TCP Protocol - Transport Layer

TCP/IP class

Jim Binkley

outline

- intro
- sliding window protocol
- tcp concepts
  - header/piggybacking
  - ports and sockets (connection idea)
- open/close state machine
- some protocol mechanics
- performance

intro

- TCP - Transmission Control Protocol
- reliable, connection-oriented stream (point to point) protocol
  - if UDP is like U.S. Mail
  - TCP is like a phone call (cannot broadcast/multicast)
- we need stream delivery because underlying mechanism has flaws
  - out of order due to routing, or loss, or corruption (or dups due to timeout/resend)

intro

- RFC 793 and host requirements 1122
- TCP has own jargon:
  - **socket** - we'll see in a bit
  - **segment**: a TCP packet
  - **MSS: maximum segment size**, max pkt one TCP side can send another, negotiated at connection time
  - **ports**:

TCP properties

- stream orientation. stream of OCTETS (bytes) passed between send/recv
- byte stream is full duplex
  - think of it as two independent streams joined with **piggybacking** mechanism
- piggybacking - one data stream has control info for the other data stream (going the other way)
- unstructured stream
  - tcp doesn't show packet boundaries to applications

TCP properties

- unstructured stream, cont
  - but you can still structure your i/o as "messages" or structures if you want
- virtual circuit connection
  - client connects and server listens/accepts
  - i/o transfers don't have remote peer address
- tcp provides flow control
  - you don't have to worry about recv buffering

writing structures down TCP pipe

```
struct foo { int x; int y; int z; } f;

write (sock, &f,  sizeof (struct foo));
```

```
read(sock,
        &f,
        sizeof (struct
                foo));
```

* not exactly

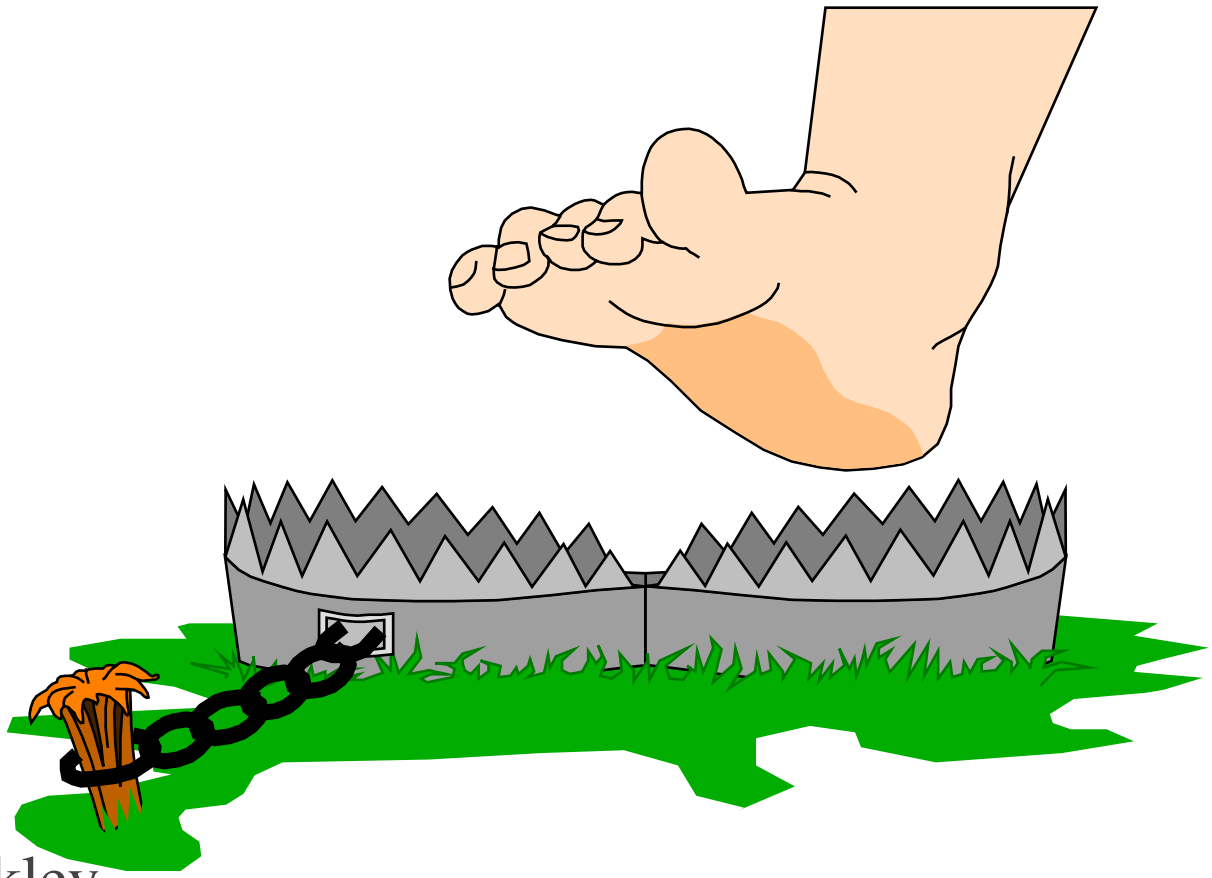
TCP properties

- efficiency - not easy to duplicate over WAN environment
- congestion detection end to end
  - backs off if it thinks net is congested
- **most TCP/IP error handling is in TCP**
  - end to end
- complex protocol
  - can treat telnet (interactive) and ftp (bulk transfer) differently + acks/timers, etc

TCP buffering and prog POV

- can't predict very well as programmer how tcp will buffer data
  - write 2 512 byte packets, might read 1 1k pkt
  - write 1 1024 byte packet, might read 2 512
  - type in 3 characters, tcp might send them as 1 3 byte packet
- It's a data stream
- writes are atomic, unless you use non-blocking I/O. reads are not

reinventing tcp...

