Link Layer: Basic Techniques

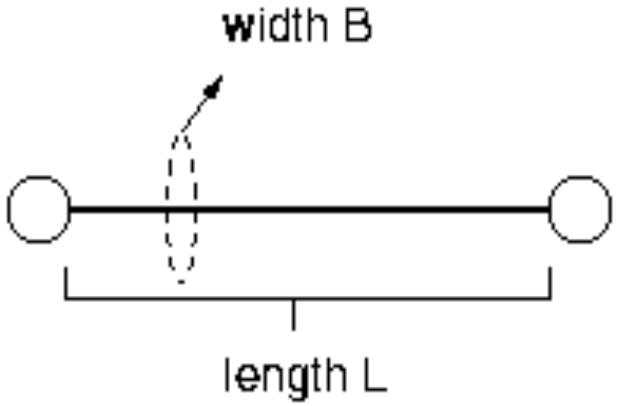
Performance Metrics

Link speed unit: bps

→ assume some physical layer that provides bandwidth bps

→ e.g., TDMA, FDMA, OFDMA with 64-QAM

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

Examples:

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

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

  $\rightarrow$ latency dominates transmission time

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

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

  $\rightarrow$ transmission time dominates latency

Be aware of which dominates when estimating rough completion time

$\rightarrow$ satellite links?

$\rightarrow$ 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

→ e.g., send two additional copies

Pros and cons?

Components of ARQ:

- acknowledgment (ACK)

  → receiver: "I got it"

- sequence number

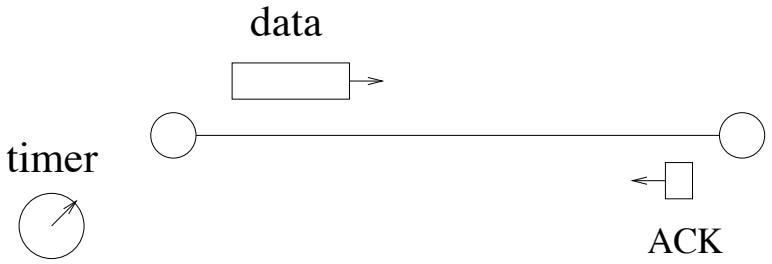
  → receiver: "I got packet 15"

