TCP-Friendly Simulations

M. Handley, S. Floyd

1 Introduction

This document presents simulation results for a class of TCP-friendly congestion control schemes intended for use in a reliable multicast context. The ideas behind the congestion control scheme are not novel - they distill ideas from several members of the IRTF Reliable Multicast research group.

The purpose of this work is to extensively evaluate these ideas as a start towards understanding where they perform well and where they perform poorly.

2 Protocol

We plan to evaluate the congestion control scheme first in a unicast context, and then in a multicast context which adds additional complications.

2.1 Unicast Protocol

A connection consists of a single rate-based sender and a single receiver. The sender sends data packets. Included in each data packet is the current send rate and the current measured RTT.

The receiver sends a single acknowledgement packet every RTT.

The receiver calculates a sample interval over which to measure loss rates. We use the number of congestion events over the number of packets sent as loss fraction, where several packet losses contribute to a single congestion event if they fall within the same RTT.

Given a current rate, R, the sample number of packet, P_{sample}, is:

$$P_{sample} = \frac{k_{sample}}{l_{exp}}$$

where l_{exp} is the expected packet drop fraction for this rate calculated from the TCP bandwidth equation. k_{sample} is a measure of how much we need to oversample compared to the minimum number of packets needed to be sampled to obtain this rate. Typically k_{sample} might be 4.

In the initial experiments we are using the older equation:

$$R = \frac{s}{t_{RTT}\sqrt{l_{exp}}} \qquad (1)$$

where s is the packet size. Hence:

$$P_{sample} = k\left(\frac{t_{RTT}R}{s}\right)^2$$

The loss fraction is not always measured over this interval however - at startup there may not be this many packets to sample over, and later this sample may include no lost packets - a situation which this equation does not hold. Thus if P_{tot} is the total number of packets received so far, and $P_{Llosses}$ is the minimum sample large enough to include L congestion events, then the actual number of packets, P_{actual}, used to calculate the loss fraction, is:

$$P_{actual} = Max(P_{Llosses}, Min(P_{sample}, P_{tot}))$$

There are two variations of this protocol, termed version 0 and version 1. The version number doesn't represent evolution - merely a different algorithm.

Version 0

The receiver calculates the loss fraction, l_{actual} from the connection history by counting the number of congestion events, c, in the last P_{actual} packets:

$$l_{actual} = \frac{c}{P_{actual}}$$

If $l_{actual} < k_{incr} l_{exp}$ the sending rate is overconformant, and this is signalled to the sender. k_{incr} is typically 0.5.

If $l_{actual} > k_{desc} l_{exp}$ the sending rate is nonconformant, and this is signalled to the sender. k_{decr} is typically 1.5.

If the sender receives an ACK indicating it is overconformant, it increases its rate by one packet per RTT. If it is nonconformant, it decreases its rate to half its current rate. These changes can happen at most once per RTT.

Version 1

The receiver calculates the loss fraction, l_{actual} from the connection history by counting the number of congestion events, c, in the last P_{actual} packets:

$$l_{actual} = \frac{c}{P_{actual}}$$

It reports this loss fraction to the send in an ACK. The sender then calculates the expected rate, R_exp from this loss fraction:

$$R_{exp} = \frac{s}{t_{RTT}\sqrt{l_{actual}}}$$

If $R_{exp} < R_{actual}$, where R_{actual} is the current sending rate, the current sending rate is reduced to R_{exp}.

If $R_{exp} > R_{actual}$, the current sending rate is increased by one packet per RTT.

2.2 Additional Restrictions

Version 0 was found to oscillate badly when the data rate was close to or below one packet per RTT. To attempt to cure this oscillation, an additional set of rules was added:

- Instead of changing the bandwidth once per RTT, we do not allow it to increase more often than once every k_{sample} RTTs. Motivation: If we are detecting overshooting by essentially waiting for at least K congestion events (after a long period of no congestion), then increases should be done at most once per K RTTs.
- If the data rate is less that four packets per RTT, increase the data rate by no more than a quarter of the current sending rate. Motivation: when the data rate is close to one packet per RTT, increases of one packet per RTT are far too large and tend to cause the data rate to overshoot.

