On Estimating End-to-End Network Path Properties

Mark Allman, NASA GRC/BBN

Vern Paxson, ACIRI/LBNL

ACM SIGCOMM

September, 1999

Overview

- General transport problem: adapting to current conditions. E.g., congestion control.

  E.g., estimating how long to wait before retransmitting. E.g., estimating bandwidth available when connection begins.

- Methodology:

  - trace-driven simulation of $\mathcal{N}_2$ dataset [Pax97]
  - 18,490 connections, 100 KB each, 31 hosts
  - compare adaptive algorithms' decisions ...
  - ... with what actually happened.

How Long to Wait Before Retransmitting

- Tension: wait long enough for any tardy ACK to arrive ...
  ... but no longer.
- RTO estimation algorithm [Jac88] uses:

$$\text{RTO} = SRTT + k \cdot RTTVAR, \quad k = 4$$

- For every RTT measurement, $SRTT$ and $RTTVAR$ updated via EWMA:

$$SRTT \leftarrow (1 - \alpha_1) SRTT + \alpha_1 RTT_{\text{meas}}, \quad \alpha_1 = \frac{1}{8}.$$

- $RTTVAR$ updated based on deviation $|SRTT - \text{RTT}_{\text{meas}}|$ using $\alpha_2 = \frac{1}{4}$.

Additional Issues

- Clock measurements are done using a granularity G.
  BSD default: $G = 500$ msec.
- Clock measurement done using "heartbeat" timer.
- RTO bounded by $\mathrm{RTO}_{\mathrm{min}} = 2G = 1$ sec.
- Q: What about measuring more often than once per RTT?
  (Affects EWMA constants.)
- Q: What about using fine-grained clocks?

Assessing Different RTO Estimation Algorithms

- For each unavoidable data packet retransmission, charge the estimator with the current RTO.
- For each ACK of new data, if arrived after RTO, charge the estimator with a bad timeout. If ACK is for a segment being timed, update SRTT and RTTVAR.
- For each ACK of new data, restart the RTO timer.
- Let:

  W = total time spent waiting for necessary timeouts.

  $\widetilde{W}$ = per-connection time waiting, normalized in RTTs.

  B = mean proportion, per connection, of bad timeouts.

Varying the Minimum RTO

Minimum RTO	W	$\widetilde{W}$	B
1,000 msec	144,564	8.4	0.63%
750 msec	121,566	6.5	0.76%
500 msec	102,264	4.8	1.02%
250 msec	92,866	3.5	2.27%
0 msec	92,077	3.1	4.71%
RTO = 2,000 msec	229,564	15.6	2.66%
RTO = 1,000 msec	136,514	8.2	6.14%
RTO = 500 msec	85,878	4.5	12.17%

$G = 1$ msec.

Varying the Clock Granularity

Granularity	W	$\widetilde{W}$	B
500 msec	272,885	19.2	0.36%
[WS95] (500 msec)	245,668	15.4	0.23%
250 msec	167,360	10.2	0.67%
100 msec	142,940	8.4	0.95%
50 msec	143,156	8.4	0.84%
20 msec	143,832	8.4	0.70%
10 msec	144,175	8.4	0.67%
1 msec	144,564	8.4	0.63%

