Internet Programming & Protocols Lecture 17

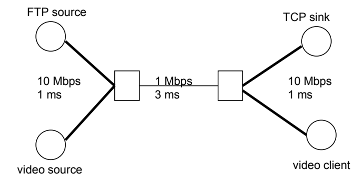
ns

OPnet

TCP flavors: Scalable TCP, HS TCP, BI-TCP

ns trace-driven simulation

- Text example 4.2, TCP competing with mpeg trace of Star Wars
  - ~dunigan/ipp05/ns/chap4-2.tcl
  - Variable bit-rate trace of video source is big file (not in my directory)
    - (time, packet length)
  - Report datarate and drops for TCP Tahoe and video stream
  - Generate bandwidth plots of UDP and TCP with record procedure
  - Use awk –f e2e.awk out.all | xgraph –m to generate RTT vs time



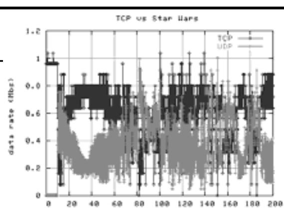
chap4-2.tcl

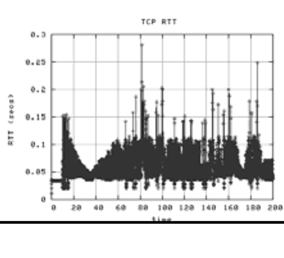
```
# create UDP source sink
set udp0 [new Agent/UDP]
$ns attach-agent $n2 $udp0
set udpsink0 [new Agent/Null]
$ns attach-agent $n3 $udpsink0
$ns connect $udp0 $udpsink0

# tracefile object
set tfile [new Tracefile]
$tfile filename starwars.nsformat
set trace0 [new Application/Traffic/Trace]
$trace0 attach-tracefile $tfile
$trace0 attach-agent $udp0
```


chap4-2.tcl

```
set mybytes 0
set ubytes 0

proc record {} {
        global ns tcp0 tcpsink0 f2 mybytes f1 sink3 ubytes

        set delta 0.1
        set now [$ns now]
         set tombytes [$tcp0 set ndatabytes_ ]
         set bw [expr $tombytes - $mybytes]
         puts $f2 "$now [expr $bw/$delta*8/1000000]"
         set mybytes $tombytes

         set tombytes [$sink3 set bytes_ ]
         set bw [expr $tombytes - $ubytes]
         puts $f1 "$now [expr $bw/$delta*8/1000000]"
         set ubytes $tombytes

                 #Re-schedule the procedure
         $ns at [expr $now+$delta] "record"
}
```


chap4-2.tcl





- UDP has no congestion control, so TCP gets bandwidth only when video stream is not consuming as much.
- Both streams are dropping packets

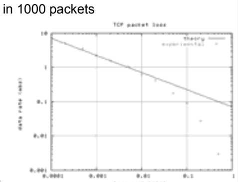
ns loss models

- UDP agents includes, CBR, Pareto, Exponential, and trace-driven
- You can also associate a loss model with a link
  - droplist

```
set droplist { 100 102 104 }
set lossylink_ [$ns link $n0 $n1]
set loss_module [new ErrorModel/List]
$loss_module droplist $droplist
$lossylink_ errormodule $loss_module
```

  - Probabilistic p= 0.001 drop on the average 1 in 1000 packets

```
set lossrate 0.001
set lossylink_ [$ns link $n0 $n1]
set loss_module [new ErrorModel]
$loss_module set rate_ $lossrate
set u [new RandomVariable/Uniform]
set rng [new RNG]
$rng seed 57643
$u use-rng $rng
$loss_module ranvar $u
$lossylink_ errormodule $loss_module
```



- Use ns random numbers to select start times

```
set sec [expr int([$rng uniform 0.1 [expr $endtime/20]])]
set frac [expr int([$rng uniform 0 25])]
set starttime $sec.$frac
```

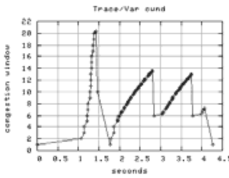
More tracing and monitoring in ns

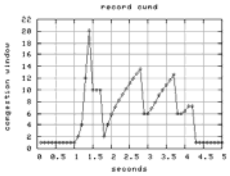
- Trace/Var
  - Record new values of ns variable every time it changes

```
set tf [open cwnd.tr w]
set tracer [new Trace/Var]
$tracer attach $tf
$tcp trace cwnd_ $tracer

  output file
f t3.090184 a_o83 ncwnd_ v6.79157
f t3.109195 a_o83 ncwnd_ v6.93882
f t3.118795 a_o83 ncwnd_ v7.08293
```