- timer

  → estimate how long getting ACK should take

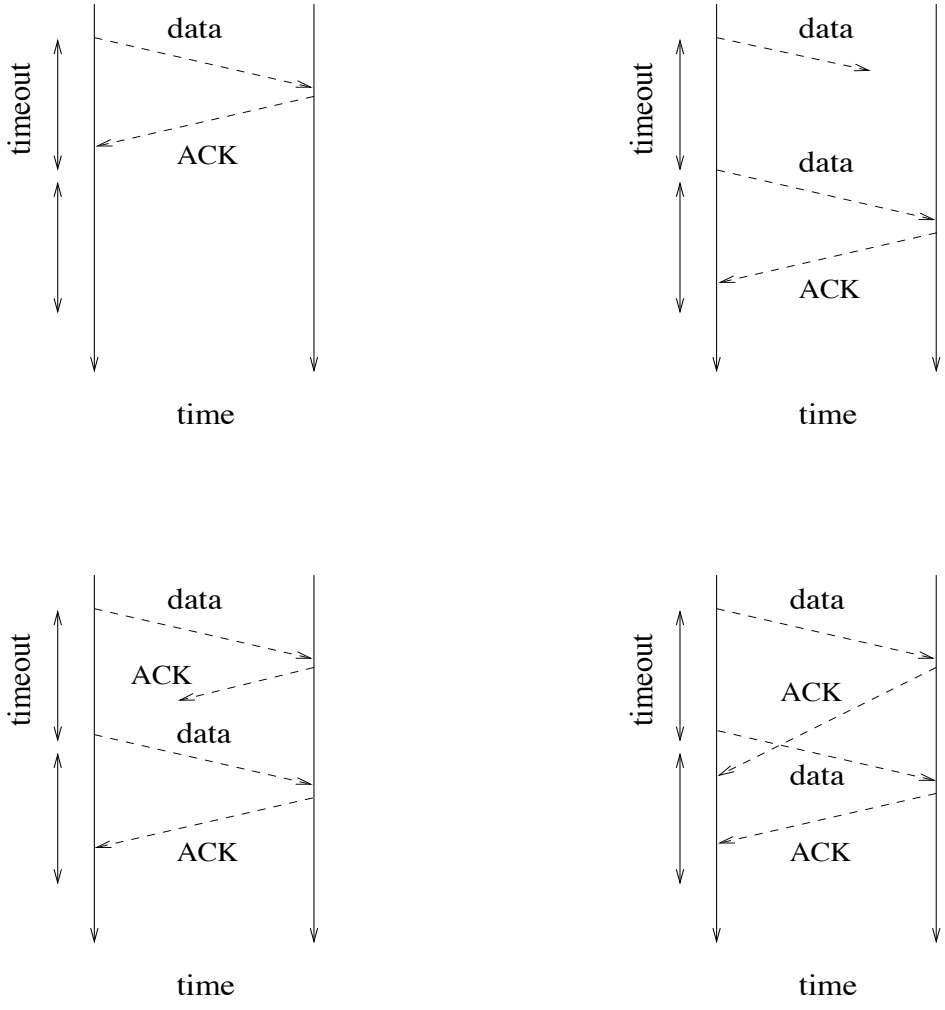
- buffer

  → 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?

  $\rightarrow$ RTT estimation

- easier for single link

  $\rightarrow$ RTT is well-behaved

- more difficult for multi-hop path in internetwork

  $\rightarrow$ latency + queueing/buffering

Does stop-and-wait need sequence numbers?

Usefulness of stop-and-wait:

→ simple throughput formula

→ reliable throughput (bps)

= data bits / total time

→ pure data bits: do not count retransmissions

Example: file size S = 8M 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 size (bits)
- RTT (round-trip time)

$\longrightarrow$ throughput = frame size / RTT

Throughput decreases with increasing RTT

$\rightarrow$ far away is not good

$\rightarrow$ general property of many protocols including TCP

Another important problem: keep the "pipe" full

$\rightarrow$ stop-and-wait: one frame per RTT

$\rightarrow$ what if bandwidth and/or RTT are very large?

Example: link bandwidth 1 Gbps, 10 msec RTT

- if frame size 1 KB, then throughput:

  $\rightarrow 1024 \times 8/0.01 = 0.8192$ Mbps

  $\rightarrow$ link utilization: less than 0.1 percent!

- "fatness of pipe": delay-bandwidth product

  $\rightarrow 1$ Gbps $\times$ 10 msec $= 10$ Mbits

  $\rightarrow$ 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:

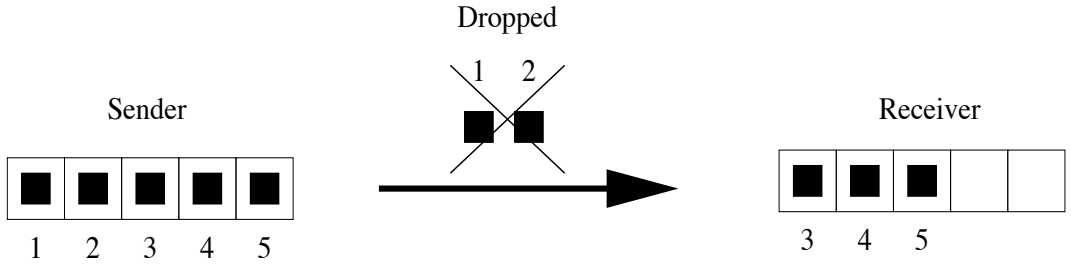
- shield application process from reliability management chore

  → 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

  $\rightarrow$ "I'm missing this and that"

  $\rightarrow$ called negative ACK
- sender retransmits missing frames

May be viable for "smaller" files but not feasible in general.

What about positive ACK?

$\longrightarrow$ pros and cons

With finite buffers:

- issue of bookkeeping is more involved

  → key concern: correctness

- issue of not overflowing receiver buffer is more involved

  → sending too fast is not good

  → sending too slow is not good

  → send at just right pace

  → flow control

We will study flow and congestion control separately.

