Internet Protocol Stack

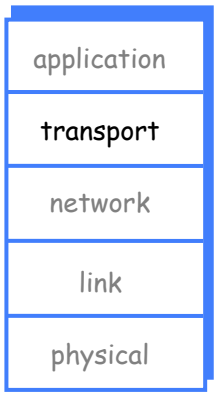
application: supporting network applications
- HTTP, SMTP, FTP, etc.

transport: endhost-endhost data transfer
- TCP, UDP

network: routing of datagrams from source to destination
- IP, routing protocols

link: data transfer between neighboring network elements
- Ethernet, WiFi

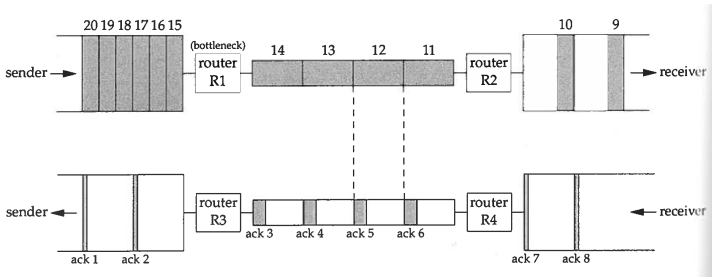
physical: bits "on the wire"



Congestion Control

What gives rise to congestion?

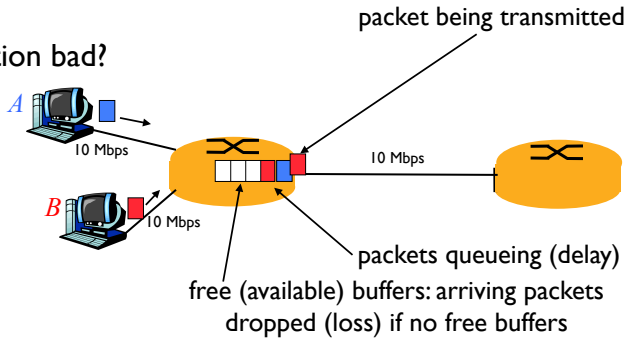
When offered load is greater than system capacity,
too much data for the *network* (as opposed to receiver) to handle



Causes and Implications of Congestion

- caused by packets arriving faster than a router can forward them
- routers queue packets that they cannot serve immediately
- if queue overflows, packets are dropped
- queued packets experience delay

Why is congestion bad?



Consequences of Congestion

If queueing delay > RTO, sender retransmits packets, adding to congestion

Dropped packets also lead to more retransmissions

If unchecked, could result in **congestion collapse**

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

What prevention?

Constraints:

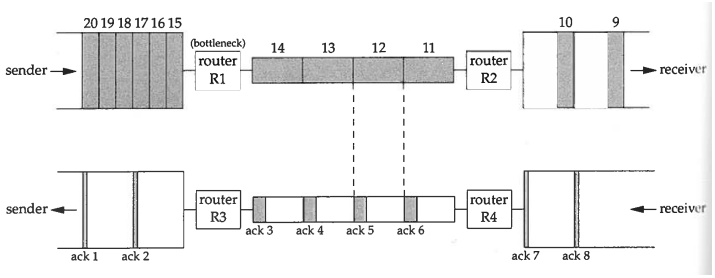
- decentralized control
- unlike routing, no local reaction at routers (beyond buffering and dropping)
- long feedback time
- dynamic network condition: connections come and go

Congestion Control

How can senders prevent/avoid congestion?

•

•

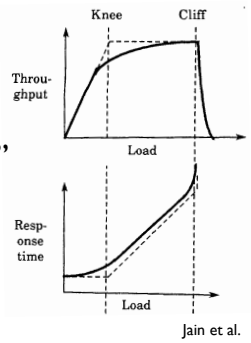


Congestion Control

How do senders know what the system capacity is?

Load vs. throughput:

- congestion avoidance: operate system at "knee"
- congestion control: operate system near "cliff"



Jain et al.

