

LINK LAYER: BASIC TECHNIQUES

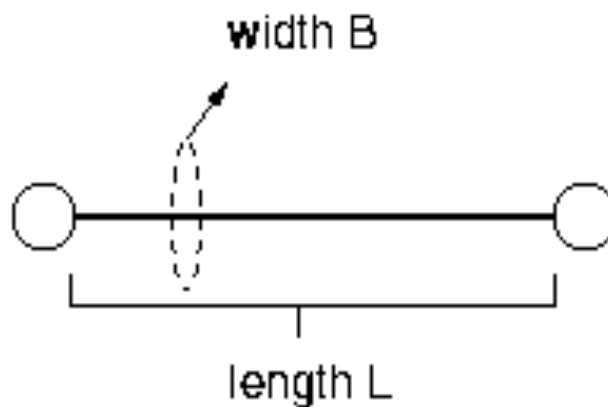
Performance Metrics

Link speed unit: bps

→ assume some physical layer that provides bandwidth
bps

→ e.g., TDMA, FDMA, OFDMA

Simplest case: point-to-point link



→ wired or wireless

→ bandwidth B bps, length L (unit: distance or time)

How long does it take to send bits?

→ completion time (unit: msec)

→ time elapsed between sending first bit and receiving last bit

- Single bit:

- $\approx L/\text{SOL}$ (lower bound)

- propagation delay or latency

- exact value depends on physical layer

- Multiple, say S , bits:

- $\approx L/\text{SOL} + S/B$

- latency + transmission time

- logic?

Examples:

- $S = 8000$ bits, $B = 1$ Gbps, $L = 2000$ miles

$$\begin{aligned} 2000/186000 + 8000/10^9 \\ \approx 10.75 + 0.008 = 10.758 \text{ msec} \end{aligned}$$

→ latency dominates transmission time

- $S = 8000$ bits, $B = 1$ Mbps, $L = 1$ mile

$$\begin{aligned} 1/186000 + 8000/10^6 \\ \approx 0.005 + 8 = 8.005 \text{ msec} \end{aligned}$$

→ transmission time dominates latency

Be aware of which dominates when estimating rough completion time

→ satellite links?

→ data center links?

Links are unreliable

→ bits can flip or unrecognizable

→ for many network applications: require reliable communication

Two approaches to achieving reliable data transmission over unreliable links

- resend missing/corrupted bits
 - reactive
- send data in redundant fashion
 - proactive

Reactive approach: ARQ (Automatic Repeat reQuest)

→ use retransmission with sequence numbers and timers

→ wired/wireless links

Proactive approach: FEC (forward error correction)

→ transmit redundant information

Pros and cons?

Example: simple methods for error detection and correction

→ single bit flip in 7-bit ASCII character

Error detection

→ add parity bit: odd or even

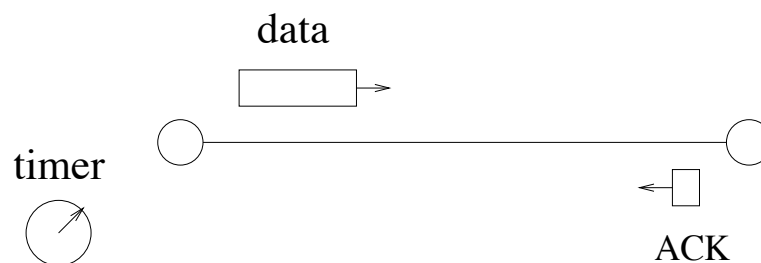
Error detection

→ 3-fold duplication and majority voting

Overhead?