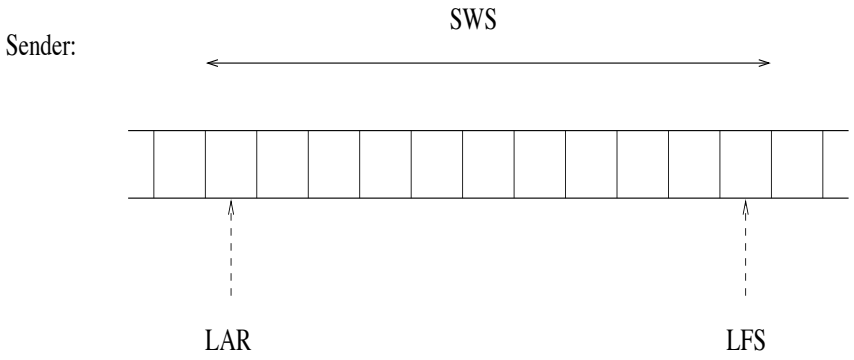
→ complex problem

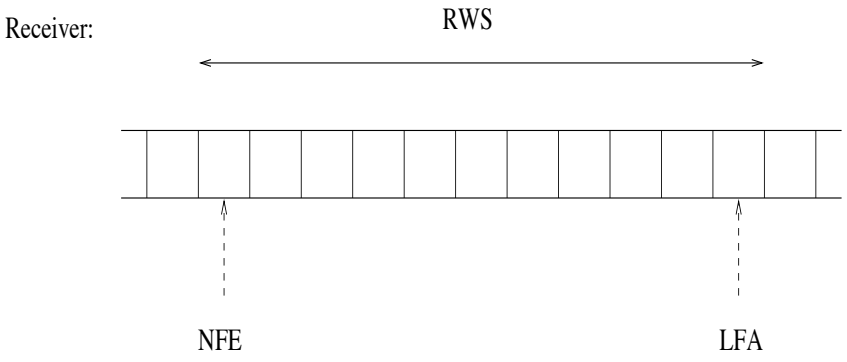
→ significant impact on throughput

Sliding window operation with positive ACK:

$\rightarrow$ view buffer as a window that slides over data

$\rightarrow$ 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.

$\longrightarrow$ IDs

Maintain invariants:

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

Sender:

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

Receiver:

- Receive data with sequence number Y
- Forwind to (new) first hole & update NFE

  $\rightarrow$ 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:

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

$\longrightarrow$ why?