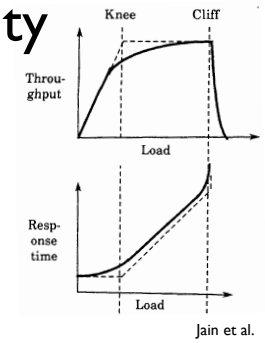
How do senders discover system capacity and control congestion?

•

•

Discovering System Capacity

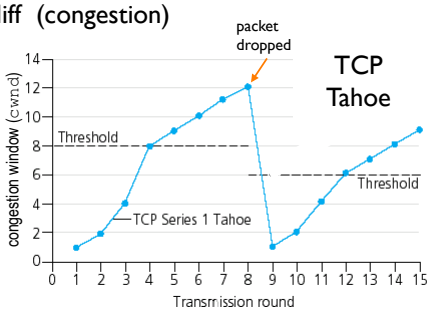
How do senders know where the "knee" is and how do they know when the system has "gone off the cliff"?



Jain et al.

What TCP and other congestion control schemes do:

- probe for point right before cliff ("pipe size")
- slow down transmission on detecting cliff (congestion)
- fast probing initially, up to a threshold ("slow start")
- slower probing after threshold is reached ("linear increase")



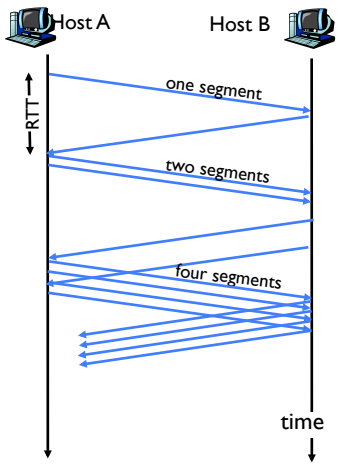
Why not send a large amount of data and then slow down?

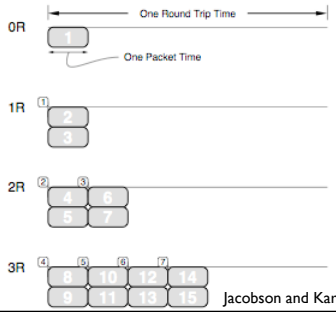
TCP Slow-start

When connection begins, increase rate exponentially until first loss event:

- double `cwnd` every RTT (every ACK received)

⇒ initial rate is slow but ramps up exponentially fast





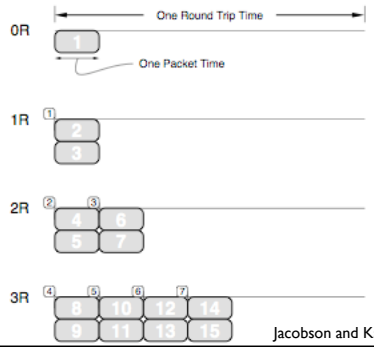
Jacobson and Karels

Self-Clocking TCP

ACK is used for flow control and retransmission
TCP also uses ACK for congestion control

TCP follows a so-called "Law of Packet Conservation":
Do not inject a new packet into the network until a resident departs (ACK received)

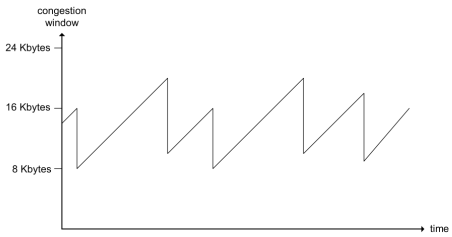
Since packet transmission is timed by receipt of ACK, TCP is said to be *self-clocking*



Jacobson and Karels

TCP Congestion Control

Sender maintains a congestion window (`cwnd`)
Sender does congestion control by expanding and shrinking `cwnd`



Sender's send window (`wnd`) is:
`wnd = MIN(rwnd, floor(cwnd)),`

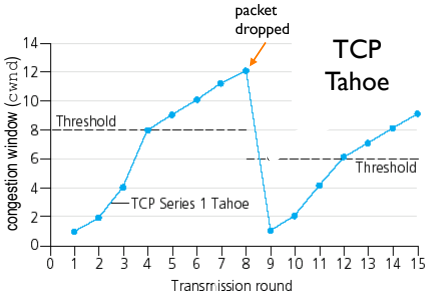
- where `rwnd`: receiver's advertised window
- initially set `cwnd` to 1
- increase `cwnd` if there's no congestion
- on congestion, decrease `cwnd`

Questions:
1. How to determine that there's no congestion?
2. By how much do we increase `cwnd`?
3. How to detect congestion?
4. By how much must `cwnd` be decreased?