We term these additional rules the "slowincr" rules. We performed most tests both with and without these restrictions so that their impact was clear.

3 Results

3.1 Competing with TCP

In this set of simulations we examine how TFRM performs when competing with TCP in a steady state scenario. We run n TCP connections and n TFRM connections through the same bottleneck for 30 seconds. We measure the throughput of each TCP and TFRM flow over the last 15 seconds to ignore any differences in start-up performance caused by TCP's exponential slow-start behaviour. The RTTs are similar but vary over a range of 44ms to 86ms. We try out but DropTail and RED buffering, and link bandwidths of 1.5Mb/s, 15Mb/s, and 60Mb/s and values of n ranging from 1 to 64 (64 TCP and 64 TFRM) flows.

For buffering, we use the following values:

With 1.5Mb/s bottleneck:

DropTail: 25 packet buffers.

RED: 25 packet buffers, min-thresh=3, max-thresh=13, max-p=0.1

With 15Mb/s bottleneck:

DropTail: 250 packet buffers.

RED: 250 packet buffers, min-thresh=25, max-thresh=125, max-p=0.1

With 60Mb/s bottleneck:

DropTail: 1000 packet buffers.

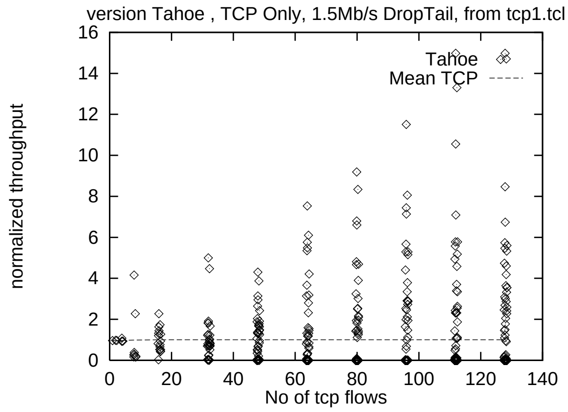
RED: 1000 packet buffers, min-thresh=100, max-thresh=500, max-p=0.1

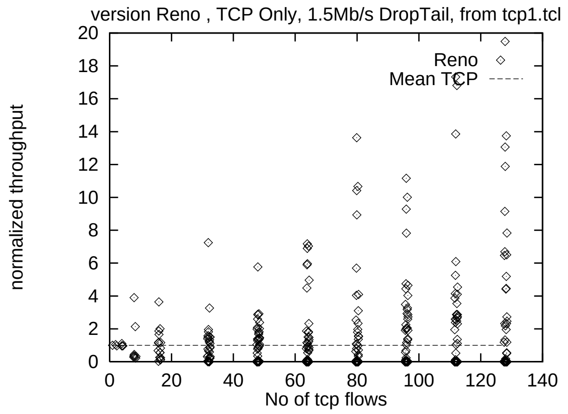
Each simulation run produces a set of mean bandwidths for the TCP flows and for the TFRM flows, plus a overall mean TCP bandwidth and a mean overall TFRM bandwidth. We show the indivdual rates as jittered points on the following graphs; each graph shows 11 simulation runs with differing values of n. The vertical axis of each graph gives normalized bandwidth - the actual bandwidth achieved by the flow multiplied by $2n/LinkBandwidth$. Thus a flow achieving a normalized bandwidth of 2.0 would be achieving twice its fair share.

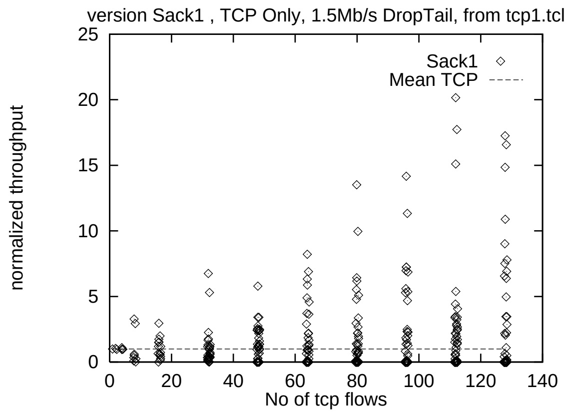
3.2 Control Experiments

We're going to look at the spread of bandwidth of different TFRM flows through the same bottleneck. In order to have a baseline for comparison, we will run a set of control experiments where we run a set of TCP flows in the same topology. The idea is to see whether intra-protocol fairness of TFRM is significantly different from the intra-protocol fairness of TCP.