  - record procedure samples good enough
- Event trace (TCP state info)
  - Inserted in trace-all file
  - $ns eventtrace-all [$file]

```
E 1.766513 0 3 TCP TIMEOUT 1 54 10
E 1.766513 0 3 TCP SLOW_START 1 54 1
E 2.832301 0 3 TCP NEWRENO_FAST_RETX 1 149 13
E 2.837195 0 3 TCP NEWRENO_FAST_RECOVERY 1 149 6
      src dst  event            fid seq# cwnd
```





Monitoring queues in ns

- Tracing queue variables at specified interval

```
set qmon [$ns monitor-queue $n2 $n3 [open qm.out w] 0.1];
[$ns link $n2 $n3] queue-sample-timeout;
File:  1.3 2 3 1040.0 1.0 5 2 2 4120 1080 2000
time src dst avrgB  avrgpkts arrivals depart drops barriv bdepart bdrops
```

- Monitoring a queue
  - Snapshot of queue activity with your record or finish procedure
  - Variables **pdrops_ pdepartures_ parrivals_ bdrops_ bdepartures_ barrivals_**

```
set qmon [$ns monitor-queue $n0 $n1 1 2]
set curr_qsize [$qmon set size_]
puts “drops [$qmon set pdrops_]  "
```

- Monitoring a flow
  - If you need to know which flows are experiencing drops
  - Need to set flow ids $tcp set fid_ 1

```
set fm [$ns makeflowmon Fid]
$ns attach-fmon [$ns link $r1 $r2] $fm 0
foreach f  [$fm flows] {
        puts " flow [$f set flowid_]  drops: [$f set pdrops_]"
}
```

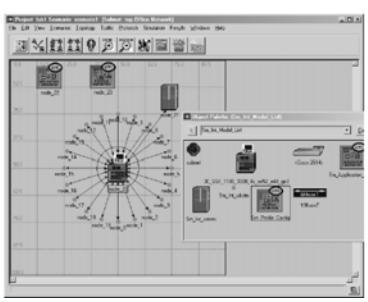

ns summary

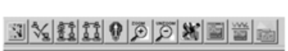
- ns popular in the network research community
  - Free
  - Considerable testing of ns TCP implementations (credibility)
  - Easy to add new variations
    - Add a little C++, adjust the Makefile and defaults.tcl
  - Other features: routing, LAN, wireless, multicast
- Awkward to setup topologies
  - No drag & drop
  - C++ and Tcl pretty ugly
- Not suitable for huge topologies
  - Though scripting loops are better than doing 5000 drag & drops
- All simulations are going to be slow for lots of nodes and lots of packets
  - Need parallelism … open research
- Alternatives
  - ssfnet – written in java (credibility -- validation of TCP flavors?)
  - OPNET

OPNET

- Industrial strength simulator – good credibility
- Big $’s though “free” for university/researchers
  - Technical support vs “ns mailing lists”
- Nice GUI for topology design (drag & drop)
- Presentation of results more intuitive
- More efficient (faster?) than ns, better for larger simulations
- LAN simulations let you drag & drop particular vendor router/switches
- Some TCP customizations supported (C like)

OPNET GUI





Theory, experiment, simulation

- **Live internet tests**
  - See results in ultimate environment
  - Real TCP stacks/OS, traffic
  - Vary time and host/paths
  - Worry about impact?
- **Test beds**
  - Controlled traffic, but real OS
  - Usually LAN based, no queuing
  - Repeatable
  - Not very good for cross-traffic
- **Emulators**
  - Same as testbed
  - Plus control delay, loss, data rates, dup’s, out-of-order
  - Easy to reconfigure
- **Need tools to probe and measure**

- **Simulations**
  - Easily reconfigured
    - Complex topology
    - Vary TCP flavor
  - Repeatable
  - Detailed feedback/instrumentation
  - Add delay, loss, cross-traffic, queues
  - Randomness for confidence
  - Investigate “new” networks/protocols
  - cheap
  - Can be slow
  - Not real TCP