TCP Congestion Control

Variables:

- `cwnd`: congestion window (init: 1)
- `ssthresh`: threshold for slowing down `cwnd` increase (init: 64KB)



Recall that TCP uses cumulative ACK:

- ACKs the last-byte received in order
- tells sender the next-expected seq#
- subsequent out-of-order packets generate the same cumulative ACK

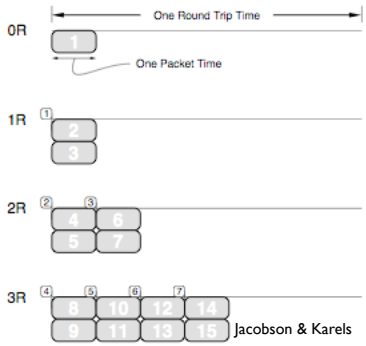
Increasing `cwnd`

Probing the "pipe-size" (system capacity) in two phases:

1. slow-start: exponential increase

```
while (cwnd <= ssthresh) {
    cwnd += cwnd
} for every ACKed cwnd
```

OR: send out 2 packets for every new ACK received

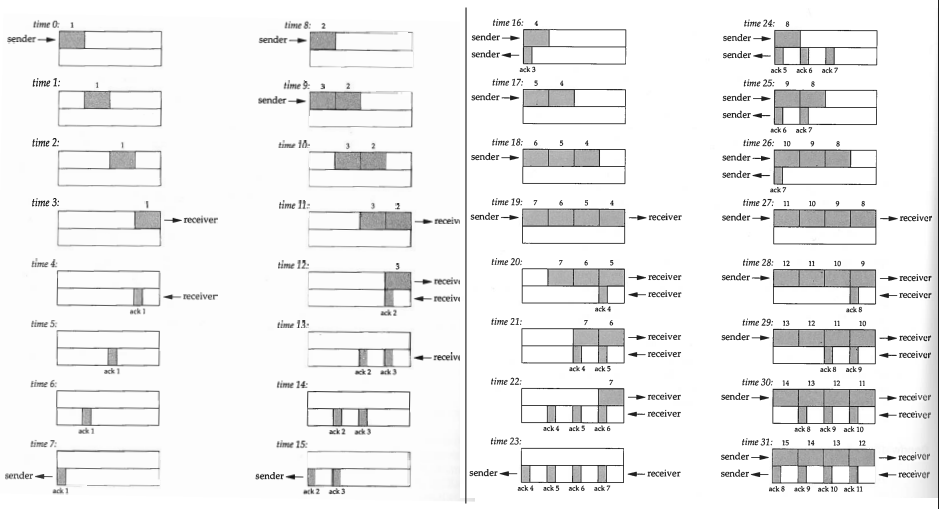


2. congestion avoidance: linear increase

```
while (cwnd > ssthresh) {
    cwnd += 1
} for every cwnd-full of ACKs
```

OR: `cwnd += 1/floor(cwnd)` for every ACK

TCP Slow Start Example



Stevens

Decreasing `cwnd`

How to detect congestion?

Interpret packet loss as congestion signal

When is packet considered lost?