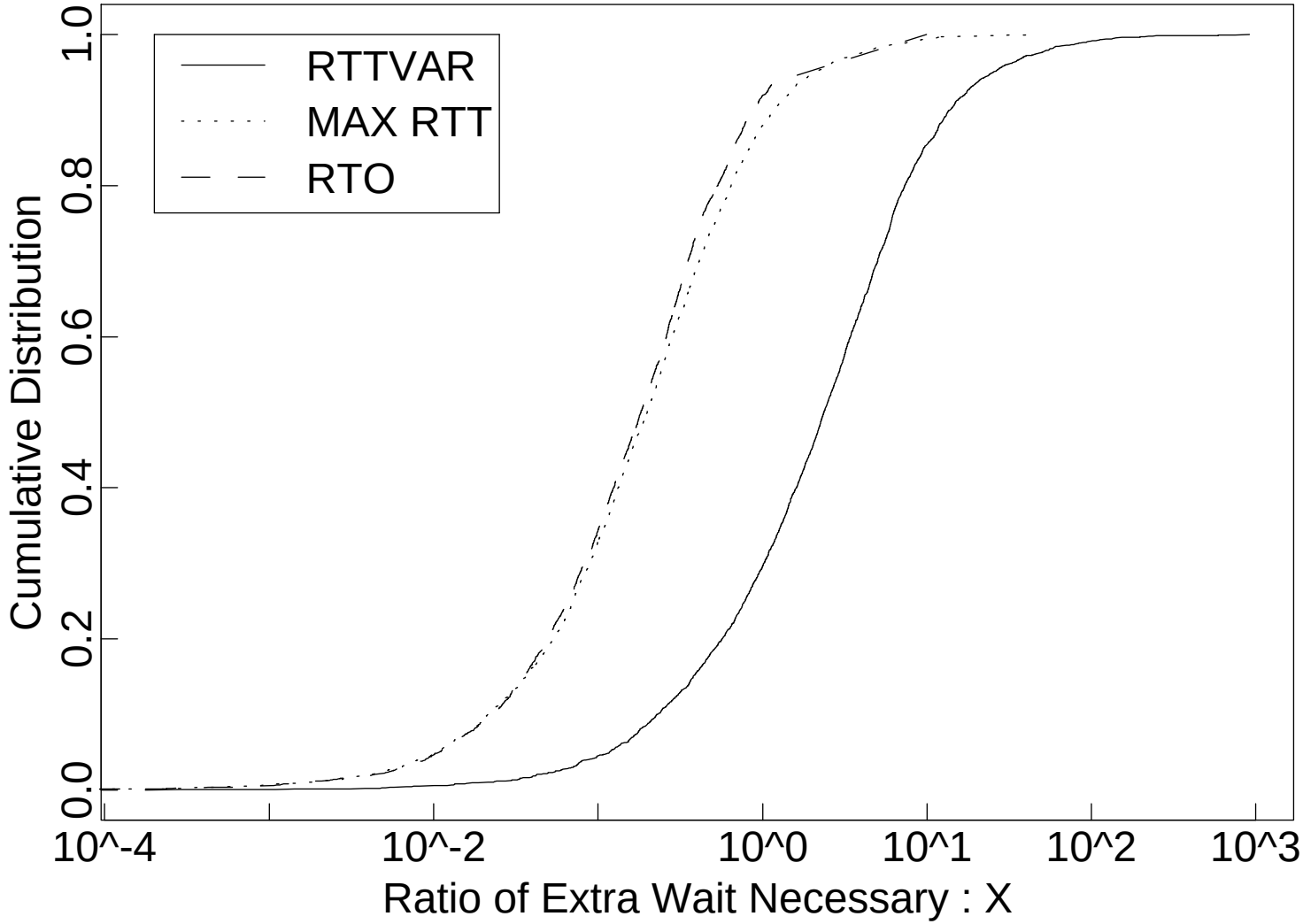
$\mathrm{RTO}_{\mathrm{min}} = 1$ sec.

Varying EWMA Parameters

$G = 1$ msec, $\mathrm{RTO}_{\mathrm{min}} = 0$ sec

Parameters	W	$\widetilde{W}$	B
[WS95]	245,668	15.4	0.23%
[WS95]-*every*	241,100	14.7	0.25%
take-first $(\alpha_1, \alpha_2 = 0, \mathrm{RTO}_{\mathrm{min}} = 1\ s)$	158,199	8.5	0.74%
take-first $(\alpha_1, \alpha_2 = 0)$	131,180	4.4	2.93%
very-slow $(\alpha_1 = \frac{1}{80}, \alpha_2 = \frac{1}{40})$	113,903	3.9	3.97%
slow-every $(\alpha_1 = \frac{1}{32}, \alpha_2 = \frac{1}{16})$	102,544	3.4	4.28%
slow $(\alpha_1 = \frac{1}{16}, \alpha_2 = \frac{1}{8})$	96,740	3.4	3.84%
std $(\alpha_1 = \frac{1}{8}, \alpha_2 = \frac{1}{4})$	92,077	3.1	4.71%
std-every $(\alpha_1 = \frac{1}{8}, \alpha_2 = \frac{1}{4})$	94,081	3.1	5.09%
fast $(\alpha_1 = \frac{1}{2}, \alpha_2 = \frac{1}{4})$	90,212	3.0	7.27%
take-last $(\alpha_1, \alpha_2 = 1)$	93,490	3.3	19.57%
take-last-every $(\alpha_1, \alpha_2 = 1)$	97,098	3.5	20.20%
take-last $(\alpha_1, \alpha_2 = 1, \mathrm{RTO}_{\mathrm{min}} = 1\ s)$	145,571	8.5	1.30%