TCP flavors

- Some modifications to TCP were to make it more net friendly
  - Slow-start, AIMD, expo. timeouts, delayed ACKs, Nagle
- Some optimizations to make a TCP flow faster
  - Fast recovery, fast retransmit, SACK, FACK
- Initial evolution: TCP Tahoe, Reno, NewReno, SACK, FACK
  - Use packet loss to detect congestion and probe for bandwidth
  - AIMD(1,0.5) to backoff quickly and slowly increase speed
- Slow-start options for high speed
- TCP accelerants for long, fat nets
  - HS TCP, Scalable TCP, BI-TCP
- TCP variants to avoid packet loss
  - Vegas/FAST
  - Use bandwidth and delay estimates to select cwnd/ssthresh
    - TCP Westwood

slow-start

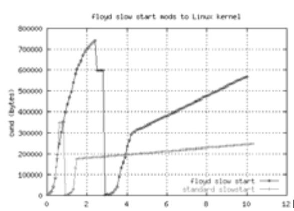
- Motivation: restore ACK clocking
  - When: intial start up, after packet loss, after idle period
  - Avoid blast of W packets (full window)
  - cwnd ← 1
- RFC's suggest a TCP stack can choose to start with cwnd = 4
  - Speeds startup (2 less RTT's), get ACK clocking going quicker
  - Acceptable blast (ns windowInitOption_ windowInit_)
- Open research, TCP quick-start, faster rate startup with router "approval"
- For long, fat nets (high speed, high delay), Floyd suggests slowing slow-start after some number of RTT's
  - Slow-start could be injecting thousands of packets into net at NIC speed
  - We had certain paths that often experienced packet loss in slow-start ☹
  - NOTE: TCP Vegas also has a slow-start moderator parameter (λ)
  - NOTE: slow-start can overshoot "available bandwidth" by a factor of 2
    - Linear phase overshoot by only 1 segment (or k segments if "virtual MSS")

Limited slow-start for large congestion windows (Floyd)

- RFC 3742, new variable max_ssthresh (typically 100)
  - Normal slow-start at first
  - Slow-start increment decreases with growing cwnd
- For each arriving ACK in slow-start:

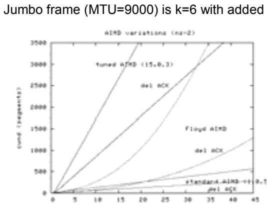
```
if (cwnd <= max_ssthresh) cwnd += MSS;  /* standard slow-start */
 else {
   K = int(cwnd/(0.5 *  max_ssthresh));
   cwnd += int(MSS/K);
}
```

Anecdotal evidence of improved slow-start between ORNL and NERSC



TCP for long Fat Networks (LFN)

- TCP linear recovery on paths with high bandwidth and long RTT
  - Takes cwnd/2 RTT's and slope of line is MSS/RTT^2 /sec bits/sec
  - 10 Gig, 100 ms RTT needs window of 83,333 segments
    - Recovering from cwnd/2 takes 4,166 seconds – over an hour!
- Some not so TCP-friendly proposals to speed recovery for LFNs
  - Floyd's HS TCP (a,b) a function of current cwnd (table lookup)
  - Scalable TCP (1%,1/8), increase cwnd by 1% each ACK
  - Virtual MSS (k, ½), increase cwnd by k/cwnd for each ACK
    - Jumbo frame (MTU=9000) is k=6 with added benefits

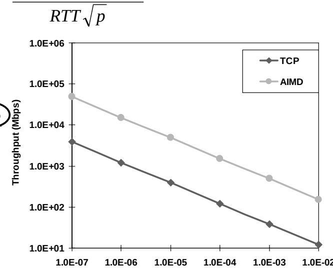


Easy to experiment with AIMD in ns
$tcp set decrease_num_ 0.875
$tcp set increase_num_ 32

Response Function of AIMD(a,b)

- Recall our inverse sqrt p law: $\dfrac{MSS\sqrt{a(2-b)/2b}}{RTT\sqrt{p}}$
- TCP: $R = \dfrac{MSS}{RTT}\dfrac{1.2}{p^{0.5}}$
- AIMD(32,1/8): $R = \dfrac{MSS}{RTT}\dfrac{15.5}{p^{0.5}}$