- RTO timer goes off
- receipt of 3 duplicate ACKs

Fast Retransmit

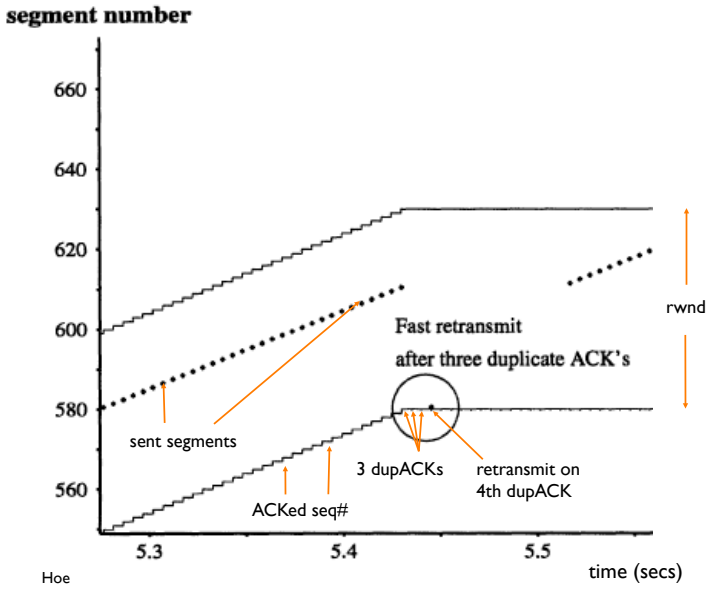
Motivation: waiting for RTO is too long

TCP Tahoe does fast retransmit:

- with cumulative ACK, receipt of packets following a lost packet cause duplicate ACKs to be returned
- interpret 3 duplicate ACKs as an *implicit* NAK
- retransmit upon receiving 3 dupACKs, i.e., on receipt of the 4th ACK with the same seq#, retransmit segment
- why 3, instead of 2 or 4, dupACKs?

With fast retransmit, TCP can retransmit after 1 RTT instead of waiting for RTO

Fast Retransmit Example



TCP Congestion Recovery

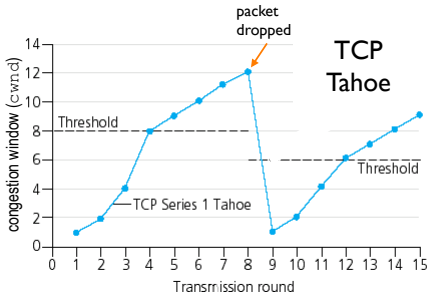
Once congestion is detected,

- by how much do we decrease `cwnd`?
- how do we recover from congestion?
  - which packet(s) do we retransmit?
  - how do we increase `cwnd` again?

First, reset `ssthresh = cwnd/2`

TCP Tahoe:

- retransmit using Go-Back-N
- reset `cwnd`=1
- restart slow-start



TCP Reno and Fast Recovery

`cwnd` re-opening and retransmission of lost packets regulated by returning ACKs

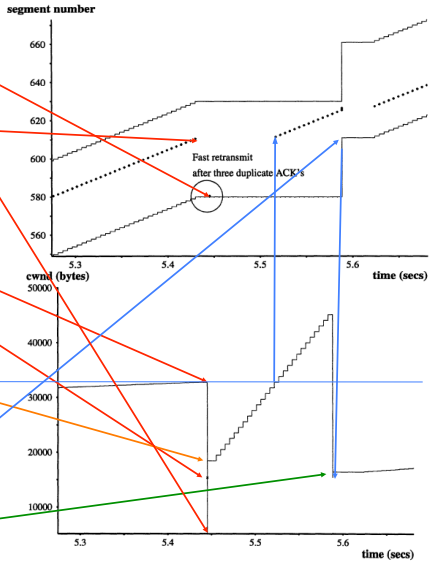
- duplicate ACK doesn't grow cwnd, so TCP must wait at least 1 RTT for fast retransmitted packet to cause a non-duplicated ACK to be returned
- if RTT is large, Tahoe re-grows `cwnd` very slowly

TCP Reno does **fast recovery**:

- current value of `cwnd` is the estimated system capacity
- after congestion is detected, want to continue transmitting at half the estimated capacity, how?
- each returning ACK signals that an outstanding packet has left the network
- don't send any new packet until half of the expected number of ACKs have returned