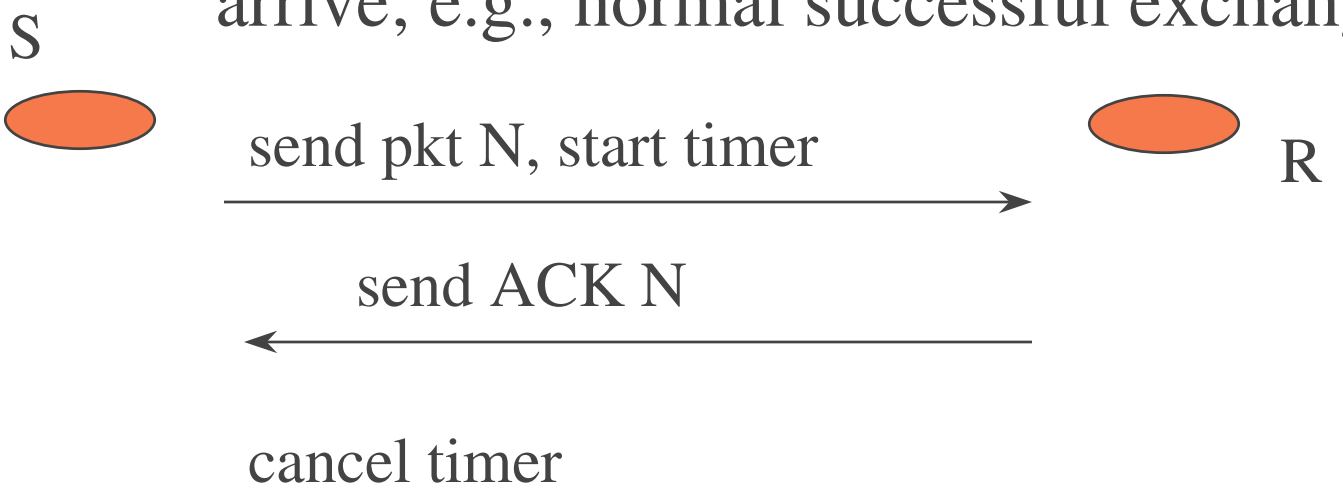


TCP - Complex protocol

- Doug Comer states that TCP is a protocol and not any particular specification or chunk of software. True, but
- Binkley states: “Don’t try this at home, borrow one from the Internet”
  - 4.4 BSD, KA9Q, linux
- TCP is not easy to debug or test in terms of interoperability (or replace with a reliable UDP app)

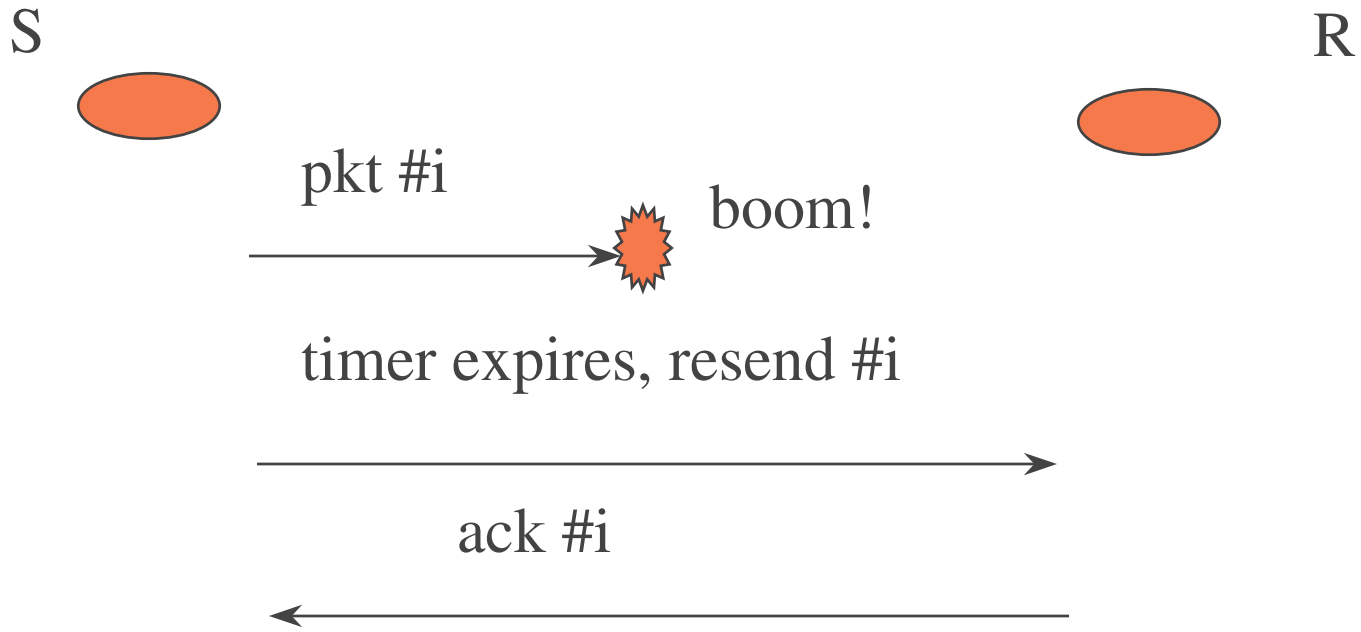
sliding window protocol basics

- positive acknowledge with retransmission
  - recv must send ACK with seq #
  - sender must timeout and resend if ACK fails to arrive; e.g., normal successful exchange:



sliding window

- and when it doesn't work



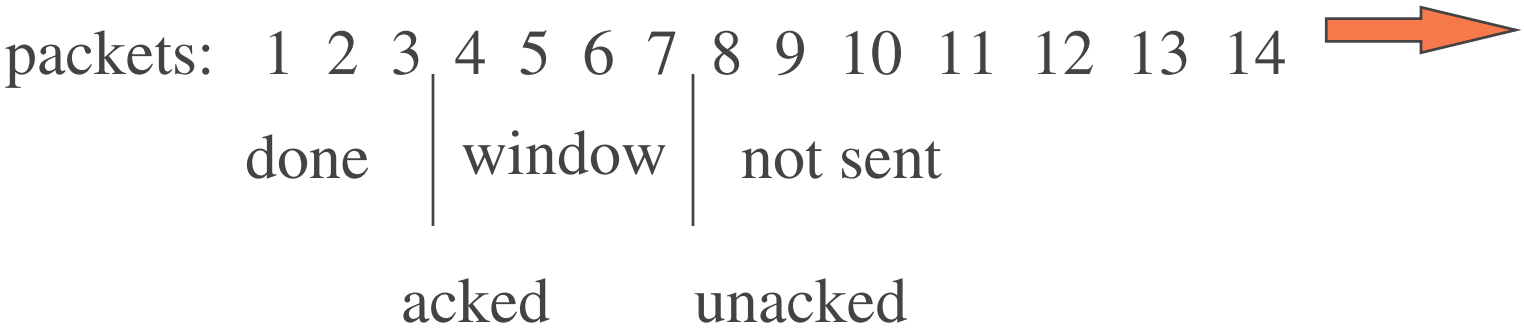
because we resend: we can duplicate pkts and ACKS!

sliding window protocol

- simple pos. ack. with retransmission is called “ping-pong” protocol, not efficient
- send && wait for ACK, therefore data only flows one way at a time, not both ways, **thus we cut efficiency of channel in half**
- we want to send many packets (asap) and get back ACKS (possibly combined into 1 cumulative ACK)

sliding window

- sliding window is more complex form of pos. ack. with retransmission
- we still retransmit and we still want ACKS

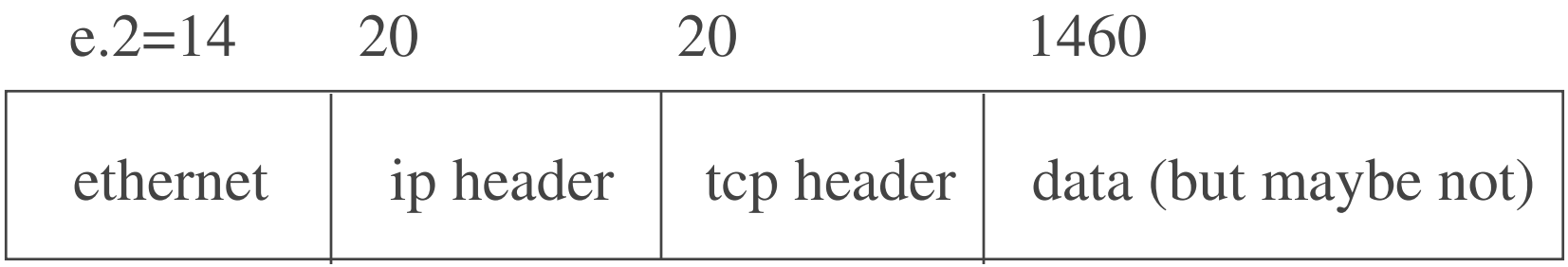


1-3 sent and ACKED, 4-7 in window and sent but not ACKED, if ACK arrives, sender slides window up

sliding window, cont.

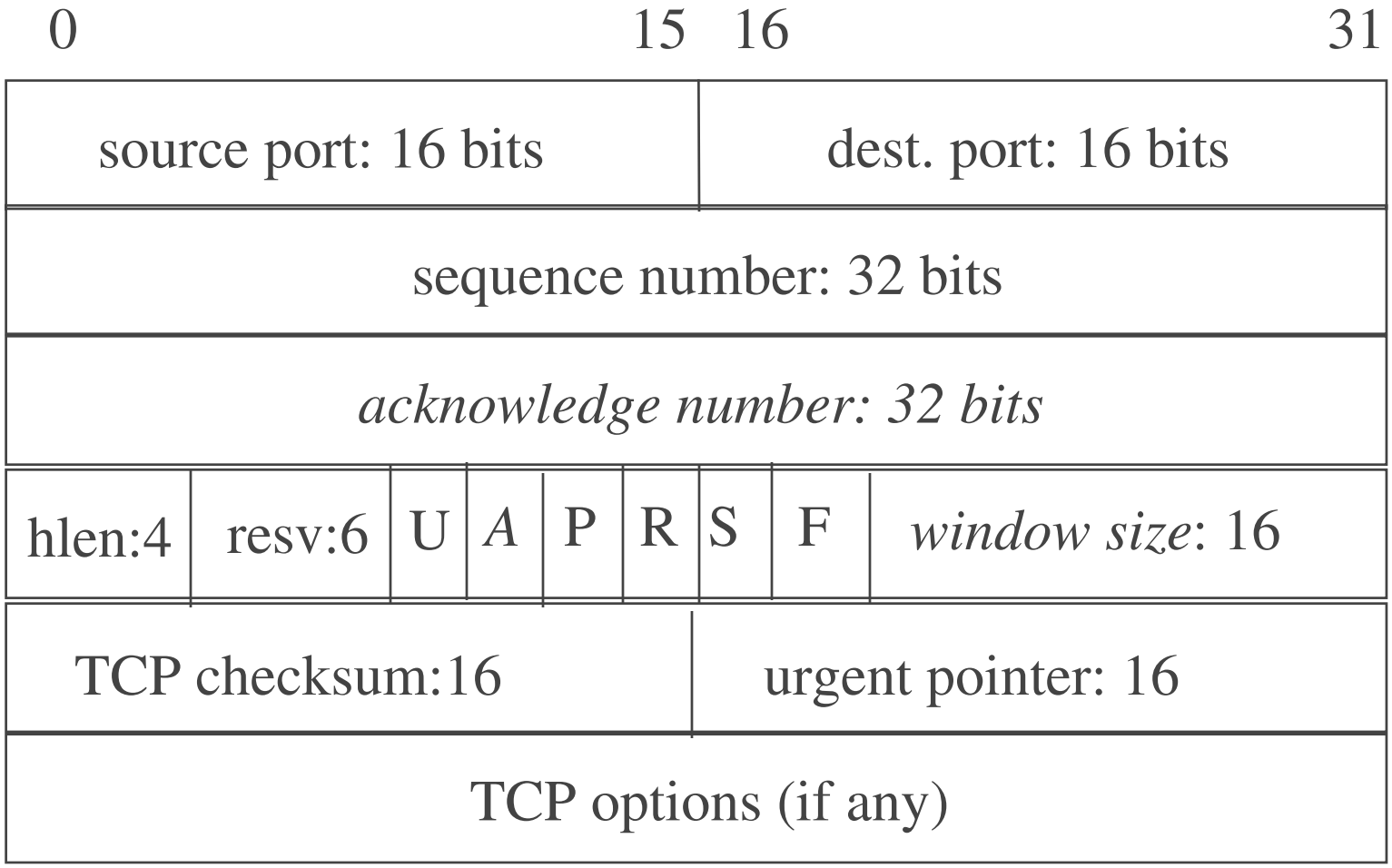
- goal in previous slide: cumulative ACK
  e.g., [ACK up to #7]
- tcp uses bytes not packets for sequencing
- recv-side controls sliding window and views that as available buffering, can stop sending by telling it window size is 0 in ACK, thus **flow control**
- with window size == 1, we get simple positive ack with retransmission