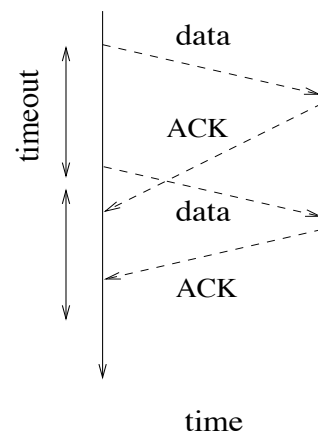
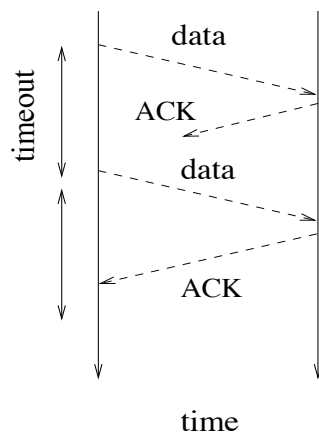
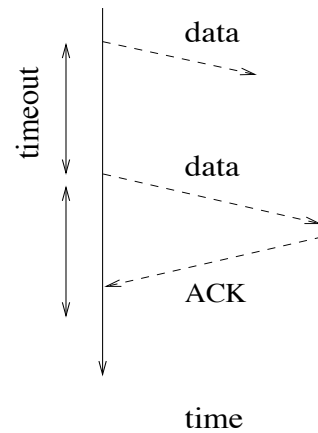
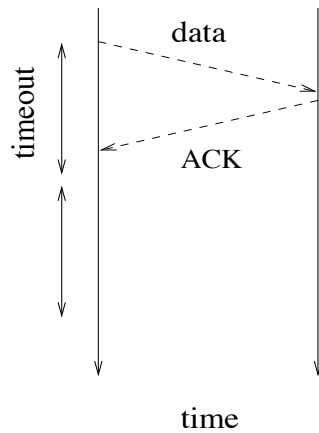
Components of ARQ:

- acknowledgment (ACK)
→ receiver: “I got it”
- sequence number
→ receiver: “I got packet 15”
- timer
→ estimate how long getting ACK should take
- buffer
→ sender keeps copy of data in buffer for resend



Simplest case of ARQ: stop-and-wait

→ handle one packet (i.e., frame) at a time



Issue of RTT (Round-Trip Time) and timer management:

- what is proper value of timer?
 - RTT estimation
- easier for single link
 - RTT is well-behaved
- more difficult for multi-hop path in internetwork
 - latency + queueing/buffering

Does stop-and-wait need sequence numbers?

Usefulness of stop-and-wait:

→ simple throughput formula

→ reliable throughput (bps)

$$= \text{data bits} / \text{total time}$$

→ pure data bits: do not count retransmissions

Example: file size $S = 8\text{M}$ bits, total bits sent = 10M bits, completion time = 100 msec

→ raw throughput = $10^7 / 0.1 = 100$ Mbps

→ reliable throughput = $8000000 / 0.1 = 80$ Mbps

→ ACK traffic not counted: unidirectional

Formally: stop-and-wait (reliable) throughput

- frame or packet size (bits)
- RTT (round-trip time)

$$\longrightarrow \text{throughput} = \text{frame size} / \text{RTT}$$

Throughput decreases with increasing RTT

→ far away is not good

→ property of many protocols including TCP

Another important problem: keep the “pipe” full

→ stop-and-wait: one frame per RTT

→ analogy: use one semi-truck to haul billions of tons of goods north to south on I65

Example: link bandwidth 1 Gbps, 10 msec RTT

- if frame size 1 KB, then throughput:
 - $1024 \times 8 / 0.01 = 0.8192$ Mbps
 - link utilization: less than 0.1 percent!
- “fatness of pipe”: delay-bandwidth product
 - $1 \text{ Gbps} \times 10 \text{ msec} = 10 \text{ Mbits}$
 - we only sent 8000 Kbits

The larger the bandwidth or delay, the worse the more underutilized.

How to increase link utilization?

