-order segments
arrival of out-of-order segment higher-than-expect seq. # . Gap detected	immediately send *duplicate ACK*, indicating seq. # of next expected byte
arrival of segment that partially or completely fills gap	immediate send ACK, provided that segment starts at lower end of gap

TCP flow control



TCP flow control

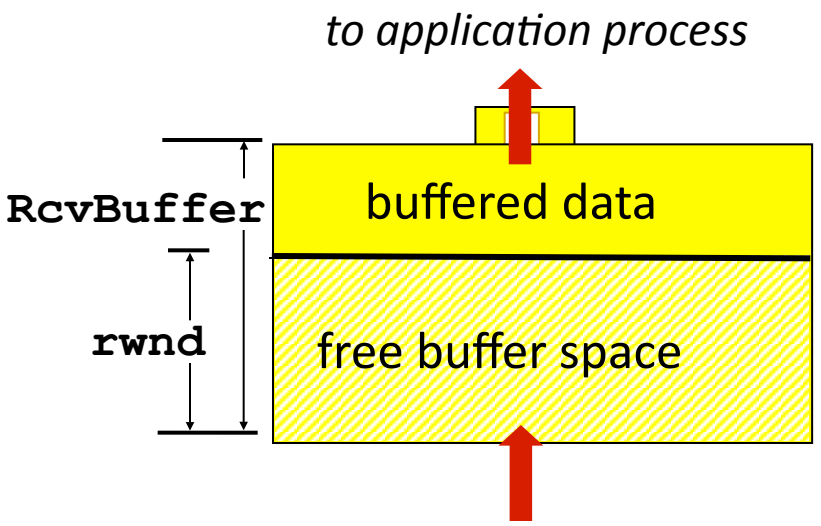


flow control

receiver controls sender, so sender won't overflow receiver's buffer by transmitting too much, too fast

TCP flow control

- receiver "advertises" free buffer space by including **`rwnd`** value in TCP header of receiver-to-sender segments
  - **`RcvBuffer`** size set via socket options (typical default is 128k bytes)
  - many operating systems autoadjust **`RcvBuffer`**
- sender limits amount of unacked ("in-flight") data to receiver's **`rwnd`** value
- guarantees receive buffer will not overflow

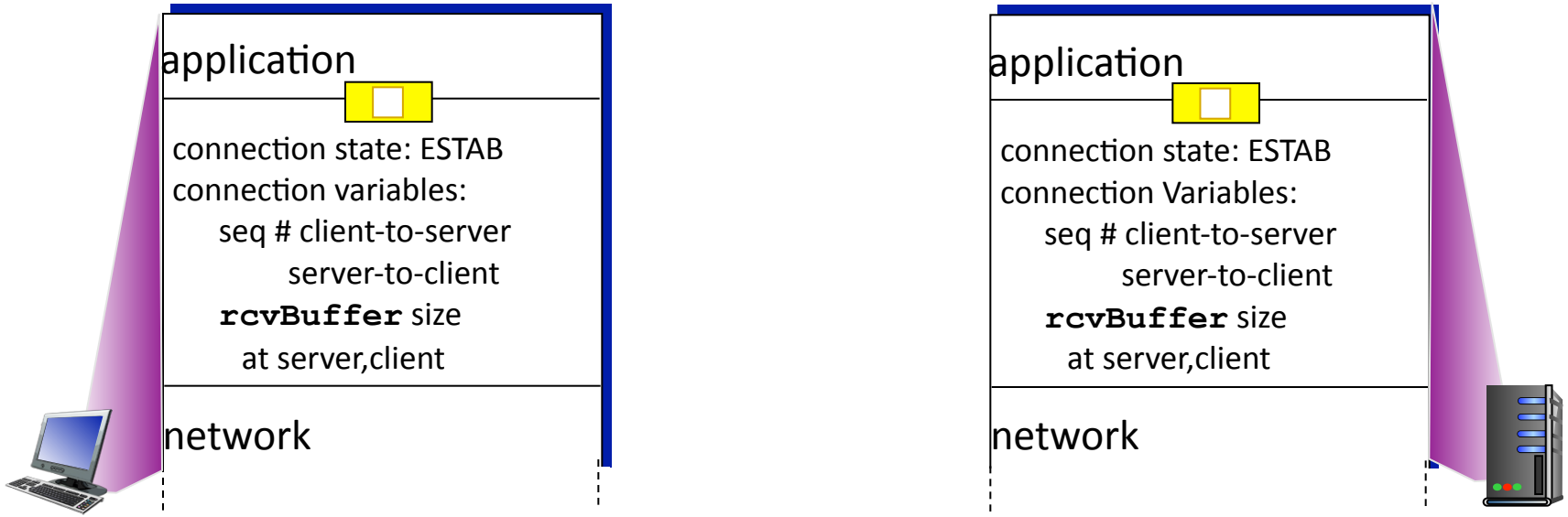


receiver-side buffering

Connection Management

before exchanging data, sender/receiver "handshake":

- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters



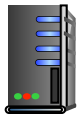

```
Socket clientSocket =
  newSocket("hostname","port
   number");
```

```
Socket connectionSocket =
   welcomeSocket.accept();
```