The throughput of AIMD is always about 13 times larger than that of TCP



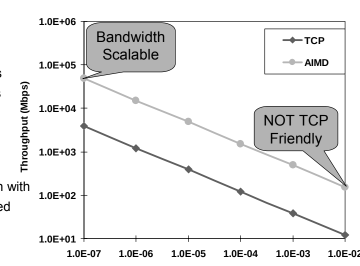
Properties of AIMD

- Bandwidth Scalability

  The ability to achieve 10Gbps with a reasonable packet loss probability

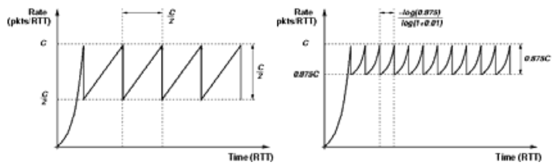
- TCP-Friendliness

  The ability to share bandwidth with TCP connections on low-speed networks



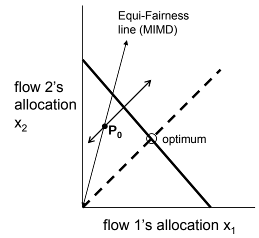
Scalable TCP

- Aggressive MIMD for high speed nets (Tom Kelley)
  - Loss event: cwnd = 0.875 * cwnd (1/8 instead of ½)
  - With each ACK: cwnd = cwnd + 0.01 (when cwnd > 16)



traditional TCP AIMD   scalable TCP

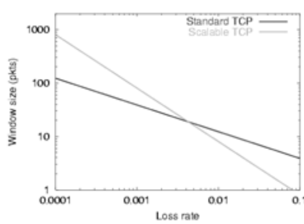
MIMD and scalable TCP



- Scalable TCP uses multiplicative decrease (1/8) and multiplicative increase (0.01) → MIMD
- Recall from our control system model, MIMD does not converge! (green line)

Scalable TCP response curve

- Response curve for TCP: $w = c/\sqrt{p}$ for SCTP: $w = k/p$
  - STCP has a linear response function, scale-invariant
- Use standard TCP if cwnd < 16



Scalable TCP

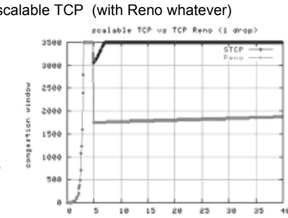
- For RTT = 200 ms, k=0.01, (note: scale invariant)

recovery time

Rate	Standard TCP	Scalable TCP
1Mbps	1.7s	2.7s
10Mbps	17s	2.7s
100Mbps	2mins	2.7s
1Gbps	28mins	2.7s
10Gbps	4hrs 43mins	2.7s

- I have modified ns to support scalable TCP (with Reno whatever)

$tcp set windowOption_ 9
$tcp set decrease_num_ 0.875
$tcp set k_parameter_ 0.01



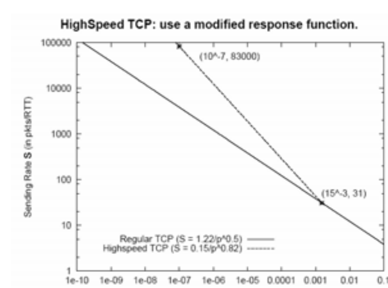
Note: (1) big window with 140 ms RTT, (2) window cut by 1/8, (3) fast recovery while 1/cwnd takes forever

Is it friendly, stable, fair?

High Speed TCP (HS TCP)

- Floyd, '02, RFC 3649 (experimental)
- TCP linear recovery on paths with high bandwidth and long RTT
  - Takes cwnd/2 RTT's and slope of line is MSS/RTT^2 /sec bits/sec
  - 10 Gig, 100 ms RTT needs window of 83,333 segments
  - From inverse square root p law, if you want to sustain a data rate of 10 gbs over a 100 ms RTT path, your loss rate must be less than 10^{-14}
- Floyd proposes a modified response function that requires a more tractable probability of 10^{-7} for 83,000 segment window
  - Standard TCP sending rate (S) in segments/RTT $S = 1.22/(p^{0.5})$
  - HS TCP $S = 0.15/(p^{0.82})$
- At low loss probabilities and big windows, you want TCP to scale
- When loss probability is high (10^{-3} or larger), you want to be TCP friendly because net is probably congested
  - HS TCP, STCP, and BI-TCP have low window threshold where if cwnd < low_win then use standard TCP AIMD (1, ½)
  - Low window threshold is 38 segments for HS TCP