Solution 1: increase frame size

→ can only go so far

- existing standards limit frame size
 - large frames can monopolize link
- bully problem

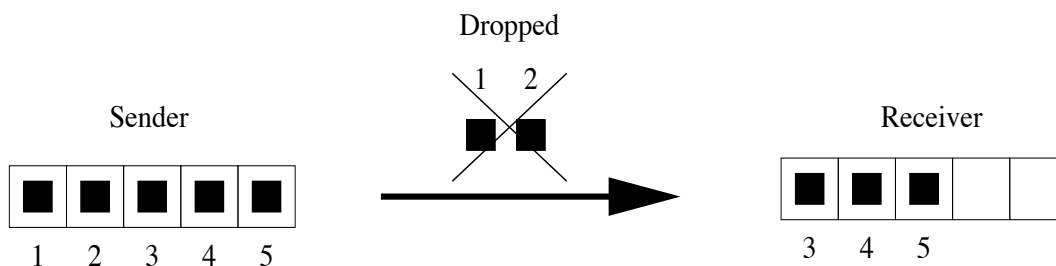
Solution 2: send block of frames to address frame size limit and bully problem

→ sliding window protocol

Sliding Window Protocol:

Issues to be addressed:

- shield application process from reliability management chore
 - exported semantics: continuous data stream
 - simple app abstraction: e.g., **read** system call
- both sender and receiver have limited buffer capacity
 - need to plug holes and flush buffer



Simple solution when receiver has infinite buffer capacity:

- sender keeps sending at maximum speed
- receiver informs sender of holes
 - “I’m missing this and that”
 - called negative ACK
- sender retransmits missing frames

What about positive ACK?

→ pros and cons

With finite buffers:

- issue of bookkeeping is more involved
 - key concern: correctness
- issue of not overflowing receiver buffer
 - sending too fast is not good
 - sending too slow is not good
 - send at just right pace

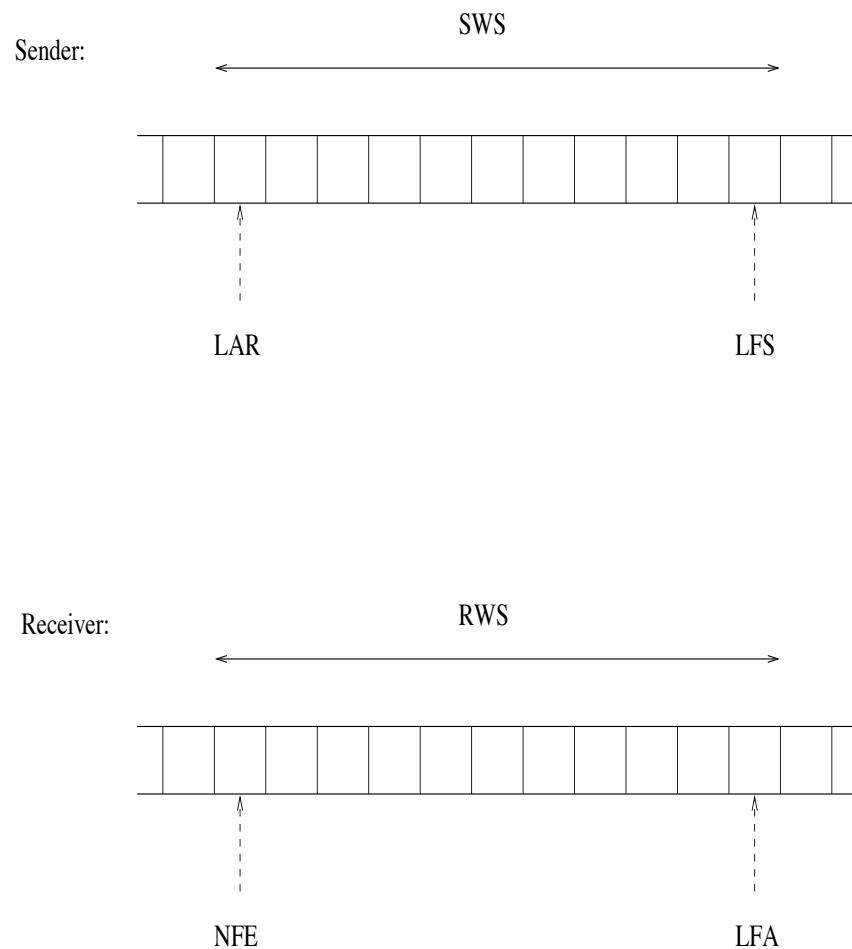
We will study pacing sending rate (i.e., congestion control) separately.