TCP 3-way handshake

client state

LISTEN



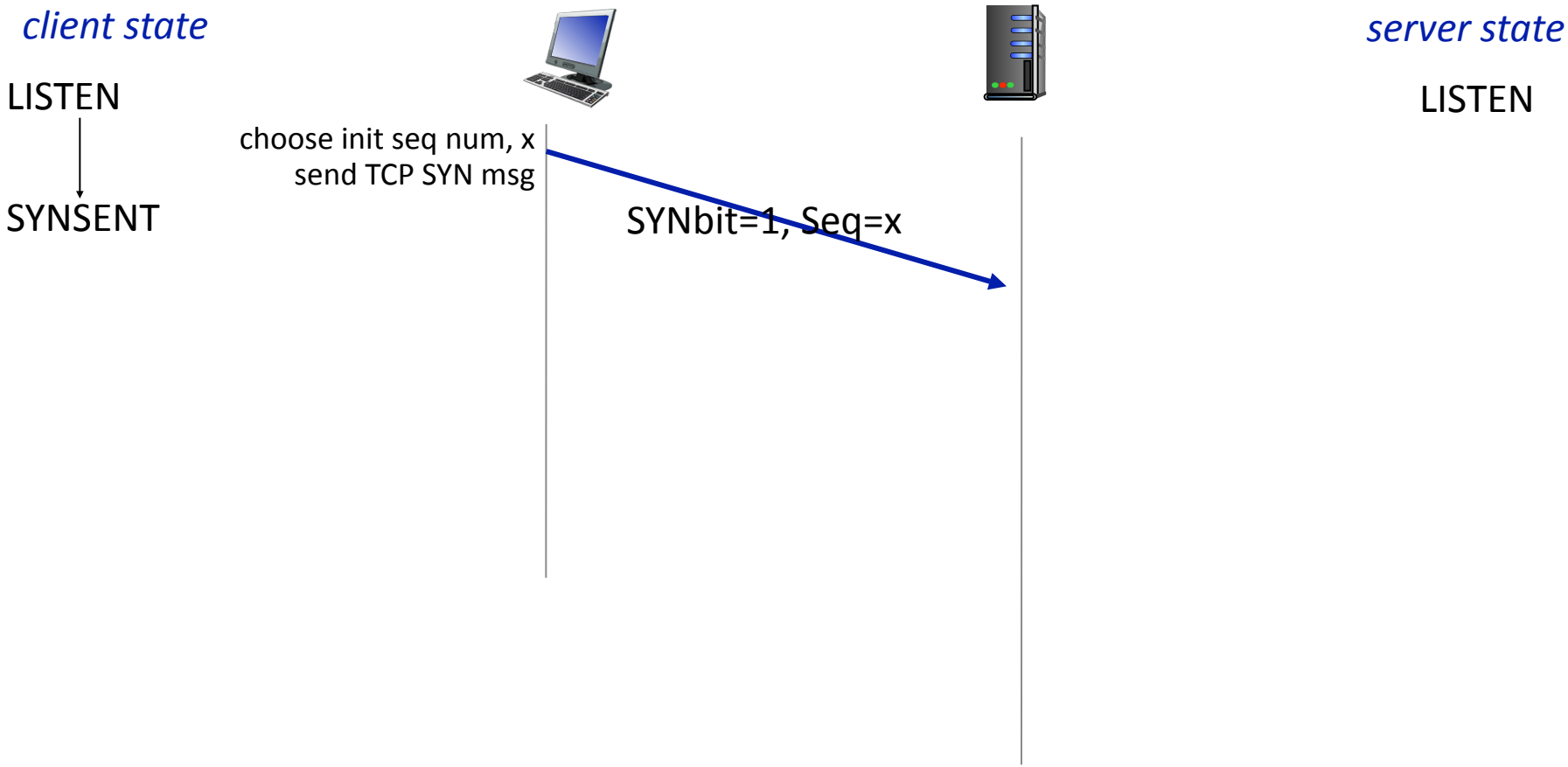
server state

LISTEN

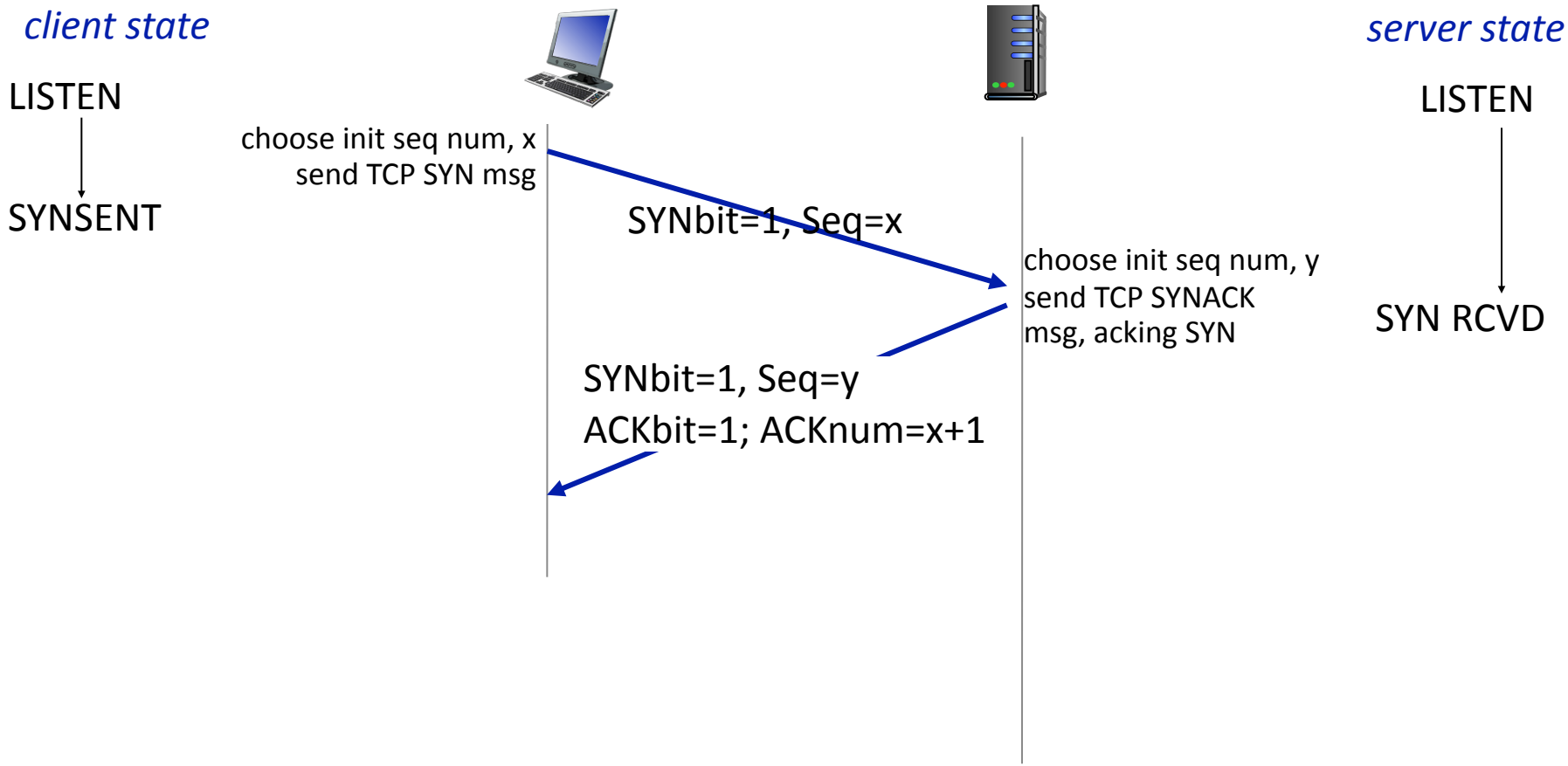
choose init seq num, x
send TCP SYN msg

SYNbit=1, Seq=x

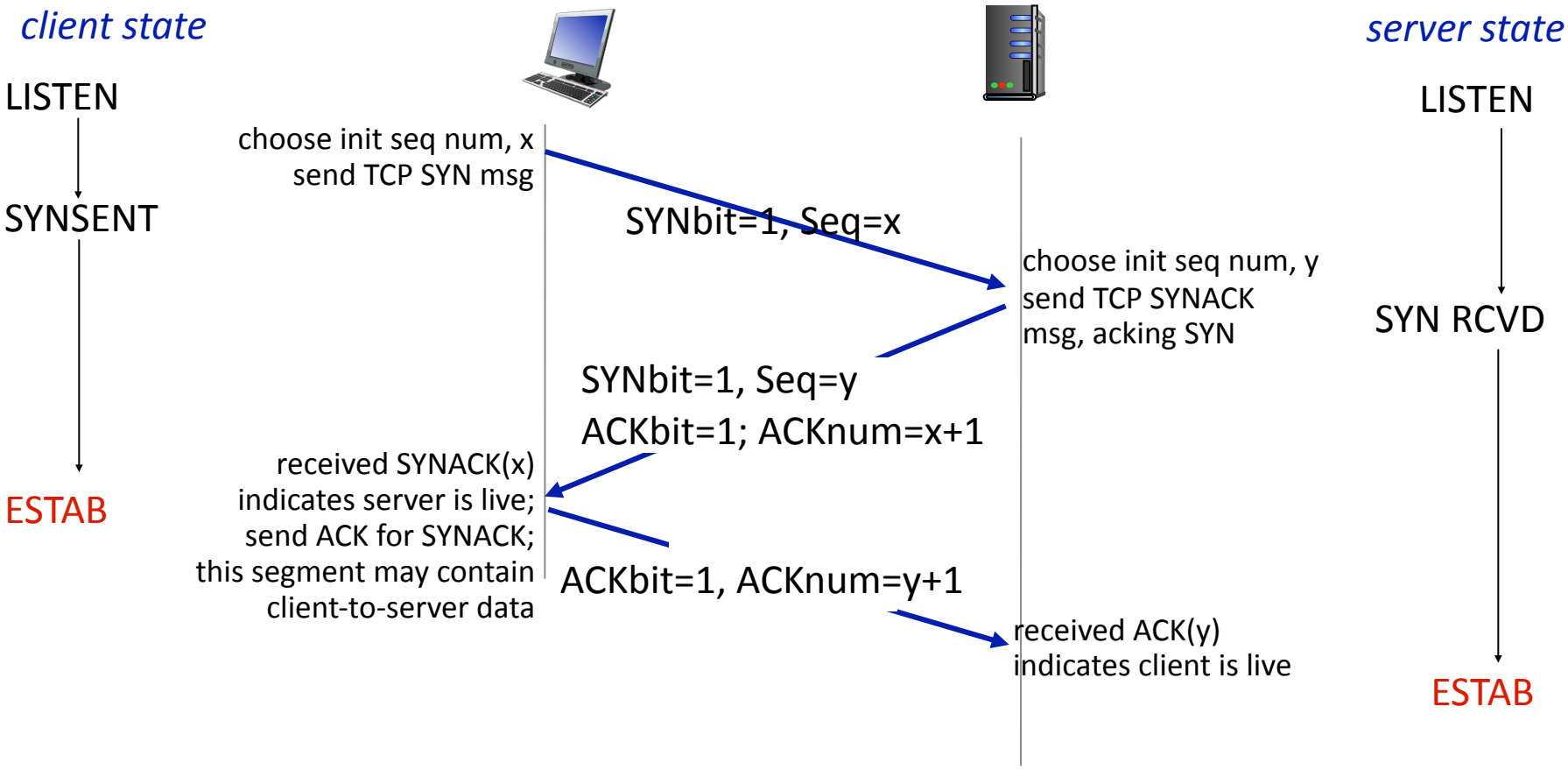
TCP 3-way handshake



TCP 3-way handshake



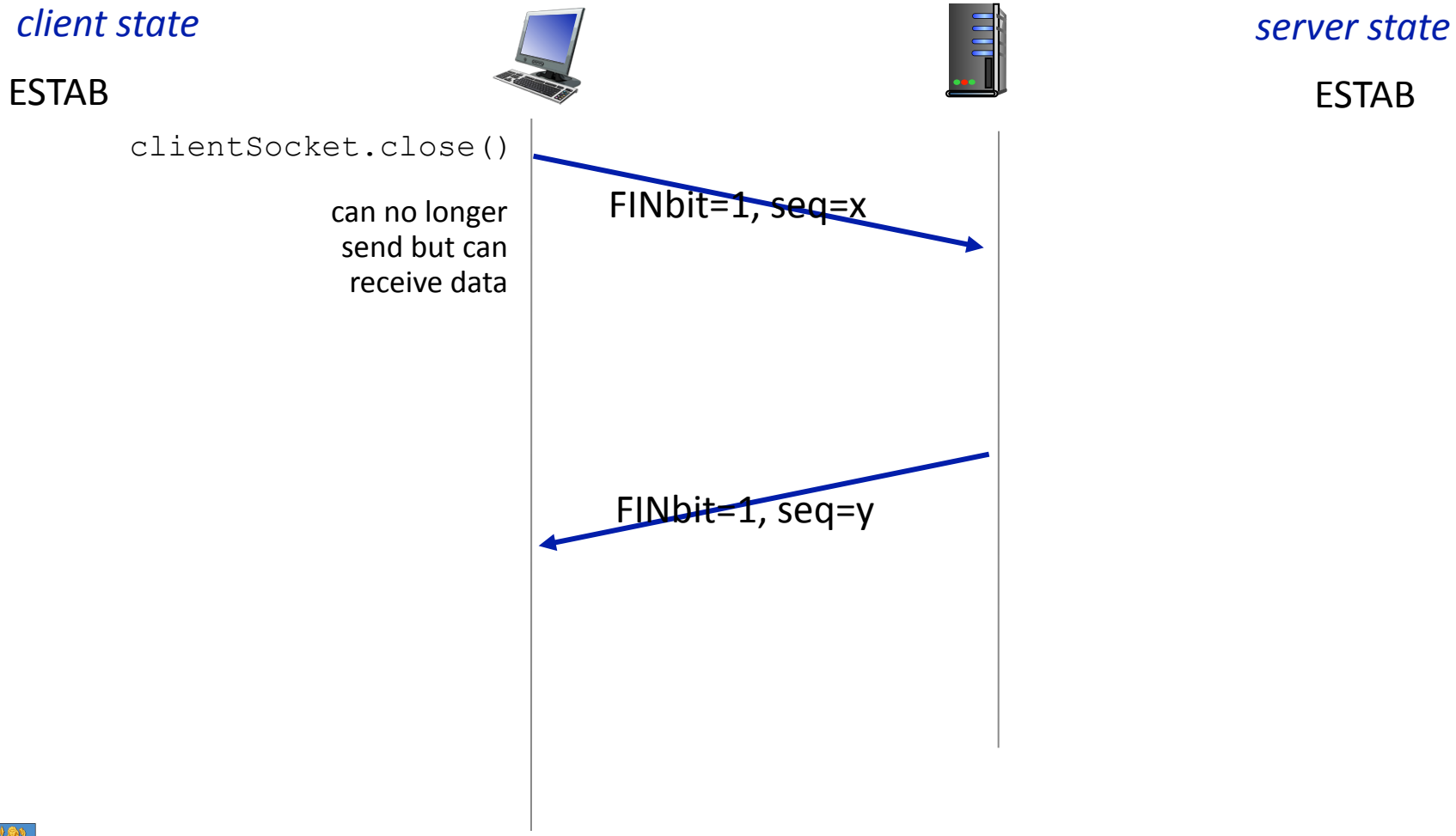