3.2.1 TCP with 1.5 Mb/s Bottleneck Bandwidth and DropTail Queuing

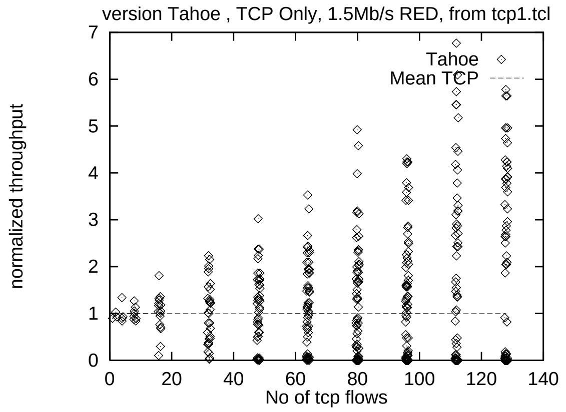


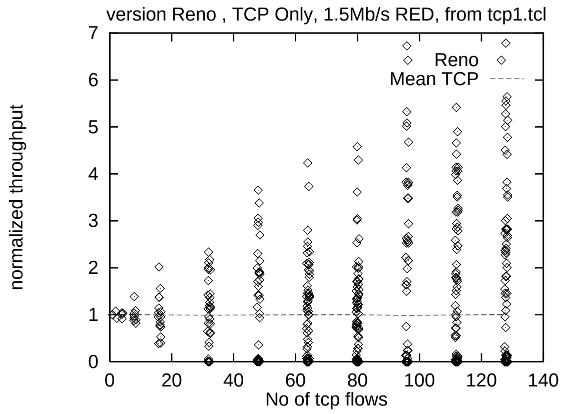


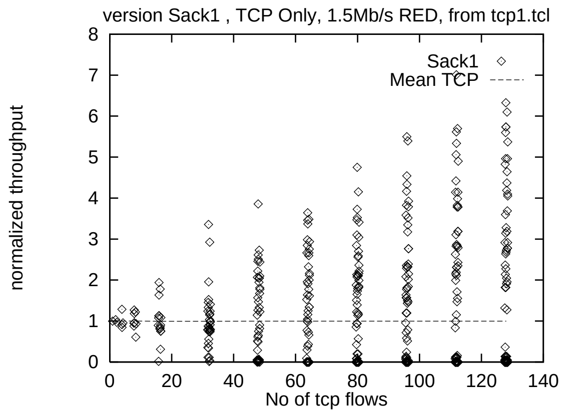


Note: with DropTail queuing, intraprotocol fairness is poor. This may be due in part to phase effects.