Fast Recovery

1. on congestion, retransmit lost segment, set `sstresh = cwnd/2`
2. remember highest seq# sent, `snd_high`; and remember current `cwnd`, let's call it `pipe`
3. decrease `cwnd` by half
4. increment `cwnd` for every returning dupACK, incl. the 3 used for fast retransmit
5. send new packets (above `snd_high`) only when `cwnd > pipe`
6. exit fast-recovery when a non-dup ACK is received
7. set `cwnd = ssthresh + 1` and resum linear increase



cwnd: number of bytes unACKed

Summary: TCP Congestion Control

- when `cwnd` is below `ssthresh`, sender in **slow-start** phase, window grows exponentially
- when `cwnd` is above `ssthresh`, sender is in **congestion-avoidance** phase, window grows linearly
- when a **3 dupACK** occurs, `ssthresh` set to `cwnd/2` and `cwnd` set to new `ssthresh`
- if more dupACKs return, do **fast recovery**
- else when **RTO** occurs, `ssthresh` set to `cwnd/2` and `cwnd` is set to 1 MSS

TCP Congestion Control Examples

TCP keeps track of outstanding bytes by two variables:

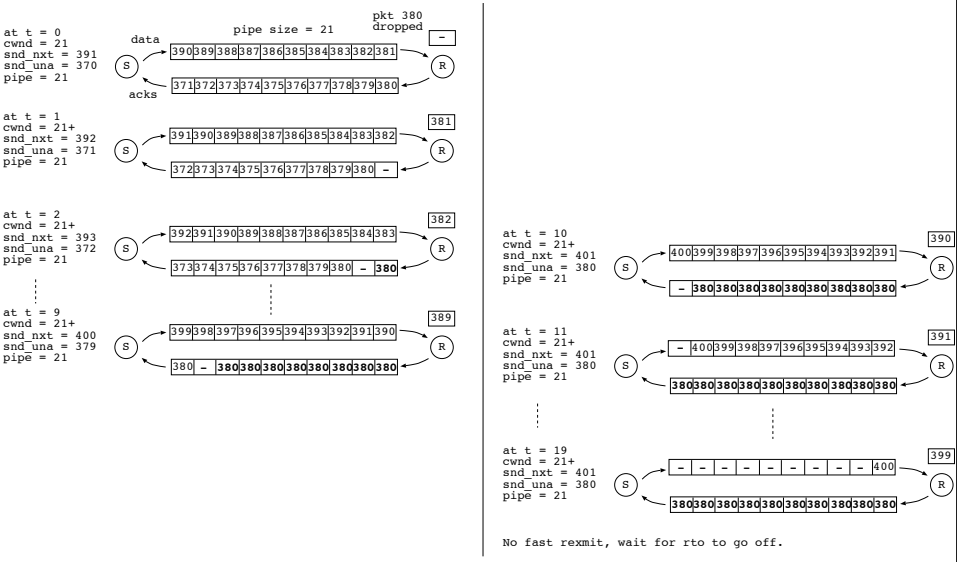
1. `snd_una`: lowest unACKed seq#,
   i.e., `snd_una` records the seq# associated with the last ACK
2. `snd_next`: seq# to be sent next