$\longrightarrow$ recall special case: stop-and-wait

Real-world ARQ Performance: TCP

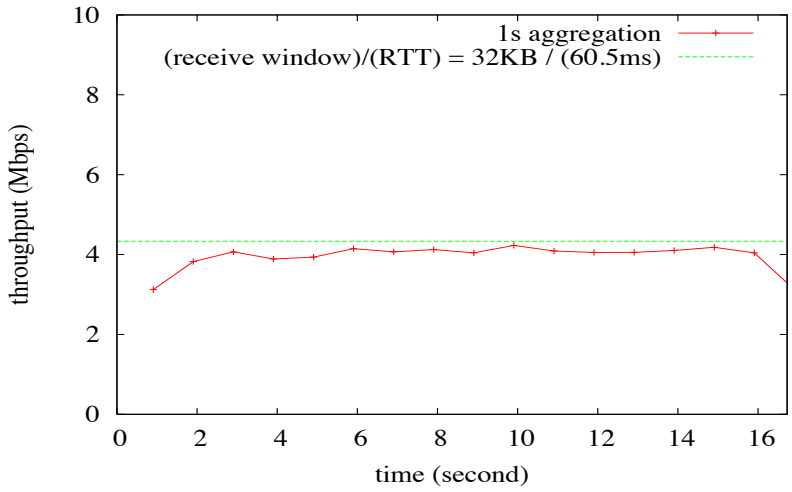
Ex.: Purdue → UCSD

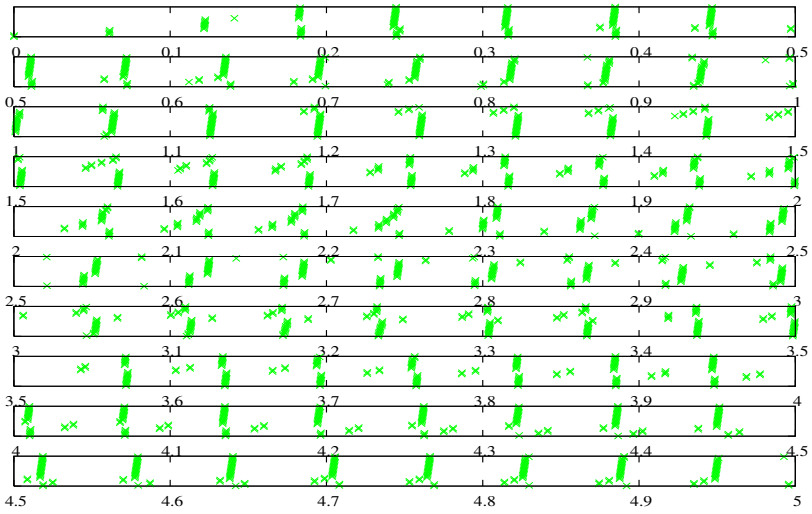
- Purdue (NSL): web server
- UCSD: web client

```
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-packenet.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


1s aggregation
(receive window)/(RTT) = 32KB / (60.5ms)
throughput (Mbps)
time (second)
0
2
4
6
8
10
12
14
16



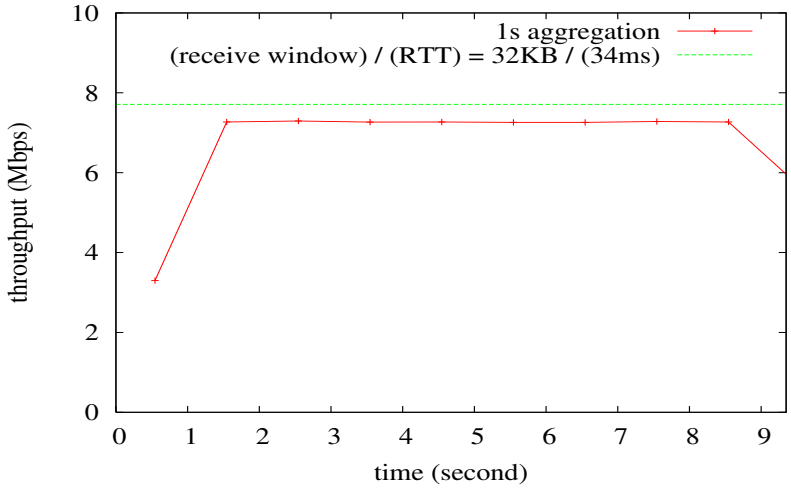
Ex.: Purdue → Rutgers

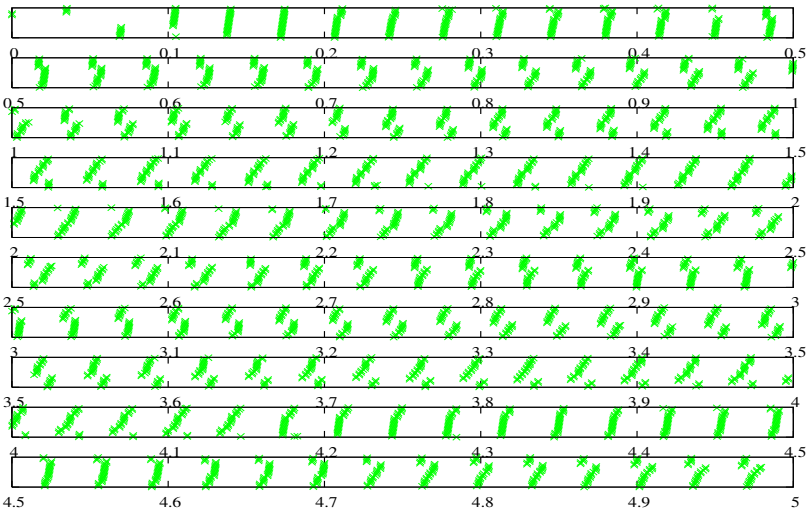
- Purdue: web server
- Rutgers: web client