TCP 3-way handshake



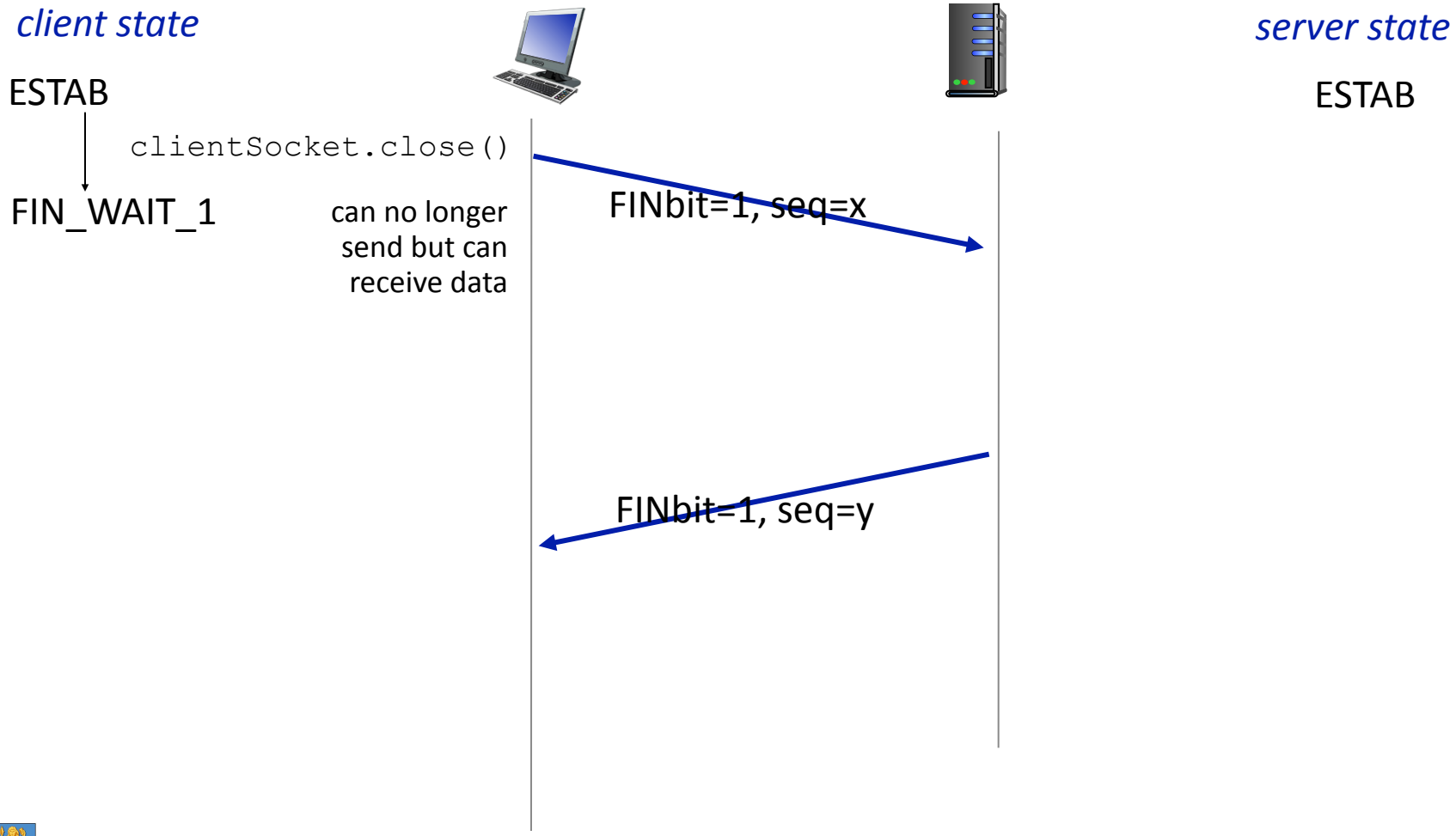
TCP: closing a connection

- client, server each close their side of connection
  - send TCP segment with FIN bit = 1
- respond to received FIN with ACK
  - on receiving FIN, ACK can be combined with own FIN
- simultaneous FIN exchanges can be handled