TCP - encapsulation



TCP header may have options, but default size is 20 bytes

TCP header


0
15
16
31
source port: 16 bits
dest. port: 16 bits
sequence number: 32 bits
acknowledge number: 32 bits
hlen:4
resv:6
U
A
P
R
S
F
window size: 16
TCP checksum:16
urgent pointer: 16
TCP options (if any)

Header - explained

- header sent in every TCP packet, may just be control message (SYN/FIN/ACK) with no data
- view TCP as 2 sender/recv data streams with control information sent back the other way (piggybacking)

header slides

- source port: 16 bits, the TCP source port
- destination port: 16 bits, note ports in 1st 8 bytes
- sequence number: 1st data octet in this segment (from send to recv): 32 bit space
- ack: if ACK flag set, next expected sequence number (piggybacking; i.e., we are talking about the flow the other way)

header, cont.

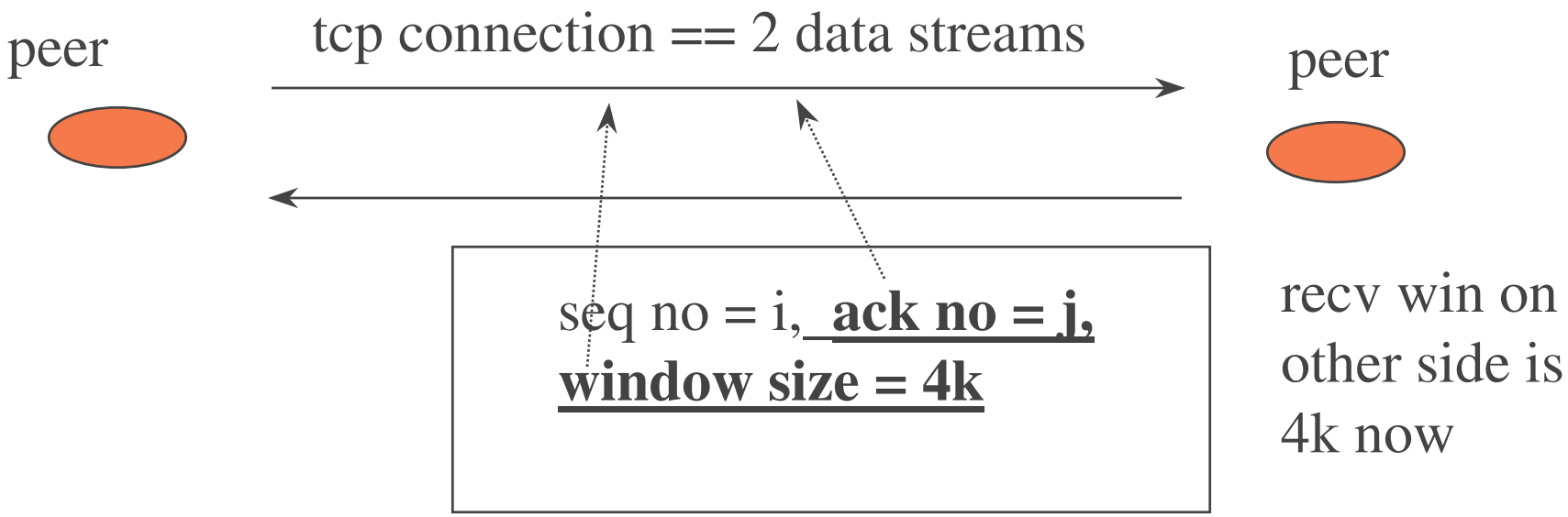
- hlen: # of 32 bit words in header
- reserved: not used
- flags
  - URG: - urgent pointer field significant
  - ACK:- ack field significant (this pkt is an ACK!)
  - PSH: - push function (mostly ignored)
  - RST: - reset (give up on) the connection (error)
  - SYN: - initial synchronization packet (start connect)
  - FIN: - final hangup packet (end connect)

header, cont.