```
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


1s aggregation
(receive window) / (RTT) = 32KB / (34ms)
throughput (Mbps)
time (second)
0
2
4
6
8
10
0
1
2
3
4
5
6
7
8
9


0
0.1
0.2
0.3
0.4
0.5
0.5
0.6
0.7
0.8
0.9
1
1
1.1
1.2
1.3
1.4
1.5
1.5
1.6
1.7
1.8
1.9
2
2
2.1
2.2
2.3
2.4
2.5
2.5
2.6
2.7
2.8
2.9
3
3
3.1
3.2
3.3
3.4
3.5
3.5
3.6
3.7
3.8
3.9
4
4
4.1
4.2
4.3
4.4
4.5
4.5
4.6
4.7
4.8
4.9
5

Ex.: Purdue → Korea University (Seoul)

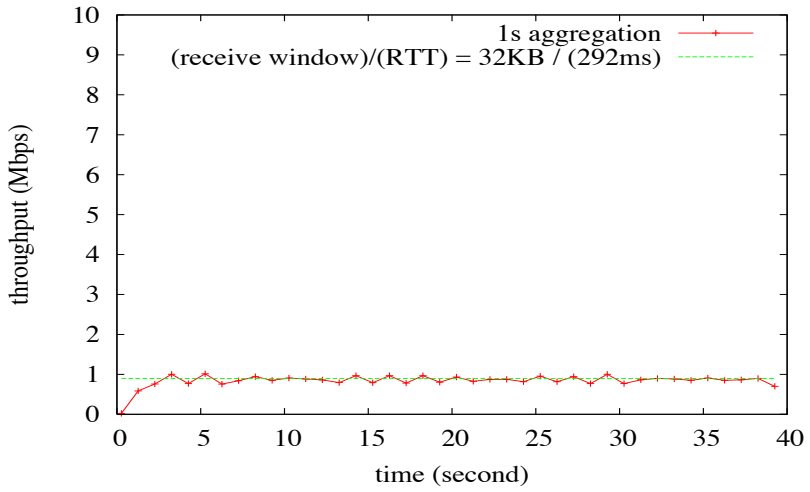
- Purdue: web server
- KU: web client

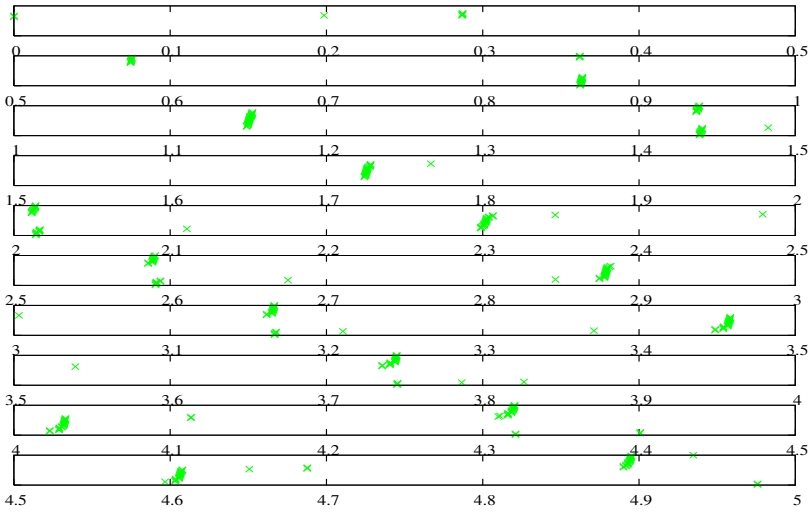
```
 1  switch-lwsn2133-z1r11 (128.10.27.250)  0.513 ms  10.061 ms  0.358 ms
 2  lwsn-b143-c6506-01-tcom (128.10.127.251)  0.487 ms  0.476 ms  0.364 ms
 3  172.19.57.1 (172.19.57.1)  0.489 ms  0.475 ms  0.490 ms
 4  tel-210-m10i-01-campus.tcom.purdue.edu (192.5.40.54)  0.613 ms  0.600 ms  0.614 ms
 5  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  kscyng-iplsng.abilene.ucaid.edu (198.32.8.81)  36.860 ms  25.873 ms  29.075 ms