Amount of outstanding bytes: `pipe = snd_next - snd_una`

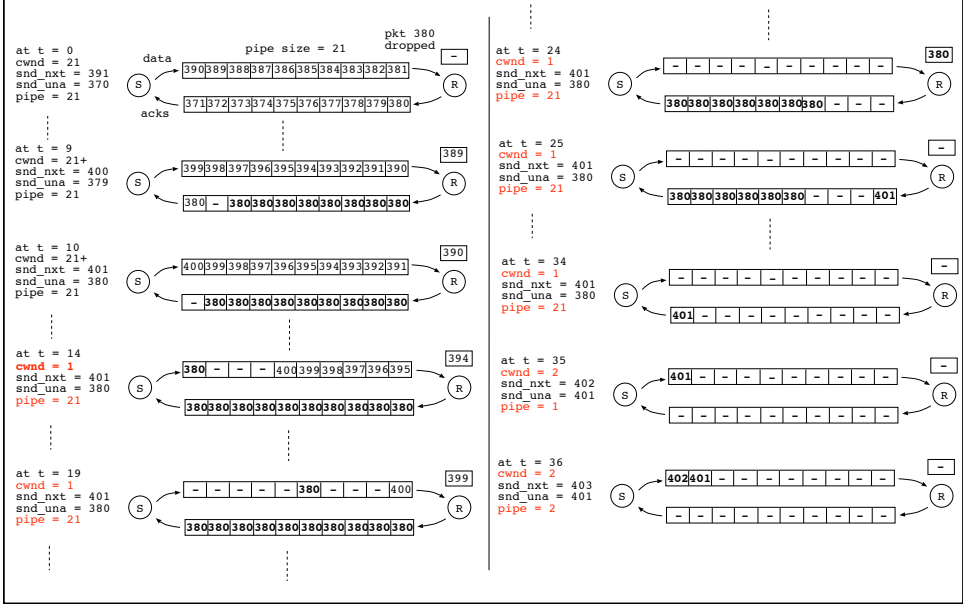
Scenario:

- 1 byte/pkt
- receiver R takes 1 transmit time to return an ACK
- sender S sends out the next packet immediately upon receiving an ACK
- `rwnd` = ∞
- `cwnd` = 21, in linear increase mode
- `pipe` = 21

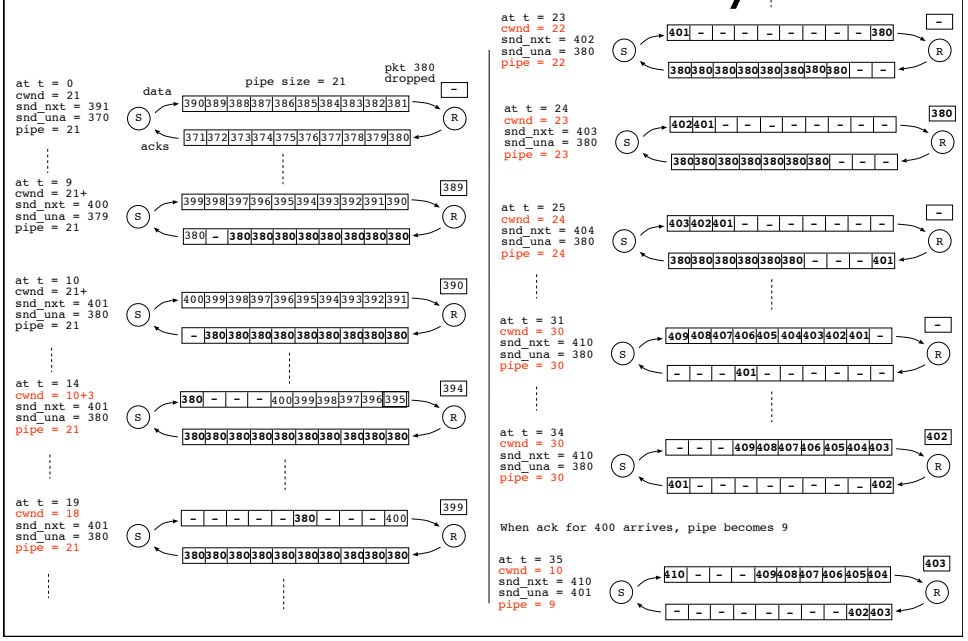
TCP Original



TCP Tahoe with Fast Retransmit



TCP Reno with Fast Recovery



Summary: Factors in TCP Performance

- RTT estimate
- RTO computation
- sender's sliding window (`wnd`)
- receiver's window (`rwnd`)
- congestion window (`cwnd`)
- slow-start threshold (`sstresh`)
- fast retransmit
- fast recovery

Summary of TCP Variants

Original TCP:

- loss recovery depends on RTO

TCP Tahoe:

- slow-start and linear increase
- 3 dupACKs as loss signal, but restart slow-start after fast retransmit