- window: window size, begins with ACK field that recv-side will accept (piggyback)
- checksum: 16 bits, pseudo-header, tcp header, and data
- urgent pointer: offset from sequence number, points to data following urgent data, URG flag must be set
- options - e.g., Max Segment Size (MSS)

TCP piggybacking (header)

- data may be sent 2-ways, sender to recv (1-way data flow) may contain piggybacked state (ack/window fields) for other data channel. This info is feedback on other channel



ports and sockets

- TCP clients and servers have a TCP port in the TCP port space 0 (not used)..64k-1
- unlike UDP, TCP uses connection as fundamental abstraction, not port
- when we connect, we end up with:
  peer (client): 18.25.0.36,1069; 128.10.2.3,25
  peer (server): 128.10.2.3,25; 18.25.0.36,1069
- each side has 4-tuple (**socket**) which is used to id incoming packets (demux to app)

ports and sockets

- TCP server architecture thus

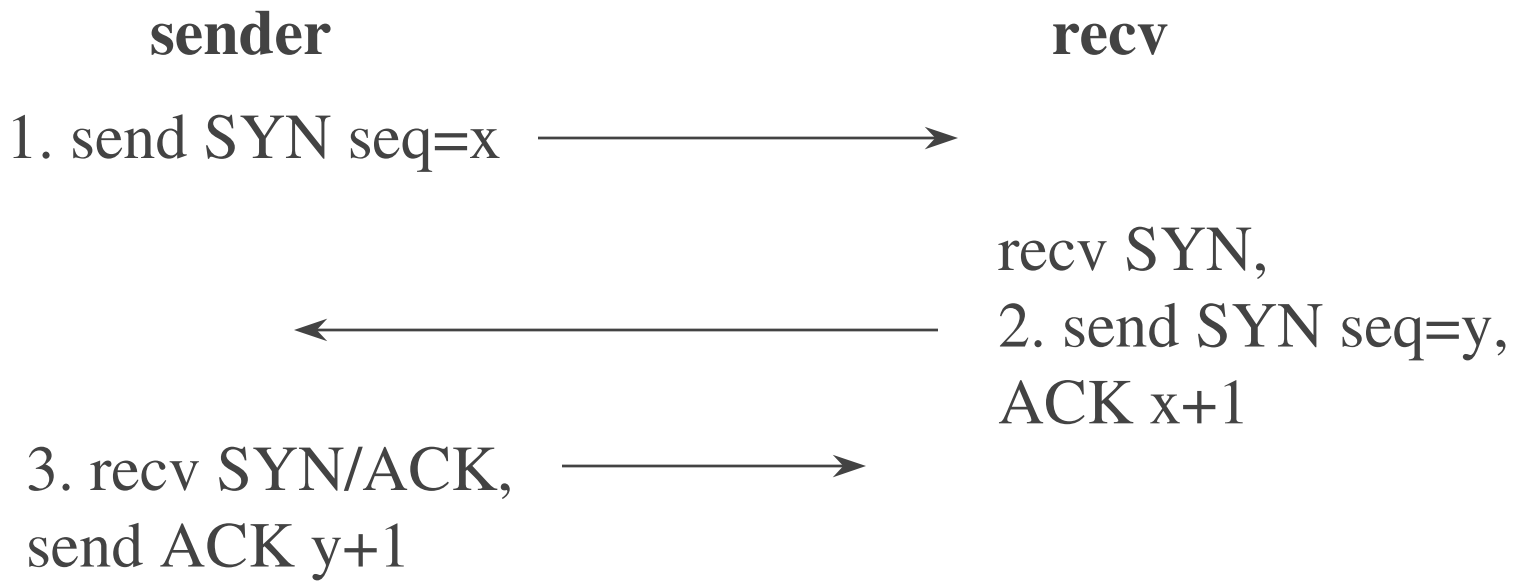
  connected_fd = accept(listen_fd, ...);

- server may spinoff “slave” thread on connected fd to take care of application-level protocol (some sequence of read/write calls)
- all server processes bound to WK port have same server-side port #; e.g., http/80

TCP open/close

- TCP distinguishes **passive** and **active** open
- servers usually do passive open, means they LISTEN
- clients usually do active open, means they connect
- reach ESTABLISHED state after 3-way handshake

open/close, 3 way handshake



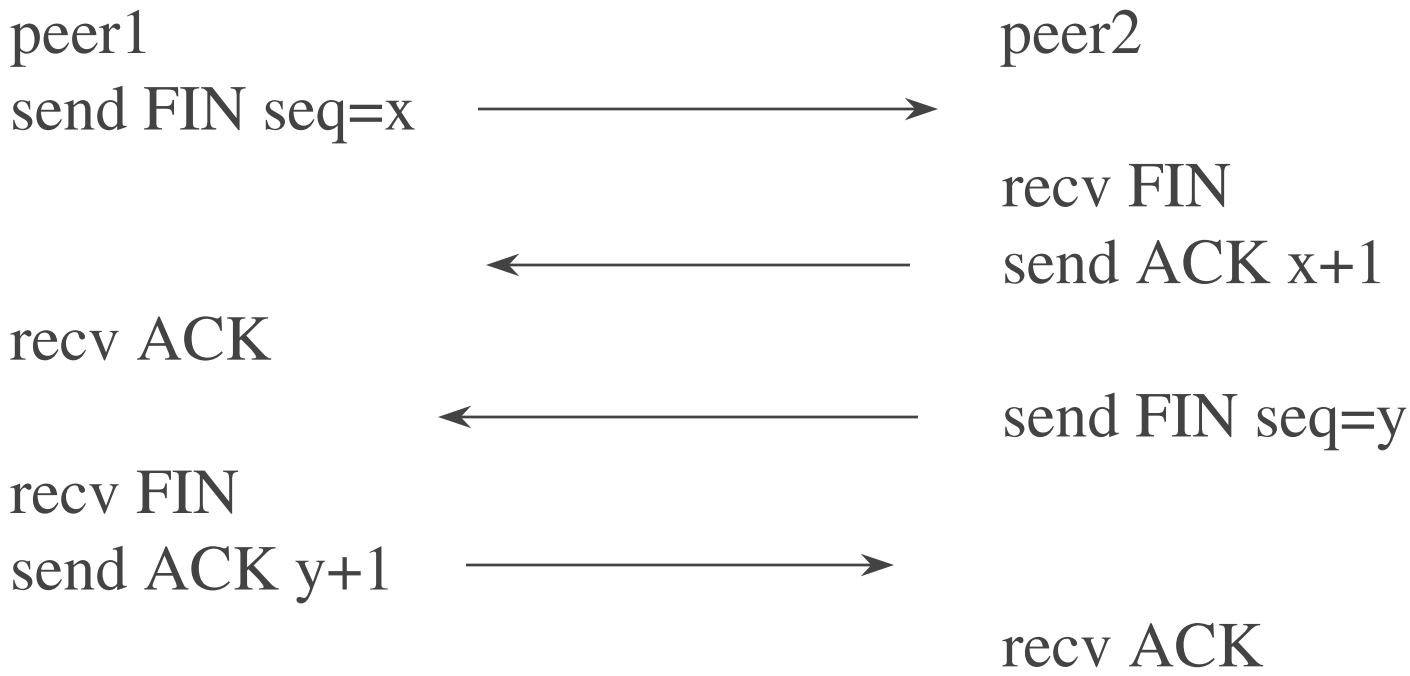
both sides can SYN at the same time and it will work

