Internet Programming & Protocols Lecture 19

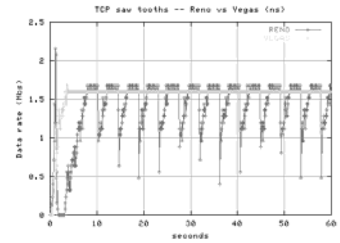
delay-based congestion avoidance

TCP Vegas

TCP FAST

TCP Westwood

Check this out …



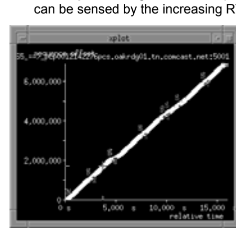
- Two runs: one with Reno, one with Vegas, 1.6 Mbs path, 100 ms RTT
  - TCP's window size exceeds queue size
  - Usual saw-tooth, packet loss, and ¾ datarate for Reno
  - BUT TCP Vegas has **no packet loss** and **runs at link speed**!
    - Is this the answer to our prayers?

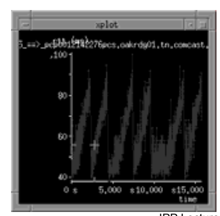
Accelerating TCP

- Tuning configuration parameters
  - SNDBUF/RCVBUF – bandwidth-delay product
  - Txquelen
  - RFC1323 (window scaling, timestamps)
  - Nagle, delayed ACK
  - Initial slow-start
- Speeding recovery after packet loss
  - Fast retransmit, fast recovery
  - SACK/FACK
  - AIMD, STCP, HSTCP, BI-TCP
- Avoiding packet loss
  - Dup threshold (out of order resiliance)
  - Slow-start and congestion avoidance (reduces losses)
  - Vegas/FAST

Delay-based congestion avoidance

- Standard TCP detects congestion by packet loss
  - Then we must go thru all sorts of gyrations to speed recovery
    - Fast retransmit, fast recovery, SACK, FACK, HSTCP, BI-TCP
- TCP Vegas tries to avoid packet loss by slowing down (reducing cwnd) when RTT starts to increase
  - **Assumption**: congestive loss is preceded by buildup in router queue which can be sensed by the increasing RTT



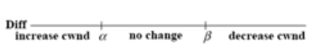


Vegas

- Sender adjusts sending rate to avoid filling the buffer
- Let BaseRTT be the minimum of all measured RTTs
- Sender-side bandwidth estimation: Compute
  ExpectRate = CongestionWindow/BaseRTT
- Sender calculates sending rate (ActualRate) once per RTT
- Sender compares ActualRate with ExpectRate

```
Diff = ExpectedRate - ActualRate
if Diff < α
       increase CongestionWindow linearly
else if Diff > β
       decrease CongestionWindow linearly
else
       leave CongestionWindow unchanged
```

If RTT grows, ActualRate will shrink, and Diff will grow, and cwnd will be reduced.



Typically: **α = 1 β = 3**

Vegas paper ('95)

- To avoid drops, slow-start moderator (γ)
  - Exponential growth every other RTT, to be able to detect/avoid congestion
  - Do normal slow-start until **expected - actual >** γ then do linear increase
  - Slow-start begins with cwnd ← 2 (not 1)
  - Some good arguments for moderating slow-start for large windows, but probably hurts performance for small windows
- Proposes new retransmission mechanism
  - RTT for every segment sent is measured with high res timer
  - Dup ACK is checked against segment timeout, if exceeded, retransmit
- Paper also proposed multiplicative decrease of ¼
  - Certainly helped performance in the event of loss
  - But has nothing to do with delay-based congestion control
  - Biased some of the performance results in the paper

Vegas implementations

- Original work done on X kernel emulator
- Linux 2.6 has Vegas (off by default)
  - sysctl's
    - net.ipv4.tcp_vegas_gamma = 2
    - net.ipv4.tcp_vegas_beta = 6
    - net.ipv4.tcp_vegas_alpha = 2
    - net.ipv4.tcp_vegas_cong_avoid = 0
- ns
  - Agent/TCP/Vegas
  - Defaults
    - Agent/TCP/Vegas set v_alpha_ 1
    - Agent/TCP/Vegas set v_beta_ 3
    - Agent/TCP/Vegas set v_gamma_ 1

Vegas performance

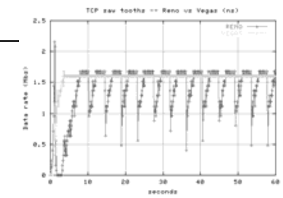
- Potentially great
- Chapter 11 pair-wise tests
  - Vegas has no losses, but