- complex problem
- significant impact on throughput

Sliding window operation with positive ACK:

→ view buffer as a window that slides over data

→ data unit: byte or frame



Notation:

- *SWS*: Sender Window Size (sender buffer size)
- *RWS*: Receiver Window Size (receiver buffer size)
- *LAR*: Last ACK Received
- *LFS*: Last Frame Sent
- *NFE*: Next Frame Expected
- *LFA*: Last Frame Acceptable

Assign sequence numbers to data: byte or frames.

→ IDs

Maintain invariants:

- $LFA - NFE + 1 \leq RWS$
- $LFS - LAR + 1 \leq SWS$

Sender:

- Receive ACK with sequence number X
- Forwind LAR to X
- Flush buffer up to (but not including) LAR
- Send up to $SWS - (LFS - LAR + 1)$
- Update LFS

Receiver:

- Receive data with sequence number Y
- Forward to (new) first hole & update NFE
→ NFE need not be $Y + 1$
- Send cumulative ACK (i.e., NFE)
- Flush buffer up to (but not including) NFE to application
- Update $LFA \leftarrow NFE + RWS - 1$

Important variation:

- receiver informs sender RWS
- sender never sends more than RWS

Sequence number wrap-around problem:

$$\text{SWS} < (\text{MaxSeqNum} + 1)/2$$

→ why?

→ recall special case: stop-and-wait

Real-world ARQ Performance: TCP

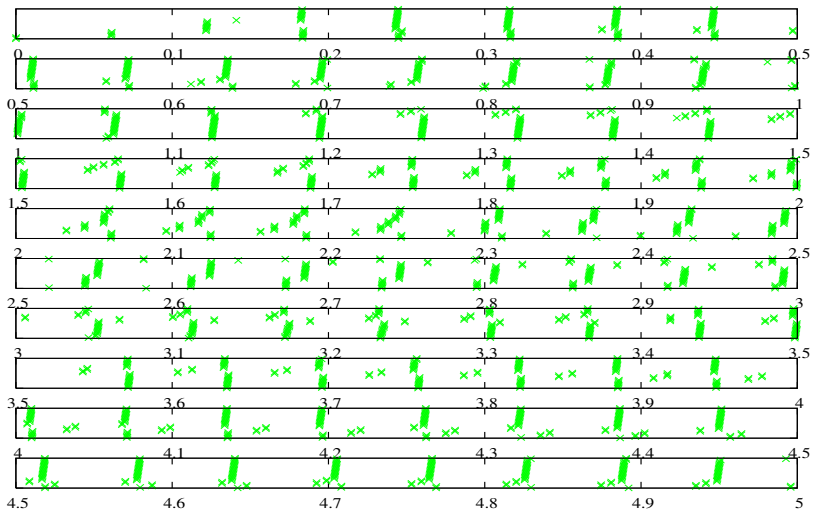
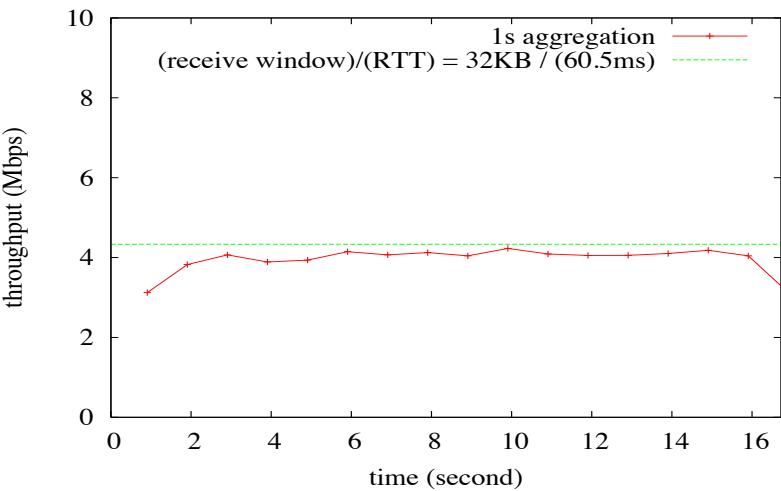
Ex.: Purdue $\rightarrow$ UCSD