HS TCP



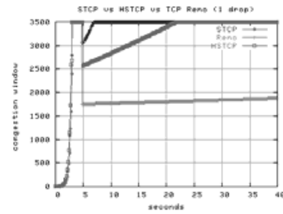
HS TCP implementation

- To get the AIMD(a,b) for HS TCP, need to solve the exponential equation for current cwnd, w
- Rather than have the kernel solve that equation during packet processing, we use a table lookup based on current congestion window, w, to get decrease factor and increase factor.
- Example, if cwnd is 1058, then reduce by 1/3 and increment is 8
- If cwnd is 83,000 segments, then our decrease factor is 0.1 (10%), and our increment is 72 segments per RTT (0.1%)
- More aggressive than standard TCP, but not as aggressive as Scalable TCP
- Experimental implementations in Linux (ORNL) and ns

w	a(w)	b(w)
38	1	0.50
118	2	0.44
221	3	0.41
347	4	0.38
495	5	0.37
663	6	0.35
851	7	0.34
1058	8	0.33
1284	9	0.32
1529	10	0.31
1793	11	0.30
2076	12	0.29
2378	13	0.28

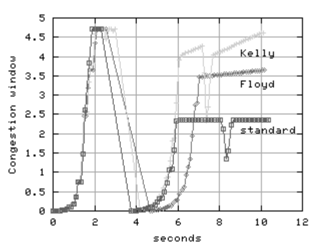
HS TCP in ns

- In ns $tcp set windowOption_ 8
- Example, RTT 140 ms, window of 3500 segments
  - From table W=3500 ➔ decrease by 0.26, add 16 segments per RTT



HS TCP and scalable TCP in the wild

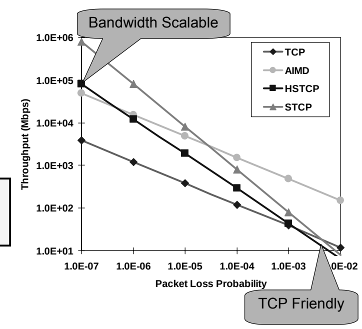
- Modified Linux kernel with HS TCP and scalable TCP
- 3 tests between ORNL and CERN with a UDP blast at about 3 seconds
- cwnd (Mbytes) information collected by instrumented kernel (Web100)



Response Functions of HSTCP and STCP

- HSTCP: $R = \frac{MSS}{RTT}\frac{0.12}{p^{0.835}}$
- STCP: $R = \frac{MSS}{RTT}\frac{0.08}{p}$

HSTCP and STCP are both bandwidth scalable and TCP friendly

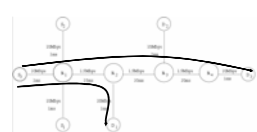


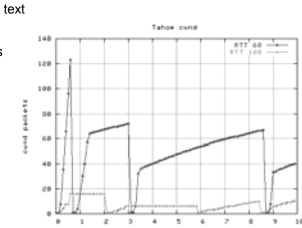
BI-TCP (NCSU)

- AIMD mods to recover fast at larger windows (scalable), but be TCP friendly at small windows
- Adjust for RTT unfairness
- Combines additive increase with binary search increase
  - When window is large, additive increase with large increment provides
    - Scalability
    - Linear RTT fairness
  - With small congestion window, binary search increase provides TCP-friendly response
- Currently in Linux 2.6 kernel, and there are mods for ns
- Many of following slides are from NCSU powerpoint (Rhee & Xu)

TCP recovery and RTT fairness

- Loss recovery is sensitive to RTT
  - Slow-start doubles cwnd (data rate) every RTT
  - Linear recovery increments cwnd by one segment (MSS) every RTT
- Nearby host will recover faster than distant host (droptail queue)
  - Example chap. 11 text
  - Congestion
  - Red flow 1349 kbs
  - Green 60 kbs





RTT Fairness on High-Speed Networks

- For a protocol with the following response function, where *c* and *d* are protocol-related constants.