 8  dnvrng-kscyng.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  honeungl3-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)

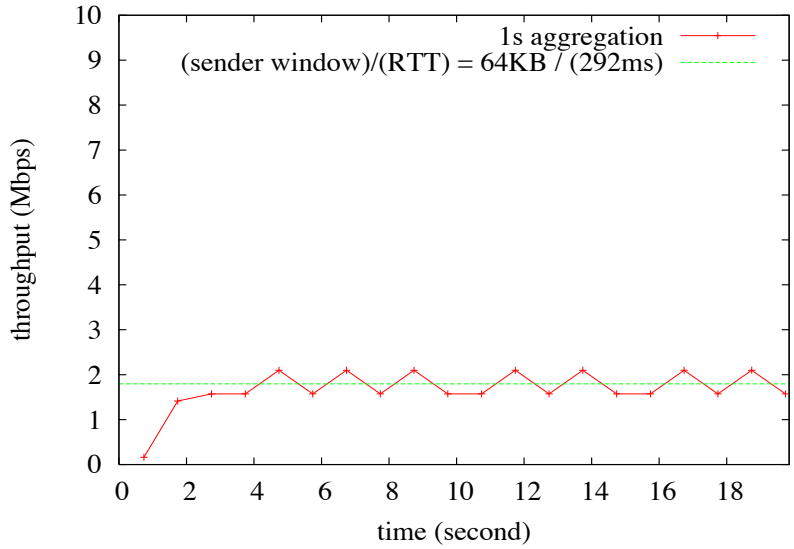
$\longrightarrow$ receiver window size: 32 KB


1s aggregation
(receive window)/(RTT) = 32KB / (292ms)
throughput (Mbps)
time (second)
0
1
2
3
4
5
6
7
8
9
10
0
5
10
15
20
25
30
35
40



Increase receiver window size: 128 KB

$\longrightarrow$ 4-fold increase

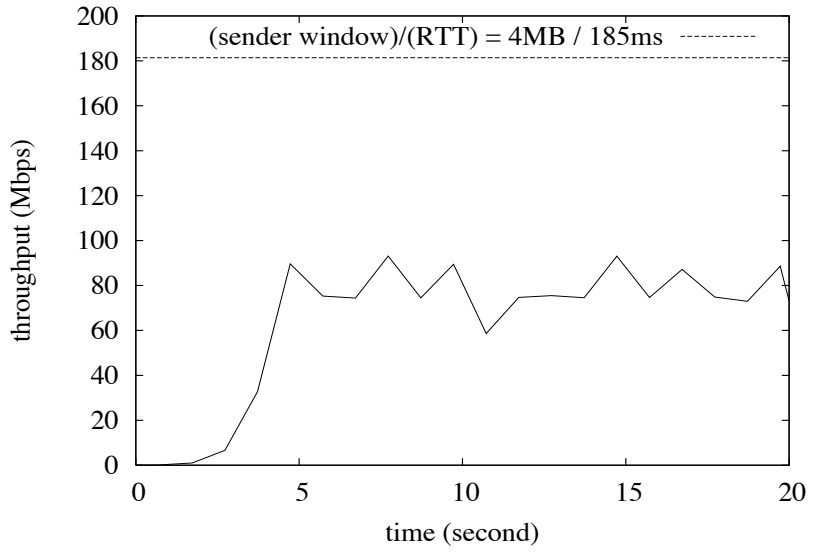


$\longrightarrow$ why only 2-fold throughput increase?

Increase receiver window size: 8 MB

$\longrightarrow$ also increase sender buffer size to 4 MB

$\longrightarrow$ RTT $\approx$ 185 msec (short route to Korea)



$\longrightarrow$ around 90 Mbps

$\longrightarrow$ download time for 10 MB file?

$\longrightarrow$ can be confused with DoS (denial-of-service) attack

$\longrightarrow$ why less than 180 Mbps?