- Purdue (NSL): web server
- UCSD: web client

```
traceroute to planetlab3.ucsd.edu (132.239.17.226), 30 hops max, 40 byte packets
 1  switch-lwsn2133-z1r11 (128.10.27.250)  0.483 ms  0.344 ms  0.362 ms
 2  lwsn-b143-c6506-01-tcom (128.10.127.251)  0.488 ms  0.489 ms  0.489 ms
 3  172.19.57.1 (172.19.57.1)  0.486 ms  0.488 ms  0.489 ms
 4  tel-210-m10i-01-campus.tcom.purdue.edu (192.5.40.54)  0.614 ms  0.617 ms  0.615 ms
 5  gigapop.tcom.purdue.edu (192.5.40.134)  1.743 ms  1.679 ms  1.727 ms
 6  * * *
 7  * * *
 8  * * *
 9  hpr-lax-hpr--nlr-packetnet.cenic.net (137.164.26.130)  56.919 ms  56.919 ms  57.658 ms
10  hpr-ucsd-10ge--lax-hpr.cenic.net (137.164.27.165)  60.326 ms  60.198 ms  60.196 ms
11  nodeb-720--ucsd-t320-gw-10ge.ucsd.edu (132.239.255.132)  60.326 ms  60.370 ms  75.130 ms
```

$\longrightarrow$ RTT $\approx$ 60.5 msec

$\longrightarrow$ receiver window size: 32 KB



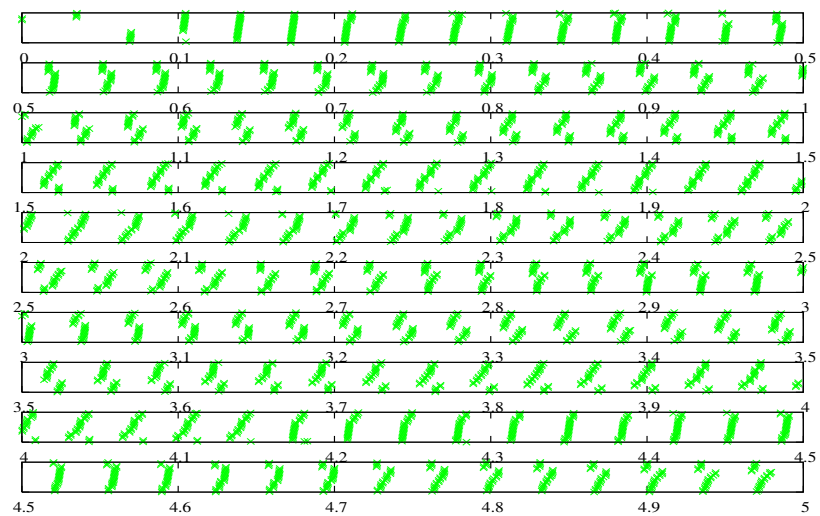
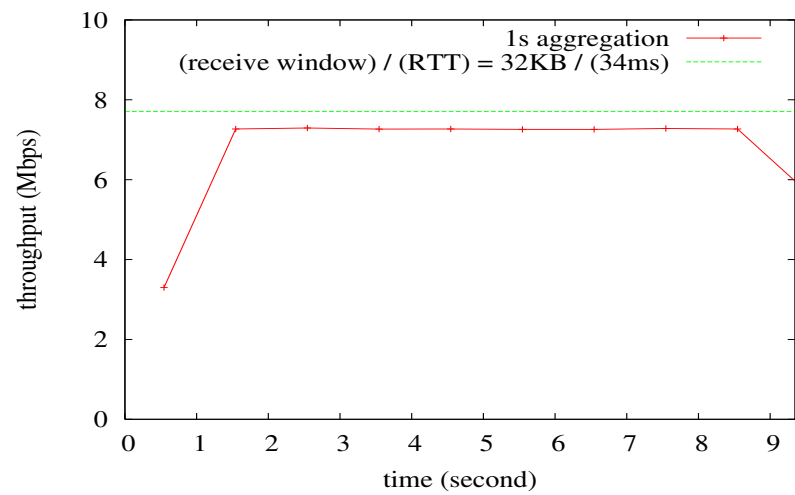
Ex.: Purdue $\rightarrow$ Rutgers