$$R = \frac{MSS}{RTT}\frac{c}{p^d}$$

- The RTT Fairness Index (or the throughput ratio of two flows) on high-speed networks is

$$\left(\frac{RTT_2}{RTT_1}\right)^{\frac{1}{1-d}}$$

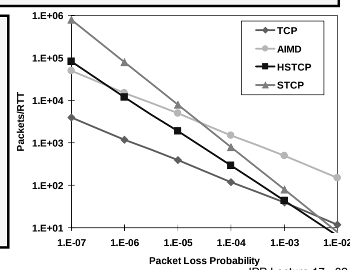
- On high speed networks, the RTT fairness of a protocol depends on the exponent *d* in the response function

Lisong Xu, Khaled Harfoush, and Injong Rhee, "Binary Increase Congestion Control for Fast Long-Distance Networks", in Proceedings of IEEE INFOCOM 2004, March, 2004, HongKong

Slide Determines the RTT Fairness

- If the response function is: $R(p) = \frac{MSS}{RTT}\frac{c}{p^d}$ then the RTT Fairness is: $\left(\frac{RTT_2}{RTT_1}\right)^{\frac{1}{1-d}}$

- The figure is shown in log-log scale, so exponent *d* in the response function is the slope of the line in the figure
- The slope of the line determines the RTT fairness of a protocol on high-speed networks



Simulation Results of RTT Fairness

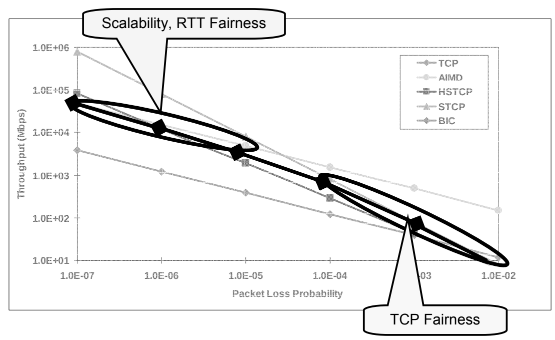
Throughout ratio of two flows on a 2.5Gbps link

Inverse RTT Ratio of two flows	1	3	6
AIMD	1	6	22
HSTCP	1	29	107
STCP	1	127	389

Example: for STCP if RTT ratio is 6, short RTT flow 389 times faster!

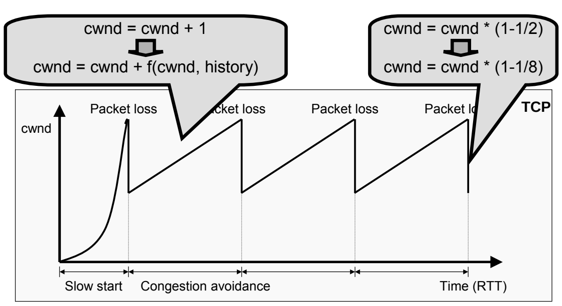
Simulation setup: BDP Buffer, Drop Tail, Reverse Traffic, Forward Background Traffic (short-lived TCP, Web Traffic)

BI-TCP Design Goal



BIC (Binary Increase Congestion control)

- BIC adaptively increase cwnd, and decrease cwnd by 1/8



A Search Problem

- A Search Problem
  - consider the increase part of congestion avoidance as a search problem, in which a connection looks for the available bandwidth by comparing its current throughput with the available bandwidth, and adjusting cwnd accordingly.

- Q: How to compare R with A?
  - R = current throughput
  - = cwnd/RTT
  - A = available bandwidth
- A: Check for packet losses
  - No packet loss: R <= A
  - Packet losses : R > A

- How does TCP find the available bandwidth?
  - Linear search

```
while (no packet loss){
   cwnd++;
}
```

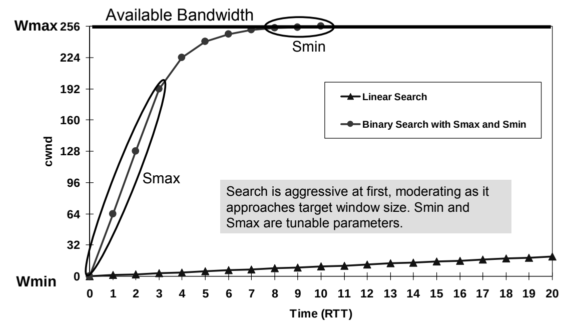
BI-TCP: Binary Search with Smax and Smin