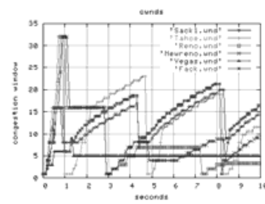
Flavor	goodput Kbs
Vegas/BI-TCP	105/1370
Vegas/Fack	128/1346
Vegas/Sack1	138/1336
Vegas/Tahoe	178/1256
Vegas/Newreno	311/1141
Vegas/Reno	368/1068
Vegas/Westwood	506/963
Vegas/Vegas	919/553

- Chapter 11, 11.7, all competing

Flavor	goodput Kbs
Fack	397
Tahoe	346
Sack1	344
Newreno	164
Vegas	140
Reno	49

Anecdotal results





Vegas performance

- Good
  - Often higher throughput, fewer losses
  - Keeps queue size small (max of α packets in q)
  - Vegas requires a high-precision timer (tick) and measures RTT on every packet (timestamps would help)
- Bad
  - Too friendly, politely backs off as other TCP variants consume the bandwidth
  - If initial BaseRTT is too high, then performance limited
  - Competing Vegas flows: 2nd flow to start observes longer RTT and doesn't get as much bandwidth
  - Congestion on reverse path can increase RTT and cause Vegas to use less bandwidth on forward path
- Open research: proper values for α, β (or dynamically adjust?!)
- LANL proposals to increase α, β for long fat pipes …
- Recent delay-based congestion avoidance interest focused on FAST

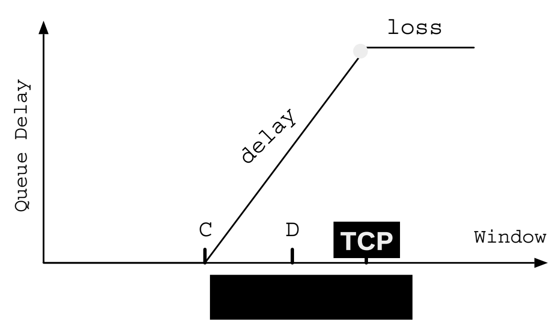
FAST

- CalTech's new ('02) TCP control algorithm for high speed nets
- Delay-based congestion avoidance
  - RTT estimators rather than Vegas bandwidth estimator
  - Sensing queuing delays (need high precision time stamps)
- Patches for Linux
- Licensing restrictions ☹

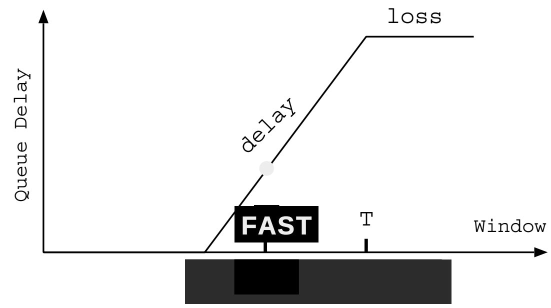
Difficulties at large window

- Equilibrium problem
  - Packet level: AI too slow, MD too drastic.
  - Flow level: requires very small loss probability.
- Dynamic problem
  - Packet level: must oscillate on a binary signal.
  - Flow level: unstable at large window.

Problem: binary signal



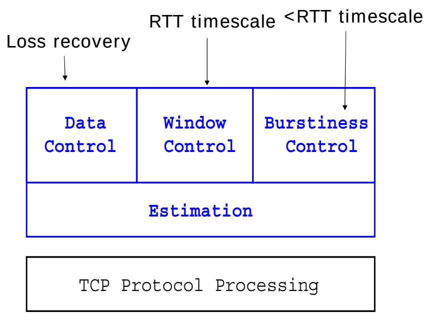
Solution: multibit signal (variation in RTT)



Queueing Delay in FAST

- Queueing delay is not used to avoid loss
- Queueing delay defines a target number of packets (α) to be buffered for each flow
- Queueing delay allows FAST to estimate the distance from the target

Architecture

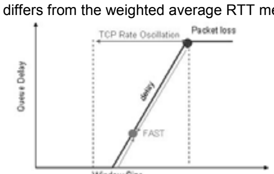


Data control – which packets to send
Window control – how many packets to send
Burst control – when to send packets