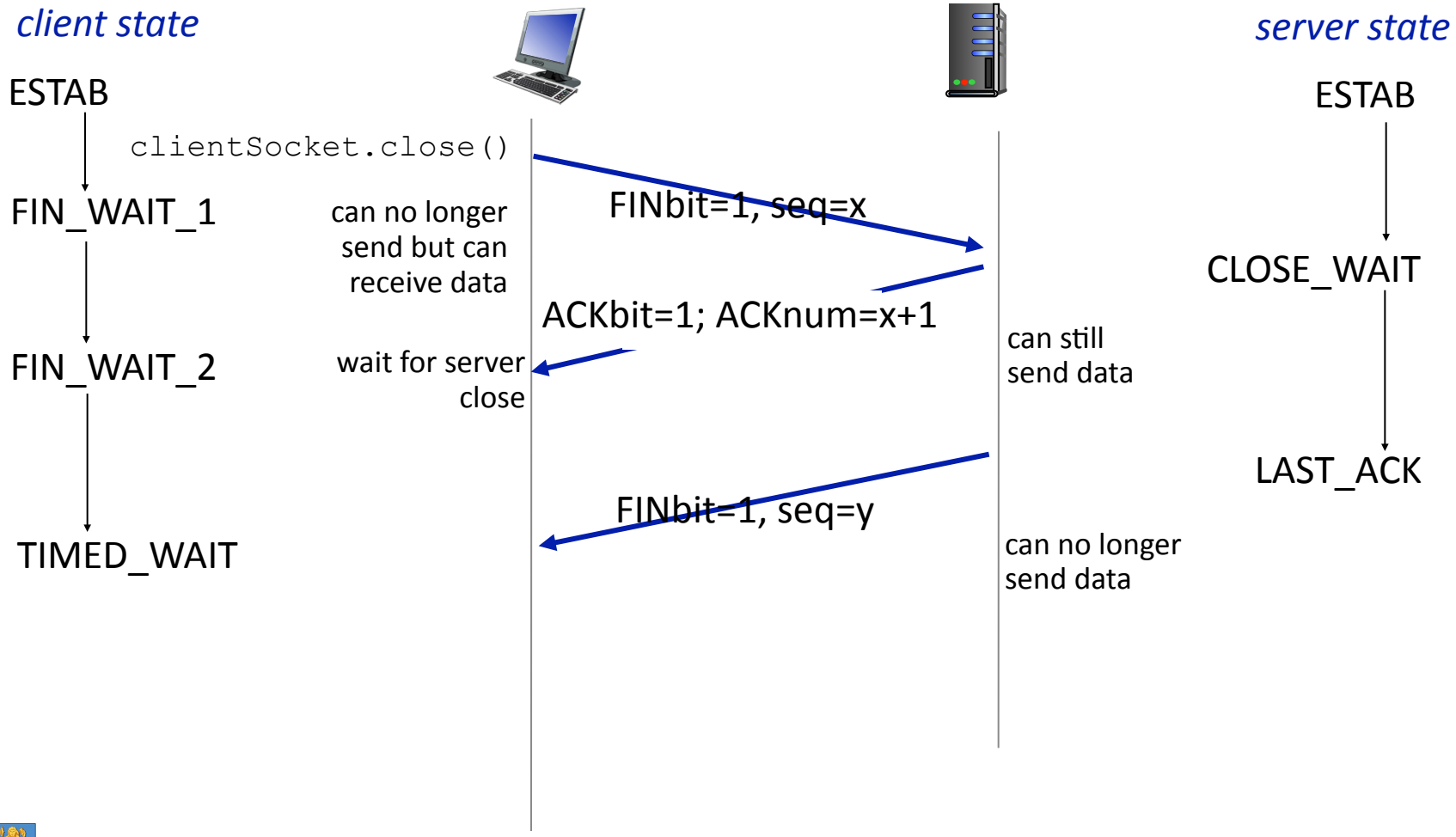
TCP: closing a connection



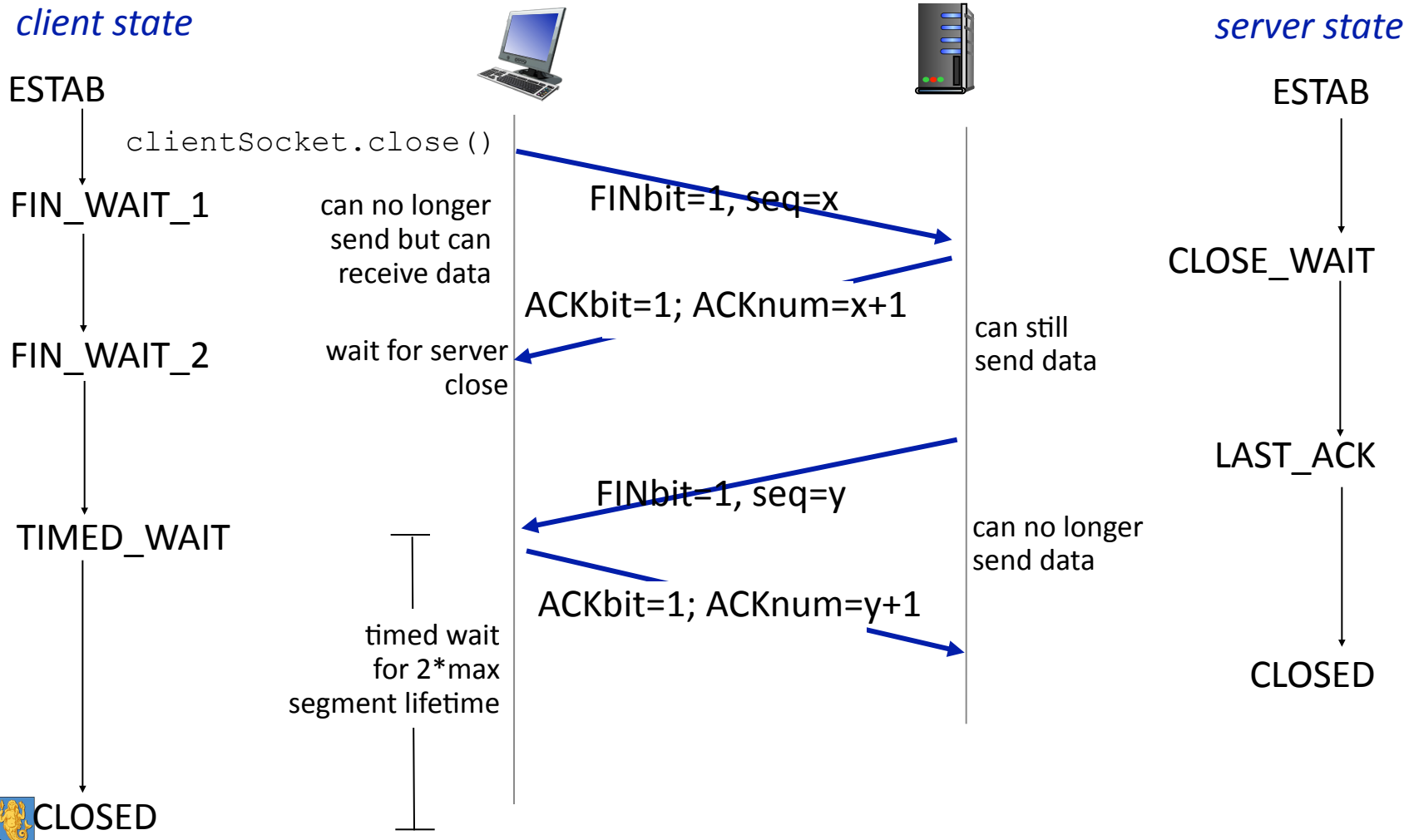
TCP: closing a connection



TCP: closing a connection



TCP: closing a connection



Summary

- Creation
  - SYN/SYN-ACK/ACK
  - Both sides now have a sequence number.
- Shut down:
  - FIN/ACK/FIN/ACK
  - Might be condensed to FIN/FIN-ACK/ACK if acks delayed and/or fast machines

- Bulk transfer
  - Receiver advertises receive window (rwnd)
  - Sender transmits until rwnd bytes unack'd
  - Receiver acks contiguous segments, plus duplicate ack for out-of-order segments
  - Sender resends all unack'd data on timeout, or receipt of duplicate ack

More TCP Issues

- Options
  - SACK enables selective acknowledgements
  - Window Scaling deals with bandwidth*delay > 64K
  - Timestamping for PAWS (protection against wrapped sequence numbers) and RTTM (round trip time measurement)
- Algorithms
  - Slow Start to fill the pipeline smoothly
  - Fast Retransmission of dropped packets
  - Rapid Recovery to avoid Slow Start when it is not necessary.

选择性确认包

Selective Acknowledgement

不是发送一个累积的ACK，而是发送一组包。

- Instead of sending a cumulative ACK, instead send a set of "upto...starting from..." pairs.
- Allows the acknowledgement to indicate that parts of the sequence space have been received. 允许ACK来指示在一个序列空间中有多少包被收到了
- Other algorithms (especially fast retransmission) have made it less useful, but it is widely supported anyway.

其他算法(尤其是快速重传)使该方法变得不那么有用，但它还是得到了广泛的支持。

窗口期伸缩术

Window Scaling

TCP“接收窗口”在每个包中都做了手脚，指出了接收方现在愿意接收多少数据。

- TCP “Receive Window” is advertised in every packet, and indicates how much data the receiver is willing to accept right now.
- It is a 16 bit quantity, so limited to 64k.

它是一个16位的量，所以限制为64k。

- At 100MBps (GigE), this means any network with RTT>655 µs, which is <100km point to point, cannot fill the pipeline (ie, will have to stop and wait for ack)
- 40GigE: 2.5km

在100兆网中,这意味着任何RTT > 655µs的网络, 也就是点对点小于100公里的话, 就不能填补管道(即,将不得不停止并等待ack)

- In reality, RTT includes time taken to generate an ack, so limit is far lower.

实际上，RTT包含生成ack所需的时间，因此限制后比你想象的要低得多。

窗口期伸缩术

Window Scaling

解决方案是协商接收窗口的缩放系数

- Solution is to negotiate a scaling factor for the receive window
增加有效的窗口大小2 ^ 30 (将窗口左移至多14位)
- Increases effective window size to 2^30 (shift window left by up to 14 bits)
- Allows bandwidth*delay up to 1GB. 允许带宽*延迟高达1GB
- Hence GigE up to 10s RTT, or 40GigE up to 250ms RTT. 因此，千兆网来回最高可达10s，或40GigE最高可达250ms 来回。

GigE = Gigabit Etherne 千兆以太网 Gb级别的
RTT = round-trip-time

PAWS

如果您有一个接近1GB的接收窗口，然后有少量的重新排序和延迟，那么您可以在序列空间中间隔4GB的包上有重复的序列号。

- If you have a receive window that is approaching 1GB then with small amounts of reordering and delay, you can have duplicate sequence numbers on packets which are 4GB apart in the sequence space.
- "Time"stamp in header is monotonically increasing, every 1ms or less. 报头中的"时间"戳单调递增，每1ms或更少。
- This allows you to distinguish between "new" and "old" packets with same sequence no. 这样就可以分辨相同序列号包的先后顺序了
- 32 bit counter running at 1000 Hz takes 50 days to wrap around!