3.2.2 TCP with 1.5 Mb/s Bottleneck Bandwidth and RED Queuing

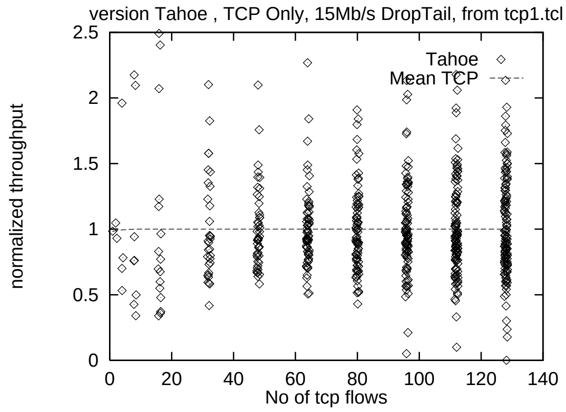


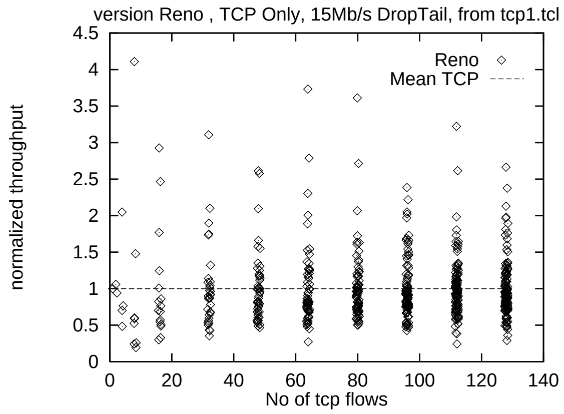


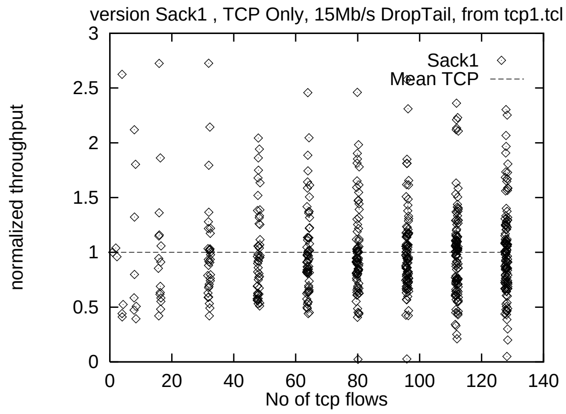


Note: RED significantly improves intra-protocol fairness. This may be because it prevents phase effects becoming persistent.

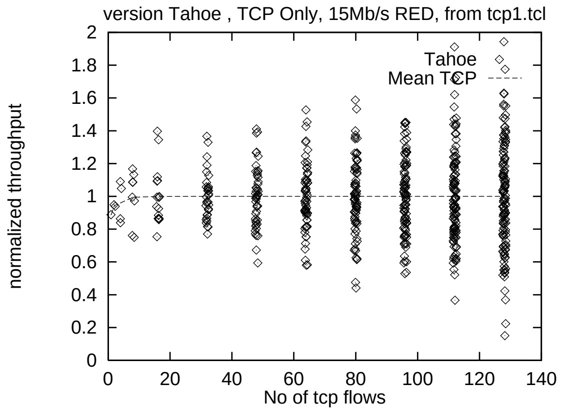
3.2.3 TCP with 15 Mb/s Bottleneck Bandwidth and DropTail Queuing