TCP Reno:

- fast recovery, i.e., consume half returning dupACKs before transmitting one new packet for each additional returning dupACKs
- on receiving a non-dupACK, resume linear-increase from half old `cwnd` value

TCP New Reno:

- implements *fast retransmit phase* whereby a non-dupACK takes TCP out of fast recovery only if the non-dupACK is ⩾ the sequence number sent out before detection of loss (`snd_high`)
- a non-dupACK that is < `snd_high` is called a partial ACK

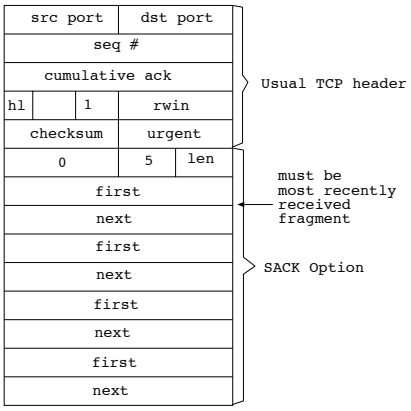
TCP SACK

Selective ACK (SACK) gives sender a better idea of which packets have been successfully delivered

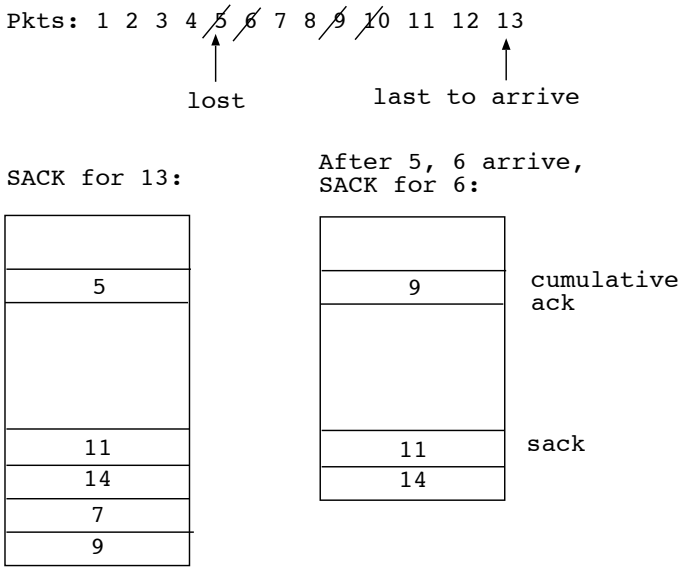
TCP does Selective Repeat instead of Go-Back-N

Added 2 TCP options:

1. SACK Permitted (option 4): carried only by SYN packets
2. SACK Option (option 5): NAK for missing segments



TCP SACK Example



TCP SACK Advantages

Decouples *when* and *how much* to send (congestion control) from *which* to send (loss recovery)

TCP SACK not yet widely deployed

Future use of SACK info:

- send new packet on first or second duplicate ACK
- tell sender when there will be no more returning ACK
- cancel previous decisions upon updated info
- construct a better count of outstanding segments

Effect of Asymmetry on TCP

Example of asymmetric networks:

- cable modem: 10 Mbps down, 512 Kbps up
- ADSL: 8 Mbps down, 1 Mbps up
- etc.

Fine for web surfing, but transfer can be slowed down by bandwidth asymmetry. How?

Slower bandwidth on reverse path stretches out ACKs (ACK dilation), resulting in slower throughput on the forward path

Effect of Asymmetry on TCP

Maximum transmission speed determined by "normalized bandwidth ratio":

$$\frac{\text{forward bandwidth/datasize}}{\text{reverse bandwidth/ACKsize}}$$

For example:

- 10 Mbps forward/100 Kbps reverse = bandwidth ratio of 100
- 1 KB data packet/40-byte ACK = data/ACK ratio of 25
- normalized bandwidth ratio = 100/25 = 4

⇒ if there are more than 1 ACK for every 4 packets, reverse bottleneck link will be saturated before forward bottleneck does