Extra Waiting Time Necessary to Avoid Bad RTO



Final RTO Thoughts

- What matters: min. RTO; timer gran. $\leq$ 100 msec.
- What doesn't: EWMA parameters; how often you time.
- Is RFC 1323's timestamp option worth the hassle?
- Is retransmitting unnecessarily really all that bad?
  - There's enough capacity ... just need to undo *cwnd/ssthresh* changes.
    - Detecting: using SACK, or timestamps.
    - Or: see whether ACK arrives within $\frac{3}{4}$ RTT_{min}.
- A possible different approach: estimate the feedback time.

Estimating Bandwidth: Goals

- Estimate the appropriate sending rate without pushing the network path as hard as the current mechanism does.

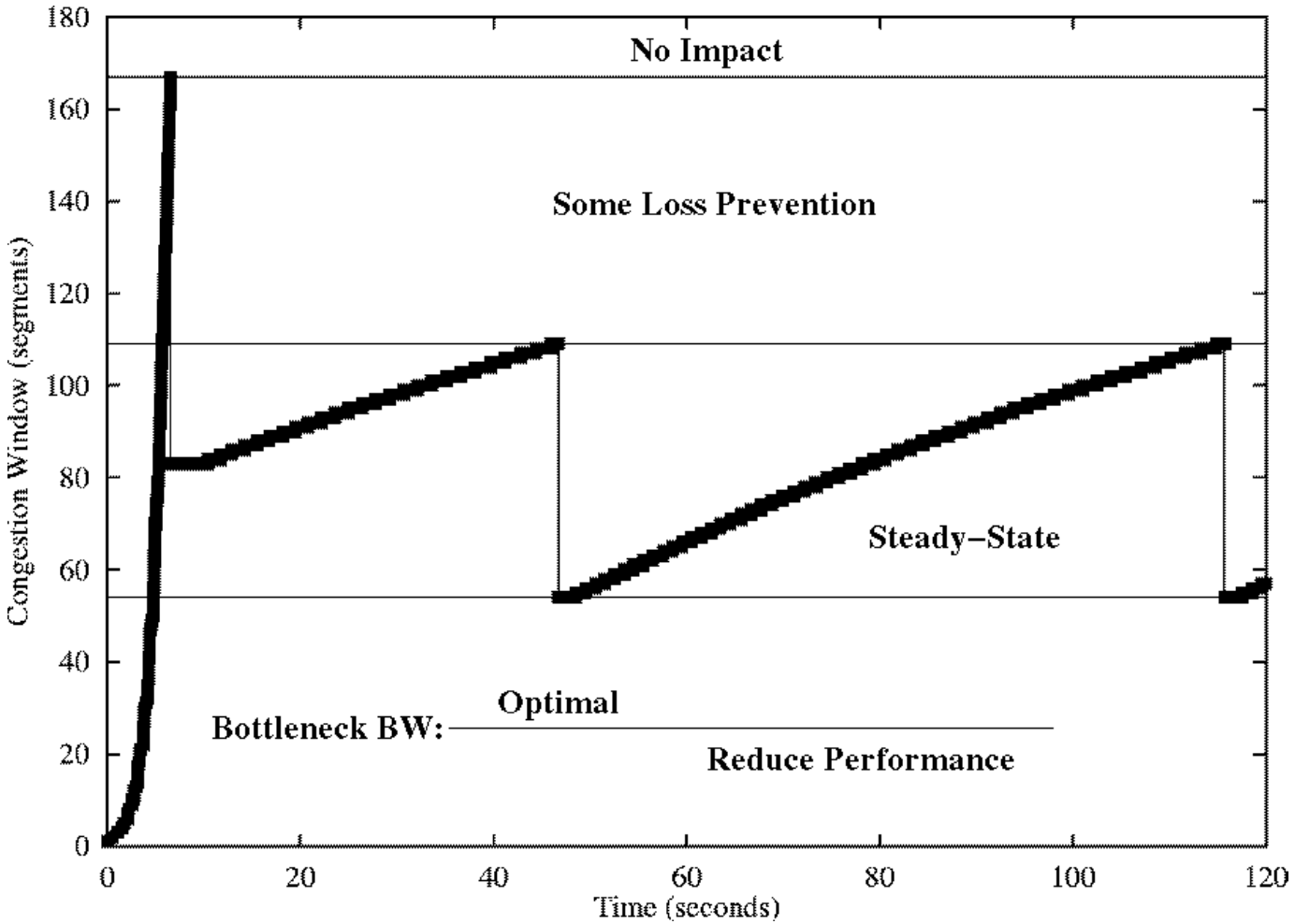
- Determine to what degree the timing structure of flights of packets can be used to estimate the available bandwidth.