Window Control Algorithm

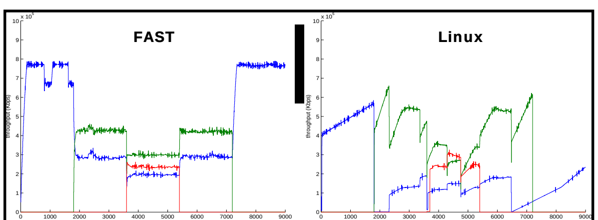
$$w \leftarrow w \cdot \frac{\text{baseRTT}}{\text{RTT}} + \alpha$$

- RTT: exponential moving average with weight of min {1/8, 3/cwnd}
- baseRTT: latency, or minimum RTT
- α determines fairness and convergence rate
- Fast maintains an exponential weighted average RTT measurement and adjusts its window in proportion to the amount by which the current RTT measurement differs from the weighted average RTT measurement.



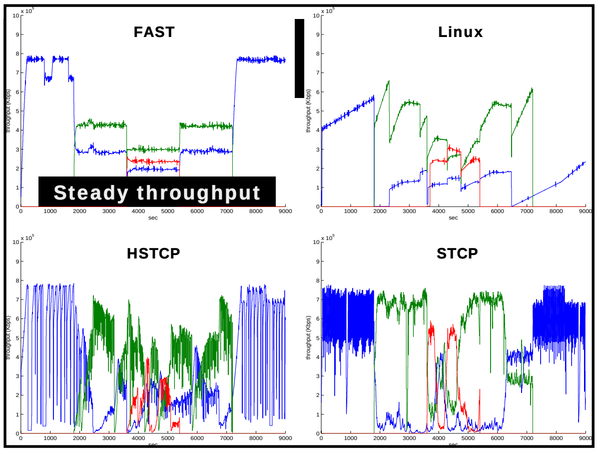
FAST implementations

- Patches available for linux and ns
- Parameters
  - tcp_fast on/off
  - tcp_fast_alpha
  - tcp_fast_beta typically 17/16 of alpha
  - tcp_fast_gamma slow start parameter
  - tcp_fast_bc burst control

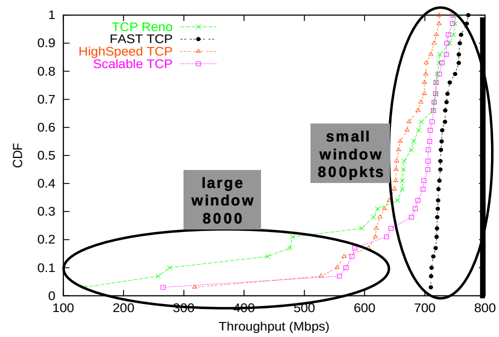


Dynamic sharing on Dummynet

- capacity = 800Mbps
- delay=120ms
- 3 flows
- iperf throughput
- Linux 2.4.x (HSTCP: UCL)

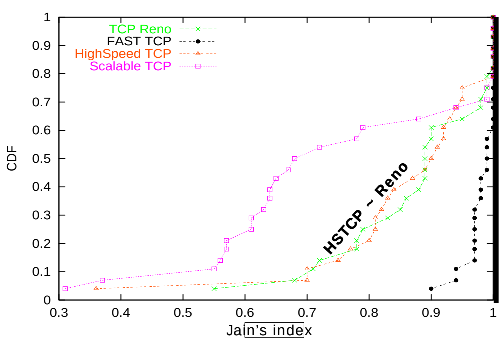


Aggregate throughput



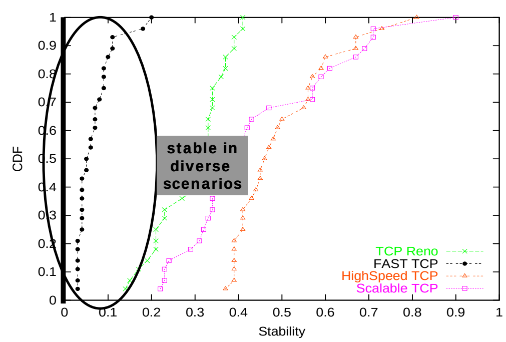
Dummynet: cap = 800Mbps; delay = 50-200ms; #flows = 1-14; 29 expts

Fairness



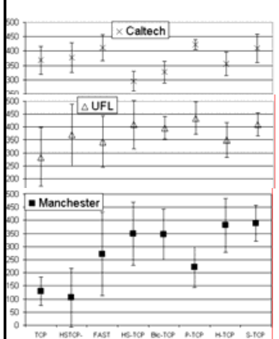
Dummynet: cap = 800Mbps; delay = 50-200ms; #flows = 1-14; 29 expts