RTTM

- Measuring RTT can be difficult in the face of retransmissions
- Instead place timestamp in header of data segments (also used for PAWS) and reflect that timestamp back in an ACK.
- Subtract the ACK timestamp from current value of counter to get RTT. Simples!

Slow Start

- In many environments, the local network is faster than the wide area network.
- So a large advertised receive window (possibly scaled) will result in the sender potentially sending large amounts of data to the local router, which then has to clock it out at a slower speed.
- Potential packet loss (limited router memory), does not make effective use of the whole path.

慢启动

Slow Start

一旦建立连接，不要一开始就发送完整的接收“窗口”

- Once connection is established, do not initially send complete receive window.
- Instead, only send one packet, and increase the number of packets sent by one for every acknowledgement received, until the window is full (or double the number of packets, or some other algorithm). 相反，只发送一个包，并在每接收到一次ACK后增加一个发送包的数量，直到窗口满为止(或将包的数量增加一倍，或其他一些算法)。
- Conceptually, this spreads out the packets in the network, and reduces congestion at intervening routers. 从概念上讲，这可以分散网络中的数据包，并减少中间路由器的拥塞。

快速恢复/快速重发

Rapid Recovery / Fast Retransmit

各种算法在各种操作系统的不同版本中实现，大多数都有“Tahoe”和“Reno”这样的名称。

- Various algorithms are implemented in various versions of various operating systems, mostly with names like “Tahoe” and “Reno”.
- Uses receipt of duplicate acks to infer that subsequent packets have been lost, and retransmits them right now rather than waiting for timeout. 使用重复ack的接收来推断后续包已经丢失，并立即重新传输它们，而不是等待超时。
- Avoids going into slow start, so that window is kept full. 避免进入慢启动，使窗口保持满。

Silly Window Syndrome

- Window is zero (receiver's buffers are full)
- Receiving application is reading one byte at a time
- Receive window is now 1 byte, so ACK sent opening window by 1
- Sender sends one more byte
- We are now using two complete TCP frames every time one byte is read

Two solutions

- Receiver: do not advertise small increases in window, but only advertise when either we can accept one more segment, or half our buffer.
- Sender ("Nagle's algorithm"): do not send small packets when there is outstanding unack'd data (more complex in reality: RFC896).

Concrete TCP

- I find it easier to understand things from real examples.
- I hope this helps.

Simple Daemon

```
my $listen = new IO::Socket::INET (LocalPort => $port,
  Proto => "tcp", Listen => 5,
  ReuseAddr => 1) or die "cannot listen $!";
warn "listening on $port";


while (my $s = $listen->accept ()) {
  my $pid = fork ();
  die "fork $!" unless defined ($pid);
  if ($pid == 0) {
     warn "connection accepted";
     $listen->close ();
     sendfiles ($s, @ARGV);
     exit (0);
  } else {
     $s->close ();
  }
}
```

sendfiles

This number matters later

```
sub sendfiles {
  my $s = shift;
  while (my $f = shift) {
     my $fh = new IO::File ("<$f") or next;
     my $buffer;
     while (my $bytes = $fh->sysread ($buffer, 8192)) {
        $s->syswrite ($buffer, $bytes) or die "cannot write $!";
     }
  }
}
```

Connection Refused

```
igb@pi-two:~$ telnet pi-one 5758
Trying 2001:8b0:129f:a90f:ba27:ebff:fe00:efe7...
Trying 10.92.213.231...
telnet: Unable to connect to remote host: Connection refused
igb@pi-two:~$
```

Connection Rejected

```
00:00:00.000000 IP6 2001:8b0:129f:a90f:ba27:ebff:fec9:acae.
41529 > 2001:8b0:129f:a90f:ba27:ebff:fe00:efe7.5758: Flags
[S], seq 1698998887, win 28000, options [mss 1400,sackOK,TS
val 19928636 ecr 0,nop,wscale 6], length 0

00:00:00.000358 IP6
2001:8b0:129f:a90f:ba27:ebff:fe00:efe7.5758 >
2001:8b0:129f:a90f:ba27:ebff:fec9:acae.41529: Flags [R.], seq
0, ack 1698998888, win 0, length 0

00:00:00.002250 IP 10.92.213.238.44781 > 10.92.213.231.5758:
Flags [S], seq 1260504038, win 29200, options [mss
1460,sackOK,TS val 19928636 ecr 0,nop,wscale 6], length 0

00:00:00.002565 IP 10.92.213.231.5758 > 10.92.213.238.44781:
Flags [R.], seq 0, ack 1260504039, win 0, length 0
```

flag = ACK

Connection Immediately Closed

```
igb@pi-two:~$ telnet pi-one 5757
Trying 2001:8b0:129f:a90f:ba27:ebff:fe00:efe7...
Trying 10.92.213.231...
Connected to pi-one.
Escape character is '^]'.
Connection closed by foreign host.
igb@pi-two:~$
```

Immediate Close

```
00:00:00.002258 IP 10.92.213.238.49560 > 10.92.213.231.5757: Flags [S],
seq 2044006020, win 29200, options [mss 1460,sackOK,TS val 19986406 ecr
0,nop,wscale 6], length 0
00:00:00.002595 IP 10.92.213.231.5757 > 10.92.213.238.49560: Flags [S.],
seq 894665383, ack 2044006021, win 28960, options [mss 1460,sackOK,TS val
19986382 ecr 19986406,nop,wscale 6], length 0
00:00:00.003596 IP 10.92.213.238.49560 > 10.92.213.231.5757: Flags [.],
ack 1, win 457, options [nop,nop,TS val 19986406 ecr 19986382], length 0
00:00:00.018316 IP 10.92.213.231.5757 > 10.92.213.238.49560: Flags [F.],
seq 1, ack 1, win 453, options [nop,nop,TS val 19986384 ecr 19986406],
length 0
00:00:00.019698 IP 10.92.213.238.49560 > 10.92.213.231.5757: Flags [F.],
seq 1, ack 2, win 457, options [nop,nop,TS val 19986408 ecr 19986384],
length 0
00:00:00.021259 IP 10.92.213.231.5757 > 10.92.213.238.49560: Flags [.],
ack 2, win 453, options [nop,nop,TS val 19986384 ecr 19986408], length 0
```

Question

- I said in previous lecture minimum TCP session SEVEN packets (SYN, SYN/ACK, ACK, FIN, ACK, FIN, ACK).
- Here there are only SIX
- Why?

Delayed Acks at Work

- After receipt of the first FIN, the caller has nothing to say, and no outstanding un-ack'd data
- So ACK is sent only when there is traffic to pass, in this case the answering FIN.
- Hence the close is FIN, FIN/ACK, ACK.
- Requires client to call close() within 500ms of receipt of FIN, with no other traffic outstanding.

Slightly Delayed Close

```
#!/usr/bin/perl -w

use warnings;
use strict;
use IO::Socket::INET;

my $s = new IO::Socket::INET (PeerAddr => "10.92.213.231:5757", Proto => "tcp") or die "$!";

sleep (2);

exit (0);
```

Typical Seven Packets

```
00:00:00.000000 IP 10.92.213.238.49583 > 10.92.213.231.5757: Flags [S], seq
1156157459, win 29200, options [mss 1460,sackOK,TS val 20078528 ecr 0,nop,wscale
6], length 0
00:00:00.000413 IP 10.92.213.231.5757 > 10.92.213.238.49583: Flags [S.], seq
1726094991, ack 1156157460, win 28960, options [mss 1460,sackOK,TS val 20078505 ecr
20078528,nop,wscale 6], length 0
00:00:00.001393 IP 10.92.213.238.49583 > 10.92.213.231.5757: Flags [.], ack 1, win
457, options [nop,nop,TS val 20078528 ecr 20078505], length 0
// 457*2^6 = 29248
00:00:00.014287 IP 10.92.213.231.5757 > 10.92.213.238.49583: Flags [F.], seq 1, ack
1, win 453, options [nop,nop,TS val 20078506 ecr 20078528], length 0
00:00:00.023675 IP 10.92.213.238.49583 > 10.92.213.231.5757: Flags [.], ack 2, win
457, options [nop,nop,TS val 20078531 ecr 20078506], length 0
// TWO SECOND DELAY
00:00:02.002245 IP 10.92.213.238.49583 > 10.92.213.231.5757: Flags [F.], seq 1, ack
2, win 457, options [nop,nop,TS val 20078728 ecr 20078506], length 0
00:00:02.002556 IP 10.92.213.231.5757 > 10.92.213.238.49583: Flags [.], ack 2, win
453, options [nop,nop,TS val 20078705 ecr 20078728], length 0
```