Methodology

- Ideally, we'd like to estimate *available* bandwidth, which we combine with the measured RTT to estimate *ssthresh*.
- However, much of our analysis is in terms of *bottleneck* bandwidth, as an upper bound on a good *ssthresh* estimate.
- PBM estimate of bottleneck bandwidth [Pax97] used for "correct" *ssthresh*
  - PBM is not appropriate for use "on the fly".
  - We used PBM' (only considers transfer up to loss point) as "upper bound" on how well we can hope to estimate.

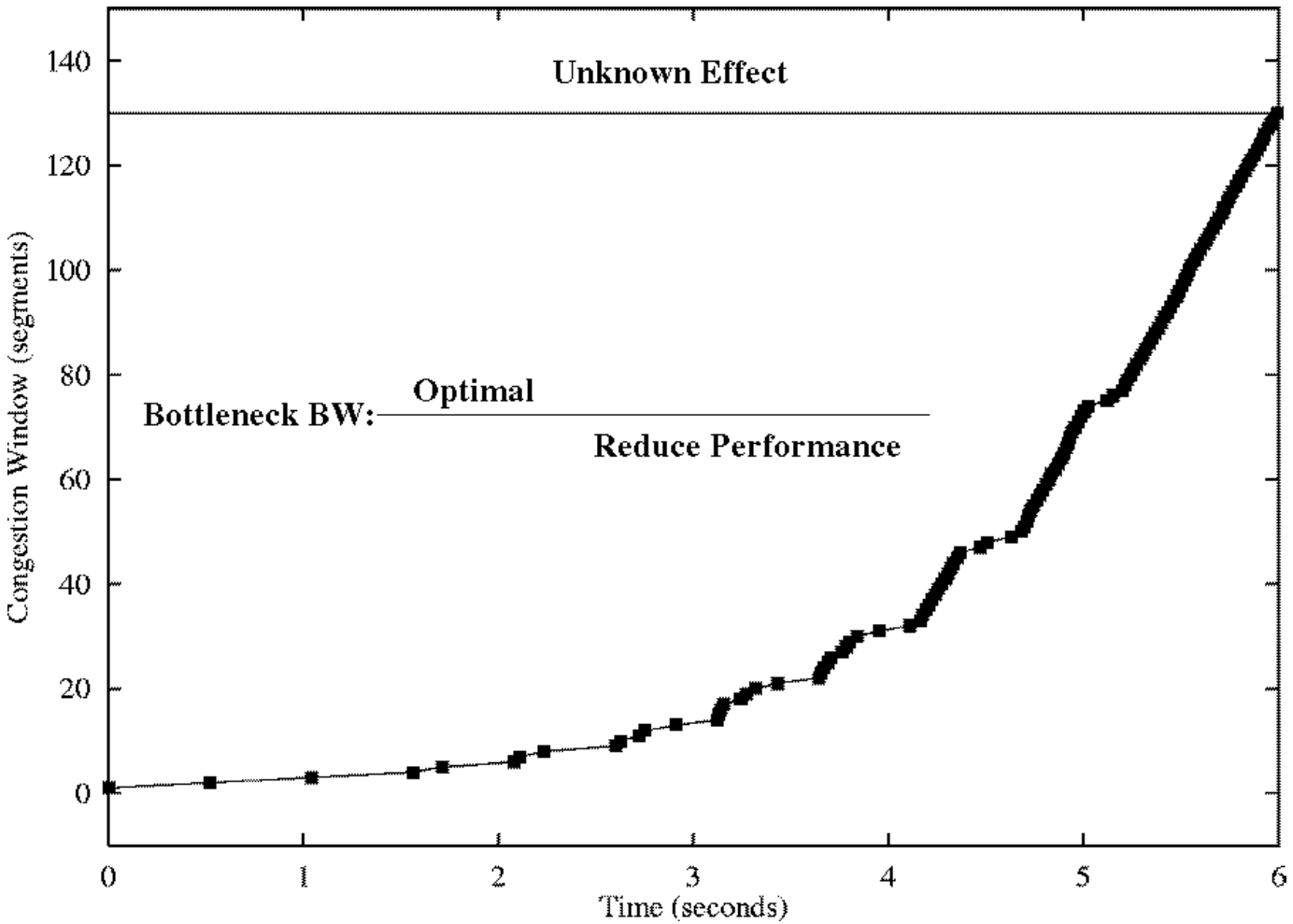
Methodology (cont.)