Stability



Dummynet: cap = 800Mbps; delay = 50-200ms; #flows = 1-14; 29 expts

Throughput – SLAC wide-area tests



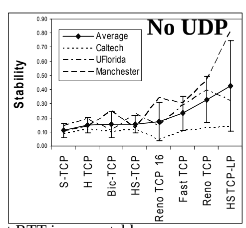
Avg throughput for optimal & large window sizes from SLAC to CalTech, UFl & Manchester

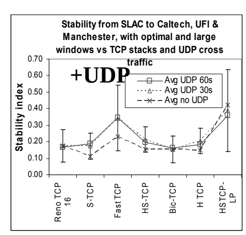
Stack more important for long RTTs

Single stream Reno & HSTCP-LP poorer on large RTTs

Stability (throughput std. dev/average)

Stability from SLAC to Caltech, U Florida & Manchester





Short RTT is more stable

Little difference between periodicity of UDP (30 & 60 secs)
HSTCP-LP & FAST have larger stability indices (less stability)

Open issues

- baseRTT estimation
  - route changes, dynamic sharing
  - does not upset stability
- small network buffer
  - at least like TCP
  - adapt α on slow timescale, but how?
- TCP-friendliness
  - friendly at least at small window
  - tunable, but how to tune?
- reverse path congestion
  - should react? rare for large transfer?
  - SLAC tests show FAST TCP is very handicapped by reverse traffic
- “DCA for TCP” shows with Internet measurements and ns that DCA can predict/avoid only 7% to 18% of congestions events.
- may need fast recovery mechanisms for non-congestive loss

Delay-based congestion avoidance is hard and doesn’t compete well with loss-based algorithms.

IP Rights for FAST ☹

- Caltech owns IP rights
  - applicable more broadly than TCP
  - leave all options open
- Will license free at least for education & research community
- Will be flexible to facilitate wide deployment

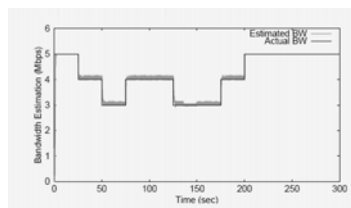
TCP Westwood

- Enhance congestion control by sender-side bandwidth estimation
  - Estimates computed by sampling and exponential filtering
  - Samples are determined from ACK inter-arrival times and info in ACKs regarding bytes delivered (like packet-pair estimators)
  - Westwood calls the estimate “Fair Share Estimate” (FSE)
- FSE is used to set cwnd and ssthresh after packet loss
  - For 3 dup ACKs
    - ssthresh ← FSE * RTTmin (instead of cwnd/2)
    - if cwnd > ssthresh then cwnd ← ssthresh
  - For timeout
    - ssthresh ← FSE * RTTmin and cwnd ← 1
    - RTTmin is min RTT observed for flow

FSE * RTTmin == bandwidth-delay product = the most recent observed data rate of the connection

Bandwidth estimation accuracy

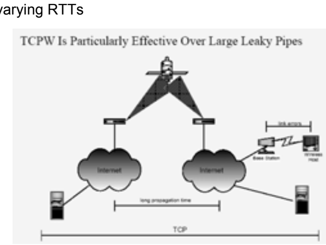
- TCP sharing flow with ON/OFF UDP flow



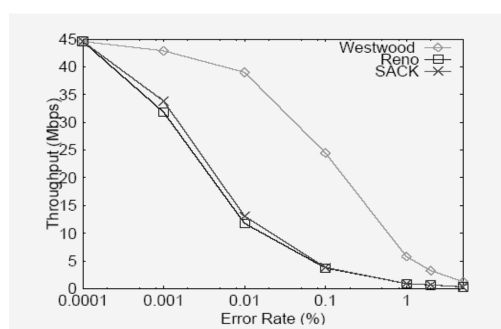
BSE Accuracy (With Concurrent On/Off UDP Traffic)

TCPW benefits

- Efficiency
  - Better link utilization when loss are due to non-congestive events (random loss, lossy medium (wireless)) as well as congestion
  - Significant gain for large pipe with big RTT
- Better fairness over varying RTTs
- Friendliness good
- Stability good