results include established connection, initial sequence numbers exchanged, ACKS ack next expected byte (cumulative)

closing a tcp connection

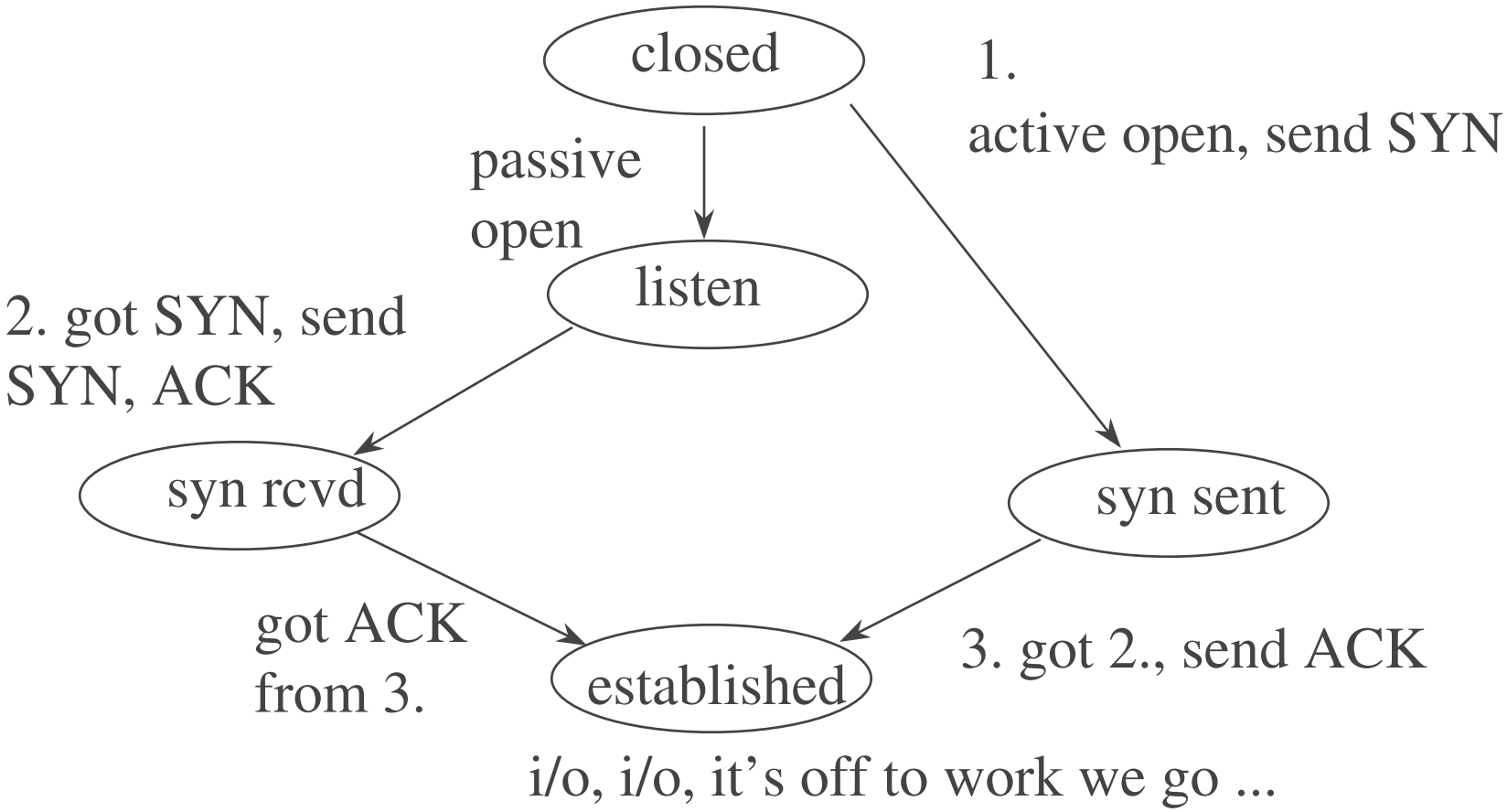
- connections are full duplex and it is possible to shutdown(2) one side at a time
- close(2) closes everything and the UNIX version doesn't quite jibe with TCP - UNIX close is async and doesn't wait for handshake
- really just 2 2-way handshakes (send FIN, recv replies with ACK per channel)
- interesting problem: how do you make sure last ACK got there (can't ACK it...)