Window Scaling

- Note Window Scale has been set to 6

```
options [mss 1460,sackOK,TS val 19986406 ecr 0,nop,wscale 6]
```

- Both implementations are the same (Linux, pretty much this week's kernel)
- Win is being sent as ~500, 500 * 2^6 = ~32000, which is a typical value for a Receive Window

Other Systems Are Different

- Solaris requires more provocation to select Window Scaling, especially on local area network
- Can be Asymmetric

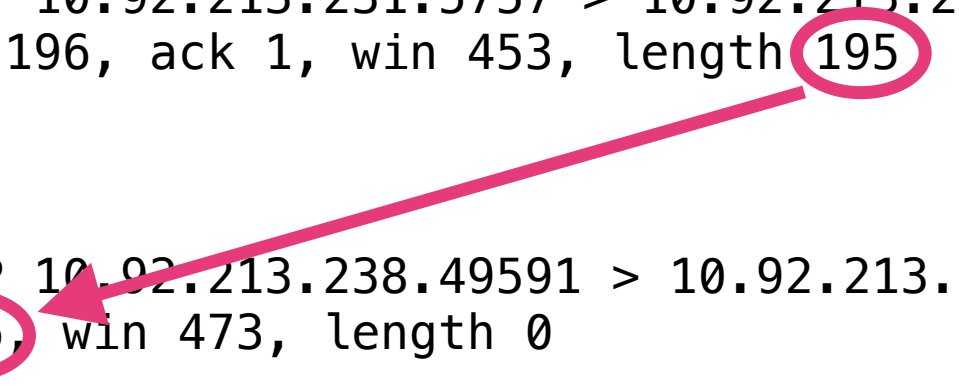
```
00:07:51.826744 IP 10.92.213.241.44205 > 10.92.213.231.5757: Flags [S], seq 87151905, win 64240,
options [mss 1460,sackOK,TS val 19039311 ecr 0,nop,wscale 1], length 0
00:07:51.827087 IP 10.92.213.231.5757 > 10.92.213.241.44205: Flags [S.], seq 3263477770, ack
87151906, win 28960, options [mss 1460,sackOK,TS val 20196188 ecr 19039311,nop,wscale 6],
length 0
```

Bulk Data

```
igb@pi-two:~$ telnet pi-one 5757
Trying 2001:8b0:129f:a90f:ba27:ebff:fe00:efe7...
Trying 10.92.213.231...
Connected to pi-one.
Escape character is '^]'.
pi-one.home.batten.eu.org, DNS, DHCP, NTP (GPS), heyu, odds and ends.

*** This is a private machine.  If you are not personally authorised by
*** igb@batten.eu.org then your access is illegal.

Connection closed by foreign host.
igb@pi-two:~$
```

Sending small amounts of data

```
SYN SYN/ACK ACK as usual, options deleted for clarity

Data Packet:

00:03:11.671436 IP 10.92.213.231.5757 > 10.92.213.238.49591:
Flags [P.], seq 1:196, ack 1, win 453, length 195

Ack:

00:03:11.672488 IP 10.92.213.238.49591 > 10.92.213.231.5757:
Flags [.], ack 196, win 473, length 0

Start of shut down:

00:03:11.675722 IP 10.92.213.231.5757 > 10.92.213.238.49591:
Flags [F.], seq 196, ack 1, win 453, length 0
```



Slightly more data

- Notice mss is 1460, so anything more than 1400 bytes will require more than one packet.
- Let's try 2KB

```
igb@pi-one:~$ i=1;while [[ $i -lt 2048 ]]; do echo -n 0; let i=i+1; done > 2kfile
igb@pi-one:~$
```

```
igb@pi-two:~$ telnet pi-one 5757
Trying 2001:8b0:129f:a90f:ba27:ebff:fe00:efe7...
Trying 10.92.213.231...
Connected to pi-one.
Escape character is '^]'.
00000000…0Connection closed by foreign host.
igb@pi-two:~$
```

```
// Note caller is now Solaris, wscale 1, options deleted
00:00:00.002792 IP 10.92.213.241.63887 > 10.92.213.231.5757:
Flags [.], ack 1, win 64436, length 0
00:00:00.021531 IP 10.92.213.231.5757 > 10.92.213.241.63887:
Flags [.], seq 1:1449, ack 1, win 453, length 1448
00:00:00.022776 IP 10.92.213.241.63887 > 10.92.213.231.5757:
Flags [.], ack 1449, win 64436, length 0
00:00:00.024380 IP 10.92.213.231.5757 > 10.92.213.241.63887:
Flags [P.], seq 1449:2048, ack 1, win 453, length 599
00:00:00.025245 IP 10.92.213.241.63887 > 10.92.213.231.5757:
Flags [.], ack 2048, win 64436, length 0
00:00:00.027568 IP 10.92.213.231.5757 > 10.92.213.241.63887:
Flags [F.], seq 2048, ack 1, win 453, length 0
00:00:00.028577 IP 10.92.213.241.63887 > 10.92.213.231.5757:
Flags [.], ack 2049, win 64436, length 0
00:00:00.028884 IP 10.92.213.241.63887 > 10.92.213.231.5757:
Flags [F.], seq 1, ack 2049, win 64436, length 0
```

Slightly more data

```
igb@pi-one:~$ i=0;while [[ $i -lt 2048 ]]; do echo 0123456789ABCDE; let i=i+1; done > 32kfile
igb@pi-one:~$ ls -l 32kfile
-rw-r--r-- 1 igb igb 32768 Feb 12 06:54 32kfile
igb@pi-one:~$
```

```
igb@zone-host:~$ more /tmp/qqq
Trying 2001:8b0:129f:a90f:ba27:ebff:fe00:efe7...
telnet: connect to address 2001:8b0:129f:a90f:ba27:ebff:fe00:efe7: Connection refused
Trying 10.92.213.231...
Connected to pi-one.home.batten.eu.org.
Escape character is '^]'.
0123456789ABCDE
0123456789ABCDE
0123456789ABCDE
0123456789ABCDE
0123456789ABCDE
…
0123456789ABCDE
0123456789ABCDE
Connection to pi-one.home.batten.eu.org closed by foreign host.
igb@zone-host:~$
```

```
00:00:00.001631 IP 10.92.213.241.54141 > 10.92.213.231.5757: tcp 0
00:00:00.001990 IP 10.92.213.231.5757 > 10.92.213.241.54141: tcp 0
00:00:00.002933 IP 10.92.213.241.54141 > 10.92.213.231.5757: tcp 0
00:00:00.020559 IP 10.92.213.231.5757 > 10.92.213.241.54141: tcp 1448
00:00:00.021842 IP 10.92.213.241.54141 > 10.92.213.231.5757: tcp 0
00:00:00.023312 IP 10.92.213.231.5757 > 10.92.213.241.54141: tcp 1448
00:00:00.024515 IP 10.92.213.241.54141 > 10.92.213.231.5757: tcp 0
00:00:00.026147 IP 10.92.213.231.5757 > 10.92.213.241.54141: tcp 1448
00:00:00.027936 IP 10.92.213.231.5757 > 10.92.213.241.54141: tcp 1448
00:00:00.029063 IP 10.92.213.241.54141 > 10.92.213.231.5757: tcp 0
00:00:00.030215 IP 10.92.213.231.5757 > 10.92.213.241.54141: tcp 1448
00:00:00.030595 IP 10.92.213.231.5757 > 10.92.213.241.54141: tcp 952
00:00:00.031462 IP 10.92.213.231.5757 > 10.92.213.241.54141: tcp 1448
00:00:00.032635 IP 10.92.213.241.54141 > 10.92.213.231.5757: tcp 0
00:00:00.034243 IP 10.92.213.231.5757 > 10.92.213.241.54141: tcp 1448
00:00:00.036050 IP 10.92.213.231.5757 > 10.92.213.241.54141: tcp 1448
00:00:00.036384 IP 10.92.213.231.5757 > 10.92.213.241.54141: tcp 1448
00:00:00.036678 IP 10.92.213.231.5757 > 10.92.213.241.54141: tcp 1448
00:00:00.037042 IP 10.92.213.231.5757 > 10.92.213.241.54141: tcp 952
00:00:00.037689 IP 10.92.213.241.54141 > 10.92.213.231.5757: tcp 0
00:00:00.038255 IP 10.92.213.231.5757 > 10.92.213.241.54141: tcp 1448
00:00:00.038591 IP 10.92.213.231.5757 > 10.92.213.241.54141: tcp 1448
00:00:00.038871 IP 10.92.213.231.5757 > 10.92.213.241.54141: tcp 1448
00:00:00.039169 IP 10.92.213.231.5757 > 10.92.213.241.54141: tcp 1448
00:00:00.039455 IP 10.92.213.231.5757 > 10.92.213.241.54141: tcp 1448
00:00:00.039982 IP 10.92.213.231.5757 > 10.92.213.241.54141: tcp 1448
00:00:00.040343 IP 10.92.213.241.54141 > 10.92.213.231.5757: tcp 0
00:00:00.040604 IP 10.92.213.231.5757 > 10.92.213.241.54141: tcp 1448
00:00:00.040922 IP 10.92.213.231.5757 > 10.92.213.241.54141: tcp 1448
00:00:00.045585 IP 10.92.213.231.5757 > 10.92.213.241.54141: tcp 1448
00:00:00.046607 IP 10.92.213.231.5757 > 10.92.213.241.54141: tcp 1448
00:00:00.047684 IP 10.92.213.241.54141 > 10.92.213.231.5757: tcp 0
00:00:00.049154 IP 10.92.213.231.5757 > 10.92.213.241.54141: tcp 1448
00:00:00.050696 IP 10.92.213.231.5757 > 10.92.213.241.54141: tcp 456
00:00:00.054361 IP 10.92.213.231.5757 > 10.92.213.241.54141: tcp 0
00:00:00.055249 IP 10.92.213.241.54141 > 10.92.213.231.5757: tcp 0
```