- Connections with loss:



Methodology (cont.)

- Connections without loss:



Tracking Slow Start Flights (TSSF)

- Sender-side, TCP specific algorithm.
- Watch for 3 ACKs within a flight and estimate from those.
- Results:

Case	No Est.	No Imp.	Prv. Loss	Stdy. State	Opt.	Tot.	Red. Perf.
With Loss	42%	1%	1%	3%	0%	4%	52%

Case	No Est.	Unk. Imp.	Opt.	Red. Perf.
Without Loss	13%	2%	2%	82%

- Major problem is underestimate of bandwidth.
  - Caused largely by delayed ACKs.

Closely-Spaced ACKs (CSA)

- Sender-side, TCP independent algorithm [Hoe96].
- Watch for n ACKs within $\nu \cdot RTT$ seconds.

Case	No Est.	No Imp.	Prv. Loss	Stdy. State	Opt.	Tot.	Red. Perf.
Loss, $CSA_{n=3}^{\nu=0.1}$	62%	20%	6%	9%	2%	17%	2%
Loss, $CSA_{n=2}^{\nu=0.05}$	53%	37%	5%	4%	0%	9%	1%
Loss, $CSA_{n=2}^{\nu=0.1}$	45%	32%	8%	10%	2%	19%	4%
Loss, $CSA_{n=2}^{\nu=0.2}$	38%	24%	9%	13%	3%	25%	13%

Case	No Est.	Unk. Imp.	Opt.	Red. Perf.
No Loss, $CSA_{n=3}^{\nu=0.1}$	24%	42%	13%	22%
No Loss, $CSA_{n=2}^{\nu=0.05}$	19%	59%	11%	10%
No Loss, $CSA_{n=2}^{\nu=0.1}$	14%	48%	11%	27%
No Loss, $CSA_{n=2}^{\nu=0.2}$	13%	34%	11%	43%

- Problems: No impact/estimate in many transfers with loss and reduces performance in many no loss transfers.

Tracking Closely-Spaced ACKs (TCSA)

- Sender-side, TCP independent algorithm [AD98]
- TCSA: Wait until CSAs converge to minimum CSA sample.
- TCSA': Observe CSAs until subsequent samples converge.

Case	No Est.	No Imp.	Prv. Loss	Stdy. State	Opt.	Tot.	Red. Perf.
Loss, TCSA	62%	14%	6%	11%	1%	19%	5%
Loss, TCSA′	70%	10%	6%	9%	2%	17%	2%

Case	No Est.	Unk. Imp.	Opt.	Red. Perf.
No Loss, TCSA	24%	25%	8%	44%
No Loss, TCSA′	27%	33%	11%	28%

- Problems:
  - TCSA underestimates often – TCSA' helps
  - Neither algorithm helps that often

Tracking Burst of Data Segments

- Receiver-side, TCP specific algorithm
- Tracks which segments will be liberated by a given ACK.

Case	No Est.	No Imp.	Prv. Loss	Stdy. State	Opt.	Tot.	Red. Perf.
Loss, Recv_{min}	11%	32%	6%	13%	4%	23%	34%
Loss, Recv_{avg}	11%	52%	10%	14%	9%	34%	3%
Loss, Recv_{med}	11%	48%	10%	14%	10%	34%	7%
Loss, Recv_{max}	11%	65%	7%	8%	8%	23%	0%

Case	No Est.	Unk. Imp.	Opt.	Red. Perf.
No Loss, Recv_{min}	1%	15%	2%	83%
No Loss, Recv_{avg}	1%	46%	23%	31%
No Loss, Recv_{med}	1%	45%	28%	26%
No Loss, Recv_{max}	1%	71%	27%	1%

- Problems: Overestimates or underestimates very often
- Winner: Recv_{max} improves $\approx 25\%$, almost never impairs

Summary / Open Issues

- Receiver-side estimation a win because a lot less noisy.
- But only a win for 25% of connections — compelling?
- What about ramping up a connection's rate more quickly?
- Need to evaluate on more recent data and conduct live experiments.
- Effects of RED?