- Purdue: web server
- Rutgers: web client

```
traceroute to planetlab1.rutgers.edu (165.230.49.114), 30 hops max, 40 byte packets
 1  switch-lwsn2133-z1r11 (128.10.27.250)  12.336 ms  0.339 ms  0.362 ms
 2  lwsn-b143-c6506-01-tcom (128.10.127.251)  0.489 ms  0.491 ms  0.488 ms
 3  172.19.57.1 (172.19.57.1)  0.490 ms  0.488 ms  0.489 ms
 4  tel-210-m10i-01-campus.tcom.purdue.edu (192.5.40.54)  0.614 ms  0.615 ms  0.614 ms
 5  switch-data.tcom.purdue.edu (192.5.40.166)  2.864 ms  2.865 ms  2.864 ms
 6  abilene-ul.indiana.gigapop.net (192.12.206.249)  2.988 ms  13.608 ms  3.113 ms
 7  chinng-iplsng.abilene.ucaid.edu (198.32.8.76)  6.740 ms  6.875 ms  6.859 ms
 8  ge-0-0-0.10.rtr.chic.net.internet2.edu (64.57.28.1)  7.113 ms  6.975 ms  6.986 ms
 9  so-3-0-0.0.rtr.wash.net.internet2.edu (64.57.28.13)  29.349 ms  24.086 ms  23.626 ms
10  ge-1-0-0.418.rtr.chic.net.internet2.edu (64.57.28.10)  44.786 ms  28.822 ms  28.839 ms
11  local.internet2.magpi.net (216.27.100.53)  30.723 ms  30.818 ms  30.744 ms
12  phl-02-09.backbone.magpi.net (216.27.100.229)  31.045 ms  36.644 ms  30.839 ms
13  remote.njedge.magpi.net (216.27.98.42)  33.221 ms  33.021 ms  33.087 ms
14  er01-hill-ext.runet.rutgers.net (198.151.130.233)  33.229 ms  33.207 ms  33.217 ms
```

$\longrightarrow$ RTT $\approx$ 34 msec

$\longrightarrow$ receiver window size: 32 KB



Ex.: Purdue $\rightarrow$ Korea University (Seoul)