increase the pack of send,
no need to ack for each packet

UNIVERSITY OF BIRMINGHAM

Slow Start

- Note behaviour of slow start
- Sender initially only has small amounts of unack'd data, but progressively speeds up by sending more and more packets "back to back"

What's going on?

- What are those 952 byte packets? 952? Where does that come from?
- 1448 * 5 + 952 = 8192, the blocks I'm calling the API with.
- When in slow start, there's enough delay on proceedings that the blockings matter.
- Once the transfer really gets going, they're aggregated.
- 1448 * 11 + 456 = 8192 * 2: two blocks

Edit daemon…

- Make the block size I'm sending from userland be 64k instead (ie, all of 32k file sent in one write())

```
00:00:00.021483 IP 10.92.213.231.5757 > 10.92.213.241.57935: tcp 1448
00:00:00.022790 IP 10.92.213.241.57935 > 10.92.213.231.5757: tcp 0
00:00:00.024287 IP 10.92.213.231.5757 > 10.92.213.241.57935: tcp 1448
00:00:00.025551 IP 10.92.213.241.57935 > 10.92.213.231.5757: tcp 0
00:00:00.027213 IP 10.92.213.231.5757 > 10.92.213.241.57935: tcp 1448
00:00:00.028780 IP 10.92.213.231.5757 > 10.92.213.241.57935: tcp 1448
00:00:00.030097 IP 10.92.213.241.57935 > 10.92.213.231.5757: tcp 0
00:00:00.031315 IP 10.92.213.231.5757 > 10.92.213.241.57935: tcp 1448
00:00:00.031664 IP 10.92.213.231.5757 > 10.92.213.241.57935: tcp 1448
00:00:00.031967 IP 10.92.213.231.5757 > 10.92.213.241.57935: tcp 1448
00:00:00.032257 IP 10.92.213.231.5757 > 10.92.213.241.57935: tcp 1448
00:00:00.032536 IP 10.92.213.231.5757 > 10.92.213.241.57935: tcp 1448
00:00:00.032837 IP 10.92.213.231.5757 > 10.92.213.241.57935: tcp 1448
00:00:00.033189 IP 10.92.213.241.57935 > 10.92.213.231.5757: tcp 0
00:00:00.033575 IP 10.92.213.231.5757 > 10.92.213.241.57935: tcp 1448
00:00:00.033799 IP 10.92.213.231.5757 > 10.92.213.241.57935: tcp 1448
00:00:00.034122 IP 10.92.213.231.5757 > 10.92.213.241.57935: tcp 1448
00:00:00.034340 IP 10.92.213.231.5757 > 10.92.213.241.57935: tcp 1448
00:00:00.034686 IP 10.92.213.231.5757 > 10.92.213.241.57935: tcp 1448
00:00:00.034897 IP 10.92.213.231.5757 > 10.92.213.241.57935: tcp 1448
00:00:00.035255 IP 10.92.213.231.5757 > 10.92.213.241.57935: tcp 1448
00:00:00.035473 IP 10.92.213.231.5757 > 10.92.213.241.57935: tcp 1448
00:00:00.035805 IP 10.92.213.231.5757 > 10.92.213.241.57935: tcp 1448
00:00:00.036099 IP 10.92.213.231.5757 > 10.92.213.241.57935: tcp 1448
00:00:00.036569 IP 10.92.213.241.57935 > 10.92.213.231.5757: tcp 0
00:00:00.036579 IP 10.92.213.241.57935 > 10.92.213.231.5757: tcp 0
00:00:00.041608 IP 10.92.213.231.5757 > 10.92.213.241.57935: tcp 1448
00:00:00.042676 IP 10.92.213.231.5757 > 10.92.213.241.57935: tcp 1448
00:00:00.043686 IP 10.92.213.241.57935 > 10.92.213.231.5757: tcp 0
00:00:00.044757 IP 10.92.213.231.5757 > 10.92.213.241.57935: tcp 912
00:00:00.047246 IP 10.92.213.231.5757 > 10.92.213.241.57935: tcp 0
00:00:00.048122 IP 10.92.213.241.57935 > 10.92.213.231.5757: tcp 0
```

Programming Tip

- For maximum performance, send the largest possible blocks into the kernel
- With 8192 byte writes, 30ms to send 32768 bytes
- With 65536 byte writes, ie "file at a file", 23ms
- 23% performance improvement by changing one constant!
- If available, use "sendfile()" system call

1M file, steady state

```
00:00:00.299366 IP 10.92.213.231.5757 > 10.92.213.241.53681: tcp 1448
00:00:00.299706 IP 10.92.213.231.5757 > 10.92.213.241.53681: tcp 1448
00:00:00.299997 IP 10.92.213.231.5757 > 10.92.213.241.53681: tcp 1448
00:00:00.300314 IP 10.92.213.241.53681 > 10.92.213.231.5757: tcp 0
00:00:00.300575 IP 10.92.213.231.5757 > 10.92.213.241.53681: tcp 1448
00:00:00.300854 IP 10.92.213.231.5757 > 10.92.213.241.53681: tcp 1448
00:00:00.301345 IP 10.92.213.231.5757 > 10.92.213.241.53681: tcp 1448
00:00:00.301584 IP 10.92.213.231.5757 > 10.92.213.241.53681: tcp 1448
00:00:00.301843 IP 10.92.213.231.5757 > 10.92.213.241.53681: tcp 1448
00:00:00.302088 IP 10.92.213.231.5757 > 10.92.213.241.53681: tcp 1448
00:00:00.302330 IP 10.92.213.231.5757 > 10.92.213.241.53681: tcp 1448
00:00:00.302560 IP 10.92.213.231.5757 > 10.92.213.241.53681: tcp 1448
00:00:00.302797 IP 10.92.213.231.5757 > 10.92.213.241.53681: tcp 1448
00:00:00.303115 IP 10.92.213.231.5757 > 10.92.213.241.53681: tcp 1448
00:00:00.303477 IP 10.92.213.241.53681 > 10.92.213.231.5757: tcp 0
00:00:00.303758 IP 10.92.213.231.5757 > 10.92.213.241.53681: tcp 1448
00:00:00.304058 IP 10.92.213.231.5757 > 10.92.213.241.53681: tcp 1448
00:00:00.304314 IP 10.92.213.231.5757 > 10.92.213.241.53681: tcp 1448
00:00:00.304555 IP 10.92.213.231.5757 > 10.92.213.241.53681: tcp 1448
00:00:00.305517 IP 10.92.213.241.53681 > 10.92.213.231.5757: tcp 0
00:00:00.306582 IP 10.92.213.231.5757 > 10.92.213.241.53681: tcp 1448
00:00:00.307198 IP 10.92.213.231.5757 > 10.92.213.241.53681: tcp 1448
00:00:00.307785 IP 10.92.213.231.5757 > 10.92.213.241.53681: tcp 1448
```

Acks are trailing data

```
00:00:00.624882 IP 10.92.213.231.5757 > 10.92.213.241.34058: Flags [.], seq 988985:990433, ack 1,
win 453, length 1448
00:00:00.625125 IP 10.92.213.231.5757 > 10.92.213.241.34058: Flags [P.], seq 990433:991881, ack
1, win 453, length 1448
00:00:00.625469 IP 10.92.213.231.5757 > 10.92.213.241.34058: Flags [.], seq 991881:993329, ack 1,
win 453, length 1448
00:00:00.625747 IP 10.92.213.231.5757 > 10.92.213.241.34058: Flags [.], seq 993329:994777, ack 1,
win 453, length 1448
00:00:00.626078 IP 10.92.213.241.34058 > 10.92.213.231.5757: Flags [.], ack 991881, win 64436,
length 0
00:00:00.626310 IP 10.92.213.231.5757 > 10.92.213.241.34058: Flags [.], seq 994777:996225, ack 1,
win 453, length 1448
00:00:00.626575 IP 10.92.213.231.5757 > 10.92.213.241.34058: Flags [.], seq 996225:997673, ack 1,
win 453, length 1448
00:00:00.627065 IP 10.92.213.231.5757 > 10.92.213.241.34058: Flags [.], seq 997673:999121, ack 1,
win 453, length 1448
```