- Binary search

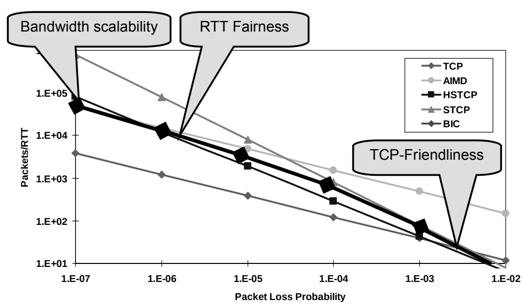
```
while (Wmin <= Wmax){
    inc = (Wmin+Wmax)/2 - cwnd;
    if (inc > Smax)
        inc = Smax;
    else if (inc < Smin)
        inc = Smin;
    cwnd = cwnd + inc;
    if (no packet losses)
        Wmin = cwnd;
    else
        break;
}
```

- Wmax: Max Window
- Wmin: Min Window
- Smax: Max Increment
  - Cwnd > Smax do linear with Smax/cwnd
  - Binary search would be too fast
  - 32
- Smin: Min Increment
  - Turn off bin search
  - 0.01
- low_window: 14
  - Cwnd smaller than this use standard TCP AIMD

Binary Search with Smax and Smin



Response Functions



BI-TCP peformance comparison

- RTT fairness, 2 flows 2.5 Gps link

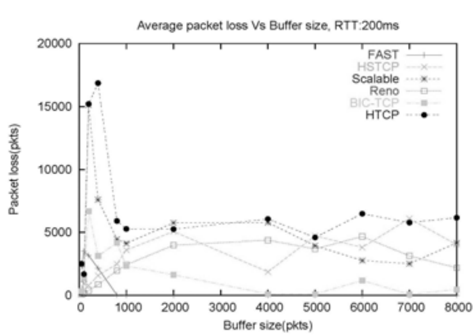
	Throughput ratio		
Inverse RTT ratio:	1	3	6
BIC	1	12	38
AIMD	1	6	22
HSTCP	1	29	107
STCP	1	127	389

- Feature comparison

	BI-TCP	AIMD	HSTCP	STCP
Scalable	Y	Y	Y	Y
TCP friendly	Y	no	Y	Y
Fair	Y	Y	no	no

- Good packet loss performance

Packet loss vs. Buffer Size

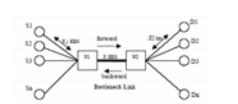


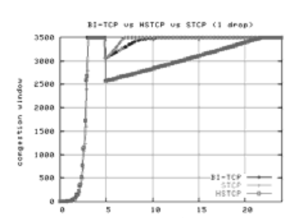
BI-TCP in ns

- NCSU has mods to tcp.cc and tcp.h for BI-TCP for ns
- In your tcl

  $tcp set windowOption_ 12

  BI-TCP ns simulation topology
  - Randomized start times
  - 20-50% background web traffic
  - 25 small TCP flows
  - 20% traffic on reverse path
  - 50 minutes real time for 3 minutes
- '05 NCSU proposed CUBIC






```
#ns default parameters for BI-TCP
Agent/TCP set bs_beta_ 0.875
Agent/TCP set bs_max_increment_ 32
Agent/TCP set bs_min_increment_ 0.01
Agent/TCP set bs_log_factor_ 4
Agent/TCP set bs_fast_convergence_ 1
Agent/TCP set low_window_ 14
```

TCP for LFN's

- BI-TCP is in Linux 2.6 kernel
  - S.ysctl's
```
net.ipv4.tcp_bic_low_window = 14
net.ipv4.tcp_bic_fast_convergence = 1
net.ipv4.tcp_bic = 1
```
- HS TCP, STCP, and BI-TCP try to improve TCP performance by recovering faster from packet loss
- Recall (lecture 14) SLAC's results for iperf tests across the Internet
  - TCP Reno single stream has low performance and is unstable on long distances
  - FAST TCP is very handicapped by reverse traffic
  - STCP is very aggressive on long distances
  - Bi-TCP performs very well in almost all cases
- We will look at FAST and Vegas shortly
  - Delay-based congestion avoidance
    - Try to improve TCP performance by avoiding loss in the first place!
- We will also eventually look at parallel TCP streams to speed recovery

Next time …

- Improving TCP performance by (automatically) selecting "best" buffer/window size
  - Bandwidth estimation
  - Auto-tuning

assignments 7 and 8