close

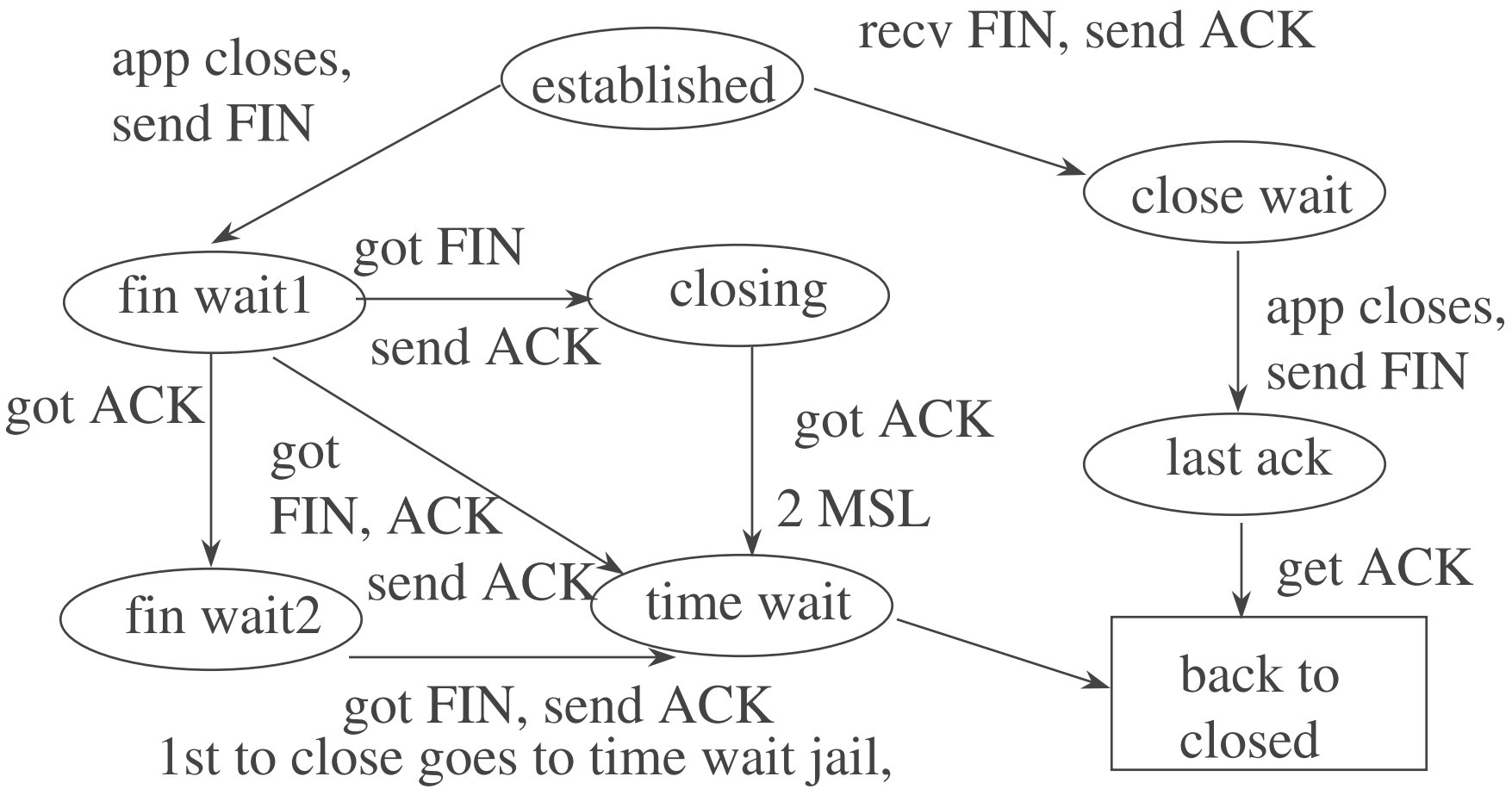


1st to send FIN is last to send ACK,
no way to know if that ACK arrived

state machine - simplified open



state machine - close



the FINE for the FIRST FIN is ...

- both apps could close first and send FIN, hence left side is more complex, but state machine supports async close
- s/he who closes first, gets stuck in TIME_WAIT state since we aren't getting an ACK back for our ACK sent, must wait 2 MSL (max segment length) time, 1 or 2 minutes typically

Protocol Mechanisms (some)

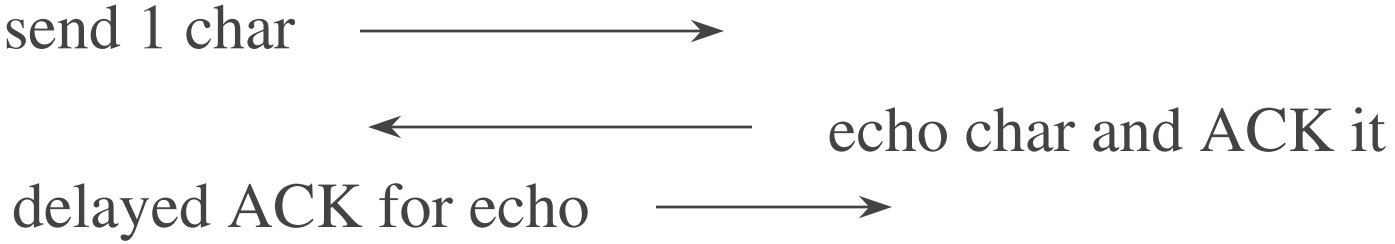
- Stevens, p. 227
  "There is no single correct way for two TCPs to exchange a given amount of data"
- window size - flow control
- delayed ack
- nagle algorithm
- adaptive retransmission + backoff
- congestion control

TCP variable window size

- flow control occurs because the receive side controls the window size
- if window size == 0, the sender cannot send data
- sender will send window probe (1 byte of data) to see if window is open (ack might be lost). Separate timer for this function called persistance timer
- this is end to end flow control, doesn't include routers

delayed ACK

- try to not send ACK immediately and hope that data will show up so that ACK can piggyback (free ride, not extra packet)
- delay typically 200 milliseconds
- this is recv-side timer, not send ACK timer
- with telnet might see



nagle algorithm

- traditional telnet over wan can add to congestion because we have 40 bytes of header for 1 echoed byte of data
- rfc 896 - nagle algorithm
- tcp connection can only have one unacked outstanding small segment. Can't send more until you get an ACK, sender may collect more data
- only affects sender who sends small data amounts
- TCP_NODELAY socket option turns this off

nagle, cont.