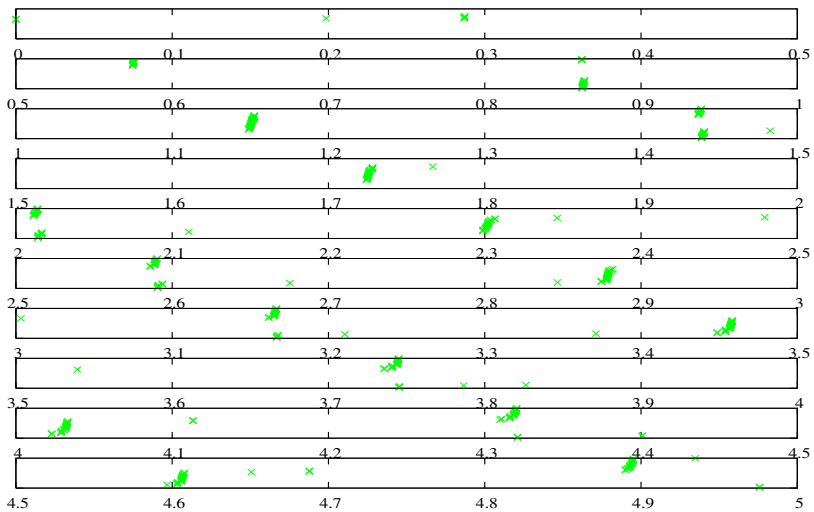
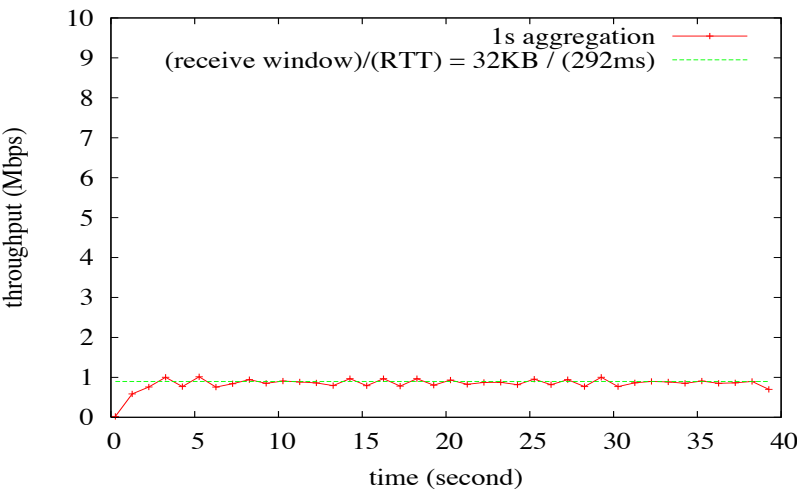
- Purdue: web server
- KU: web client

```
1  switch-lwsn2133-z1r11 (128.10.27.250)  0.513 ms  10.061 ms  0.358 ms
2  lwsn-b143-c6506-01-tcom (128.10.127.251)  0.487 ms  0.476 ms  0.364 ms
3  172.19.57.1 (172.19.57.1)  0.489 ms  0.475 ms  0.490 ms
4  tel-210-m10i-01-campus.tcom.purdue.edu (192.5.40.54)  0.613 ms  0.600 ms  0.614 ms
5  switch-data.tcom.purdue.edu (192.5.40.166)  7.982 ms  7.969 ms  14.596 ms
6  abilene-ul.indiana.gigapop.net (192.12.206.249)  8.977 ms  7.721 ms  6.857 ms
7  kscopyng-iplsng.abilene.ucaid.edu (198.32.8.81)  36.860 ms  25.873 ms  29.075 ms
8  dnvrng-kscopyng.abilene.ucaid.edu (198.32.8.13)  24.218 ms  23.125 ms  36.317 ms
9  snvang-dnvrng.abilene.ucaid.edu (198.32.8.1)  47.815 ms  78.440 ms  54.048 ms
10 losang-snvang.abilene.ucaid.edu (198.32.8.94)  55.080 ms  55.131 ms  60.674 ms
11 transpac-1-lo-jmb-702.lsanca.pacificwave.net (207.231.240.136)  55.165 ms  55.212 ms  59.1
12 tokyo-losa-tp2.transpac2.net (192.203.116.146)  175.068 ms  170.832 ms  170.444 ms
13 tyo-gate1.jp.apan.net (203.181.248.249)  170.488 ms  170.893 ms  171.818 ms
14 sg-so-02-622m.bb-v4.noc.tein2.net (202.179.249.5)  277.150 ms  275.966 ms  276.136 ms
15 kr.pr-v4.noc.tein2.net (202.179.249.18)  278.422 ms  276.486 ms  280.132 ms
16 61.252.48.182 (61.252.48.182)  276.170 ms  279.606 ms  279.421 ms
17 202.30.43.45 (202.30.43.45)  271.663 ms  269.492 ms  268.761 ms
18 honeung13-seoul.kreonet.net (134.75.120.2)  269.781 ms  269.913 ms  273.516 ms
19 203.241.173.74 (203.241.173.74)  272.663 ms  278.774 ms  270.902 ms
```