What about broken networks?

technique

```
# use a very, very rubbish transmission protocol

sub sendfiles {
  my $s = shift;
  while (my $f = shift) {
     my $fh = new IO::File ("<$f") or next;
     my $buffer;
     while (my $bytes = $fh->sysread ($buffer, 8)) {
        $s->syswrite ($buffer, $bytes) or die "cannot write $!";
   sleep (1);
     }
  }
}
```

```
00:00:00.020080 IP 10.92.213.231.5757 > 10.92.213.238.49778: Flags [P.], seq
1:9, ack 1, win 453, options [nop,nop,TS val 21020214 ecr 21020236], length 8
00:00:00.021062 IP 10.92.213.238.49778 > 10.92.213.231.5757: Flags [.], ack 9,
win 457, options [nop,nop,TS val 21020238 ecr 21020214], length 0
00:00:01.023658 IP 10.92.213.231.5757 > 10.92.213.238.49778: Flags [P.], seq
9:17, ack 1, win 453, options [nop,nop,TS val 21020314 ecr 21020238], length 8
00:00:01.024786 IP 10.92.213.238.49778 > 10.92.213.231.5757: Flags [.], ack
17, win 457, options [nop,nop,TS val 21020338 ecr 21020314], length 0
00:00:02.027197 IP 10.92.213.231.5757 > 10.92.213.238.49778: Flags [P.], seq
17:25, ack 1, win 453, options [nop,nop,TS val 21020415 ecr 21020338], length
8
00:00:02.028213 IP 10.92.213.238.49778 > 10.92.213.231.5757: Flags [.], ack
25, win 457, options [nop,nop,TS val 21020439 ecr 21020415], length 0
00:00:03.030555 IP 10.92.213.231.5757 > 10.92.213.238.49778: Flags [P.], seq
25:33, ack 1, win 453, options [nop,nop,TS val 21020515 ecr 21020439], length
8
00:00:03.031564 IP 10.92.213.238.49778 > 10.92.213.231.5757: Flags [.], ack
33, win 457, options [nop,nop,TS val 21020539 ecr 21020515], length 0
```

We can cause Linux to lose packets

```
igb@pi-one:~$ sudo iptables -I OUTPUT -p tcp -m tcp --sport 5757 -j DROP
```

```
igb@pi-one:~$ sudo iptables --delete OUTPUT 1
```

Note that the dropped packets don't appear in tcpdump output when tcpdump is run on the local machine; if you need that, you need to monitor at the other end.

What happened?

```
00:00:02.025408 IP 10.92.213.231.5757 > 10.92.213.238.49784: Flags [P.], seq 17:25, ack 1, win
453, options [nop,nop,TS val 21041889 ecr 21041813], length 8
00:00:02.026411 IP 10.92.213.238.49784 > 10.92.213.231.5757: Flags [.], ack 25, win 457, options
[nop,nop,TS val 21041913 ecr 21041889], length 0
00:00:03.028760 IP 10.92.213.231.5757 > 10.92.213.238.49784: Flags [P.], seq 25:33, ack 1, win
453, options [nop,nop,TS val 21041989 ecr 21041913], length 8
00:00:03.029824 IP 10.92.213.238.49784 > 10.92.213.231.5757: Flags [.], ack 33, win 457, options
[nop,nop,TS val 21042013 ecr 21041989], length 0
00:00:04.032088 IP 10.92.213.231.5757 > 10.92.213.238.49784: Flags [P.], seq 33:41, ack 1, win
453, options [nop,nop,TS val 21042090 ecr 21042013], length 8
00:00:04.033088 IP 10.92.213.238.49784 > 10.92.213.231.5757: Flags
[.], ack 41, win 457, options [nop,nop,TS val 21042114 ecr 21042090],
length 0
00:00:22.039894 IP 10.92.213.231.5757 > 10.92.213.238.49784: Flags
[P.], seq 41:185, ack 1, win 453, options [nop,nop,TS val 21043891
ecr 21042114], length 144
00:00:22.040918 IP 10.92.213.238.49784 > 10.92.213.231.5757: Flags [.], ack 185, win 473, options
[nop,nop,TS val 21043914 ecr 21043891], length 0
00:00:23.040658 IP 10.92.213.231.5757 > 10.92.213.238.49784: Flags [P.], seq 185:193, ack 1, win
453, options [nop,nop,TS val 21043991 ecr 21043914], length 8
```

Note

- Packets aren't retransmitted, data is retransmitted
- So the dropped packets' data is all re-sent in the minimum number of packets it will fit in

Packets Dropped

```
igb@pi-one:~$ sudo iptables -I OUTPUT -p tcp -m tcp --sport 5757 -j DROP; sleep 30; sudo iptables
--list -v; sudo iptables --delete OUTPUT 1
Chain INPUT (policy ACCEPT 53 packets, 3108 bytes)
 pkts bytes target     prot opt in     out     source               destination

Chain FORWARD (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination

Chain OUTPUT (policy ACCEPT 54 packets, 10767 bytes)
 pkts bytes target     prot opt in     out     source               destination
   30  4153 DROP       tcp  --  any    any     anywhere             anywhere             tcp spt:
5757
igb@pi-one:~$
```

Drop Packets on Other Side

- If we drop packets at the INPUT of the receiver, we can see the retransmissions

```
sudo iptables -I INPUT -p tcp -m tcp --sport 5757 -j DROP; \
   sleep 10; sudo iptables -D INPUT 1
```

```
00:02:55.213001 IP 10.92.213.238.49817 > 10.92.213.231.5757: Flags [.], ack 17, win 457, options
[nop,nop,TS val 21158147 ecr 21158123], length 0
00:02:56.215151 IP 10.92.213.231.5757 > 10.92.213.238.49817: Flags [P.], seq 17:25, ack 1, win
453, options [nop,nop,TS val 21158223 ecr 21158147], length 8
00:02:56.216163 IP 10.92.213.238.49817 > 10.92.213.231.5757: Flags [.], ack 25, win 457, options
[nop,nop,TS val 21158247 ecr 21158223], length 0
00:02:57.215914 IP 10.92.213.231.5757 > 10.92.213.238.49817: Flags [P.], seq 25:33, ack 1, win
453, options [nop,nop,TS val 21158323 ecr 21158247], length 8
00:02:57.418894 IP 10.92.213.231.5757 > 10.92.213.238.49817: Flags [P.], seq 25:33, ack 1, win
453, options [nop,nop,TS val 21158344 ecr 21158247], length 8
00:02:57.628899 IP 10.92.213.231.5757 > 10.92.213.238.49817: Flags [P.], seq 25:33, ack 1, win
453, options [nop,nop,TS val 21158365 ecr 21158247], length 8
00:02:58.048900 IP 10.92.213.231.5757 > 10.92.213.238.49817: Flags [P.], seq 25:33, ack 1, win
453, options [nop,nop,TS val 21158407 ecr 21158247], length 8
00:02:58.888921 IP 10.92.213.231.5757 > 10.92.213.238.49817: Flags [P.], seq 25:33, ack 1, win
453, options [nop,nop,TS val 21158491 ecr 21158247], length 8
00:03:00.568905 IP 10.92.213.231.5757 > 10.92.213.238.49817: Flags [P.], seq 25:33, ack 1, win
453, options [nop,nop,TS val 21158659 ecr 21158247], length 8
00:03:03.938934 IP 10.92.213.231.5757 > 10.92.213.238.49817: Flags [P.], seq 25:33, ack 1, win
453, options [nop,nop,TS val 21158996 ecr 21158247], length 8
00:03:10.678934 IP 10.92.213.231.5757 > 10.92.213.238.49817: Flags [P.], seq 25:33, ack 1, win
453, options [nop,nop,TS val 21159670 ecr 21158247], length 8
00:03:10.679897 IP 10.92.213.238.49817 > 10.92.213.231.5757: Flags [.], ack 33, win 457, options
[nop,nop,TS val 21159693 ecr 21159670], length 0
00:03:10.680249 IP 10.92.213.231.5757 > 10.92.213.238.49817: Flags [P.], seq 33:137, ack 1, win
453, options [nop,nop,TS val 21159670 ecr 21159693], length 104
00:03:10.681190 IP 10.92.213.238.49817 > 10.92.213.231.5757: Flags [.], ack 137, win 457, options
[nop,nop,TS val 21159693 ecr 21159670], length 0
```

UNIVERSITY OF BIRMINGHAM

Why don't the packets get bigger?

- Implementation decision: could be different on other operating systems (although appears same on Solaris, as it happens)
- Makes sense on slow, lossy network: once you suspect packets are going missing, more sensible to send small packets to find out if it is working than large ones.