- algorithm is said to be “self-clocking”, you can go as fast as round trip latency will allow since you wait for return ACK
- hope is that sender slows down to give data opportunity “to pile up” before it is sent
- X-windows would not want nagle algorithm since it would send small data chunks (mouse clicks) and want those sent as real-time as possible

timeout and retransmission

- can't use fixed time for send ACK timer
- if too long, response not good if timeout occurs,
- if too short, can't know apriori how long to wait (and congestion might change the time)
- TCP uses **adaptive retransmission timer,**
- see text for details, uses fixed-point arithmetic

simple timer backoff

- if no acks at all are received tcp will use a modified form of exponential backoff
- Stevens (p.299) gives 1,3,6,12,48, 64 on one implementation, retries at a minute until 9 minutes then a reset
- will this work for the Mars Mission?
- if packets start showing up backoff is removed

congestion control in TCP

- routers may drop packets as space is not pre-allocated by definition - congestion
- routers don't have effective mechanism to indicate congestion (ICMP source quench is not it...) to sender
- assumption: packet loss due to damage is small, therefore TCP assumes it means congestion since ACKS do not come back

congestion control

- TCP uses **slow start** and **multiplicative decrease** to deal with congestion
- Van Jacobson 1988 outlined these ideas
- slow-start roughly: whenever starting traffic or recovering from congestion, start congestion window at the size of a single segment and increase it (up to a point) as ACKs show up

congestion avoidance

- multiplicative decrease - upon loss of a segment, reduce the congestion window by half down to a minimum of 1. For those segments that remain in the send window, backoff the retransmission timer exponentially.

routers and congestion - RED

- routers might use an obvious queue-drop mechanism
  - too many buffers; drop packets at end of queue call this a“**tail-drop**” policy
  - on heavily multiplexed router many TCP connections may lose a packet and be forced into slow-start
- routers may use **Random Early Detection** (or RED) - basically randomly discard packets in Queue at a certain saturation point
  - thus avoid tail-drop policy

TCP efficiency/performance

- mid 80's say with VAX on ethernet, performance was poor, now can find good approximation of 1 gigabit on faster end hosts, 90% of 100BASE common
- assumptions (from Stevens):
  - send two packets into 2 pkt window
  - get one ack
  - 2 hosts on ethernet
  - max data possible

performance, cont.

- equation:

$$\frac{\text{real data bytes}}{\text{header (overhead)}} \quad \frac{\text{bits/sec}}{\text{8 bits}} \quad \Rightarrow \quad \frac{\text{bytes}}{\text{sec}}$$

assume ethernet:

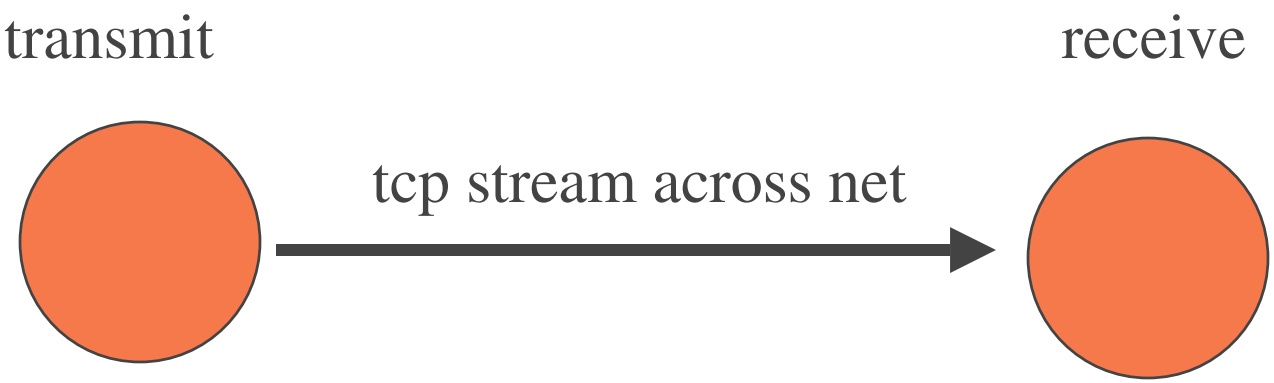
$$\frac{2 * 1460}{2 * 1538 + 84} * \frac{10000000}{8} = 1{,}155{,}063$$

$$\frac{1155063}{1250000} = 92.4\%$$

ttcp - used to measure tcp thruput

- common on UNIX hosts
- disks not part of measurement
- normalize 1st say with measurement over localhost
- then between hosts on net
- may use different window sizes,
  - 8k/16k/32k/64k ...

ttcp test - test ttcp



ttcp -t -s ...

ttcp -r -s

note: this is memory to memory, no disks involved

some observations

- commodity P3 cpus with gigabit Ethernet card can easily do 500mbits with TTCP
- 781 mbits between two Crays over 800 Mbs hippi channel
- 907 over Cray loopback
- can’t go faster than slowest link
- can’t go faster than memory bandwidth
- can’t go faster than window size/rtt

constraints are finally

- window-size (new window-size and PAWS options are significant here)
- speed of light

study questions

- assume you have a TCP connection (telnet to site Y) and you reboot a router in between, what should happen?
- if you have a TCP connection and you reboot one of the end-end systems, what should happen?
- what would be the problems with having TCP support multicast addresses?

study questions

- it is widely assumed (if not cherished) that TCP would make a poor mechanism for transfer of audio/video steady-stream data? Why is that? and can you make a case for a contrarian point of view?
- you telnet to mars... what should you think about in terms of tcp timers?
  - hint: it might take about 10 minutes one-way to Mars for light (depends on mars/earth orbits)