$\longrightarrow$ RTT $\approx$ 292 msec

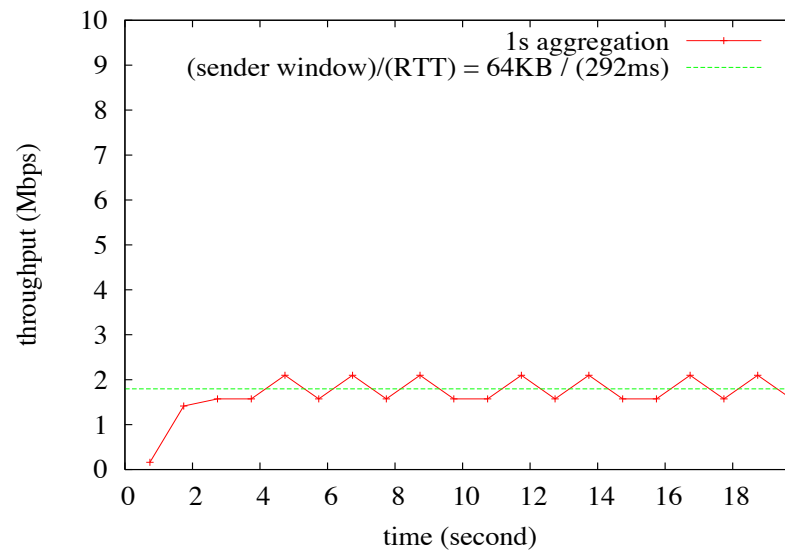
$\longrightarrow$ long route to Korea (via Singapore)

$\longrightarrow$ receiver window size: 32 KB



Increase receiver window size: 128 KB

→ 4-fold increase

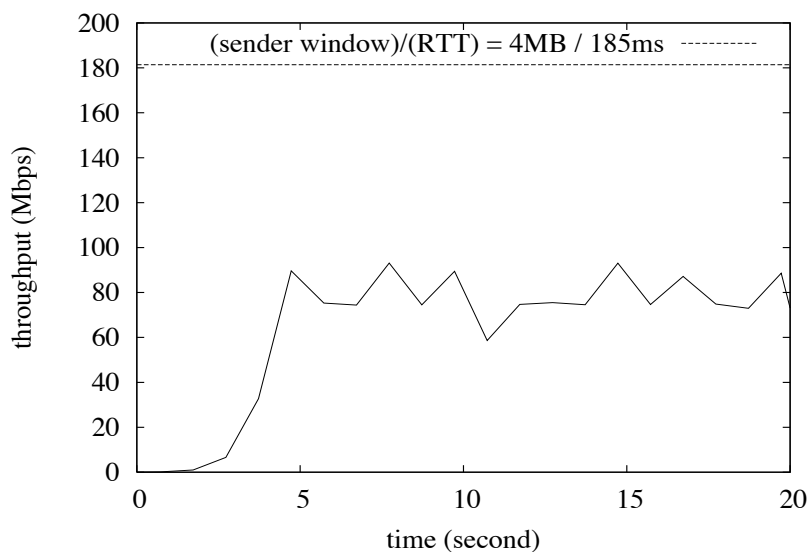


→ why only 2-fold throughput increase?

Increase receiver window size: 8 MB

→ also increase sender buffer size to 4 MB

→ $\text{RTT} \approx 185 \text{ msec}$ (short route to Korea)



→ around 90 Mbps

→ download time for 10 MB file?

→ can be confused with DoS (denial-of-service) attack

→ why less than 180 Mbps?