TCPW and random loss

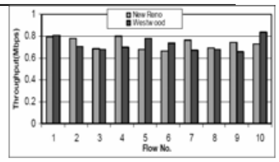


TCPW and fairness

- Fairness with competing connections

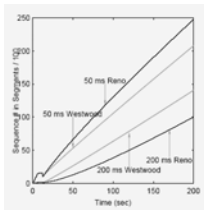
  Jain's

$$J = \frac{\left(\sum_{i=1}^{n} \gamma_i\right)^2}{\left(\sum_{i=1}^{n} \gamma_i^2\right) n}$$



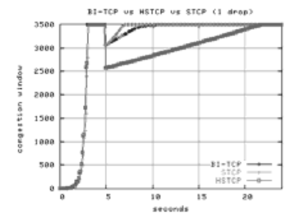
  As fair as NewReno

- RTT fairness



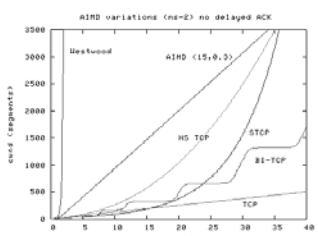
Single drop example (LFN)

- TCPW actually does not even flinch, cwnd stays at 3500
- Treats single-loss as non-congestion event since no RTT changes prior to loss – why TCPW is good for wireless



Early drop in slow-start

- LFN nightmare: early packet loss in initial slow-start
- Early drop in slow-start, so congestion-avoidance phase dominates
- RTT 140 ms, target window 3500 segments, no delayed ACKs



TCPW implementations

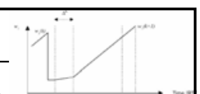
- Linux 2.6
  - sysctl net.ipv4.tcp_westwood = 0
- ns I've added Agent/TCP/WestwoodNR
  - For Vegas/Fast/Westwood in ns, you want tcpTick_ to be 0.01 (ns default now)
  - TCPW is fairest →
- TCPW+ combines bandwidth estimation with rate estimation

Table 11.2

flavor/flavor	Goodput Kbs
Reno/Reno	924/445
Tahoe/Reno	1233/196
Vegas/Reno	368/1068
Newreno/Reno	964/448
Sack1/Reno	982/466
Fack/Reno	1017/444
BITCP/Reno	1370/105
Westwood/Reno	642/649

- Estimator summary
  - **Vegas uses bandwidth estimate over a RTT**
  - **FAST uses RTT estimator**
  - **TCPW uses bandwidth estimate base on packet-pairs "averaged" over recent ACKs**

H-TCP

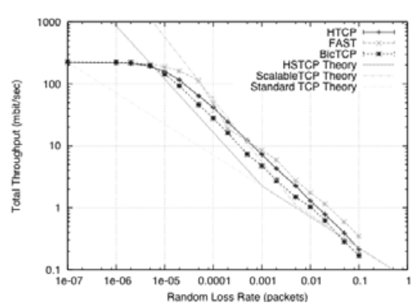


- timer-based response function to window inflation
- Multiplicative decrease b = RTTmin/RTTmax
  - RTTmax and RTTmin for the previous congestion interval
  - If RTT has variance > 20%, then b = ½ (standard TCP)
- Additive increase is 1 (a =1, standard TCP) for an initial period (1 s)
  - Later increase is a 2nd degree polynomial function of the time since the last congestion event
    - cwnd ← cwnd + f(T)/cwnd
    - a = ½ $(T-1)^2$ + 10 (T-1) + 1
    - Where T is the time since the last congestion event
    - Further modified by b a' = 2a(1-b)

T	cwnd
1.1	100
3.1	1000
4.3	2000
6.6	5000
9.2	10000

- SLAC tests show H-TCP worse than competitors (see better results at H-TCP site)
- Patches for Linux and mods for ns available

H-TCP response function



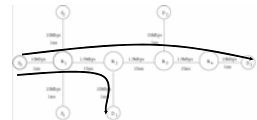
TCP … what to use?

- Tahoe
- Reno
- NewReno
- SACK
- FACK
- STCP
- HSTCP
- BI-TCP
- TCPW
- H-TCP
- Vegas
- FAST

- Differentiators
  - Slow-start
  - AIMD values
  - ACK/SACK info
  - Loss based vs delay based
  - Fair
  - Stable
  - TCP-friendly
  - RTT fairness
  - Scalable
  - Available?
- Typical: NewReno + SACK
  - Linux BI-TCP
- Don’t forget proper window size

Next time …

- Active queue management
- XCP



assignment 9