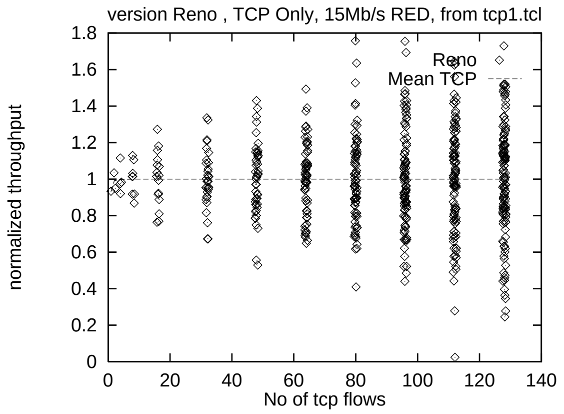


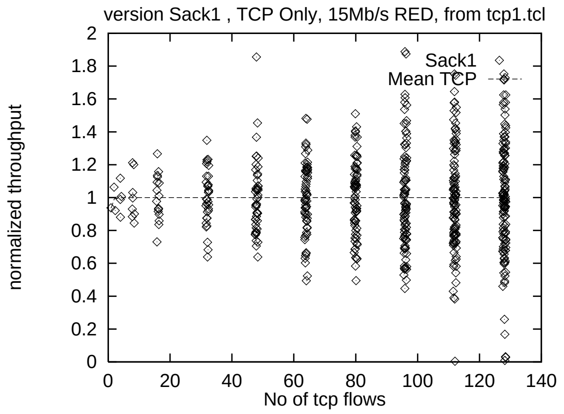




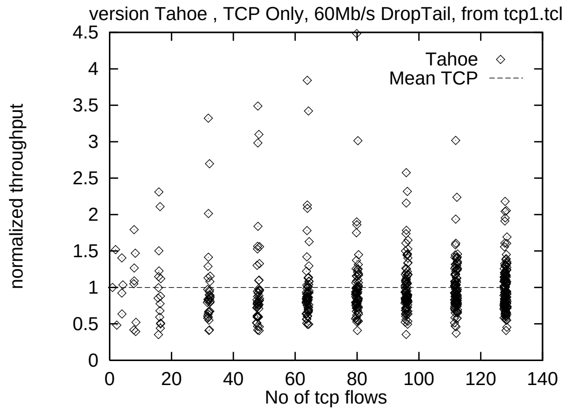
3.2.4 TCP with 15 Mb/s Bottleneck Bandwidth and RED Queuing

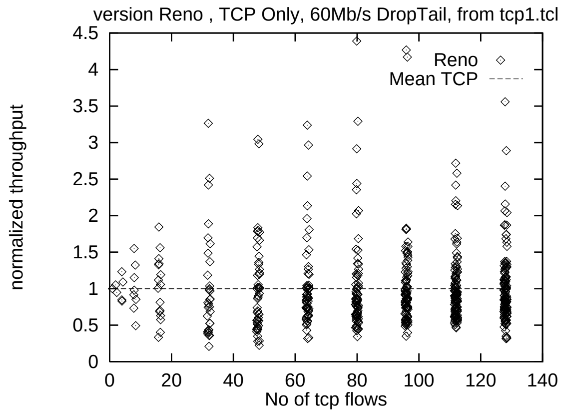


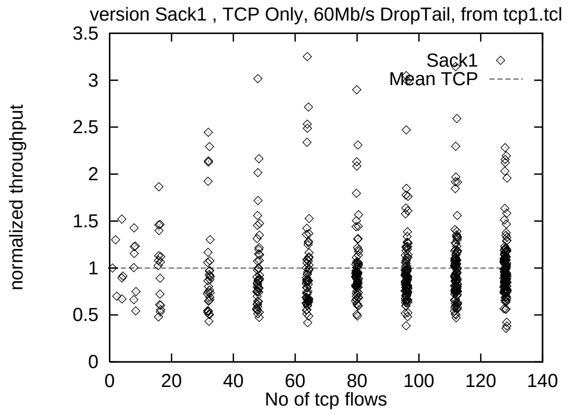




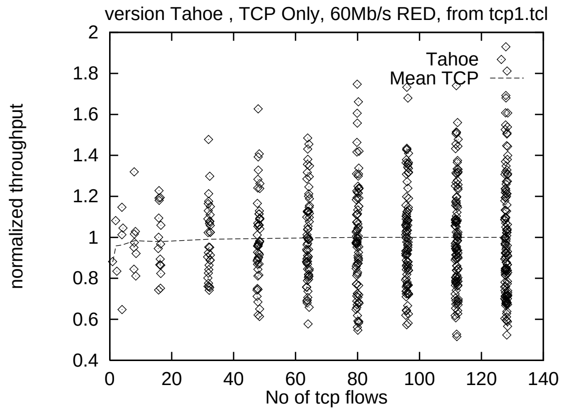
3.2.5 TCP with 60 Mb/s Bottleneck Bandwidth and DropTail Queuing

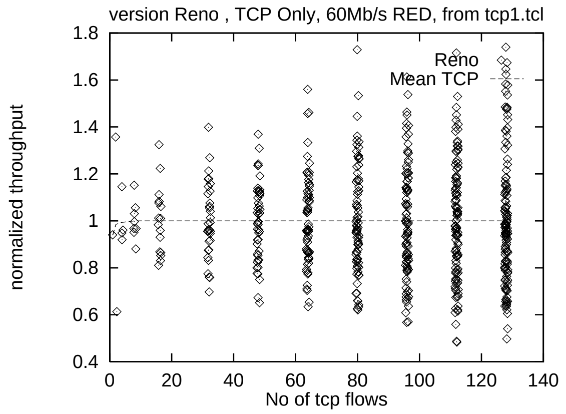


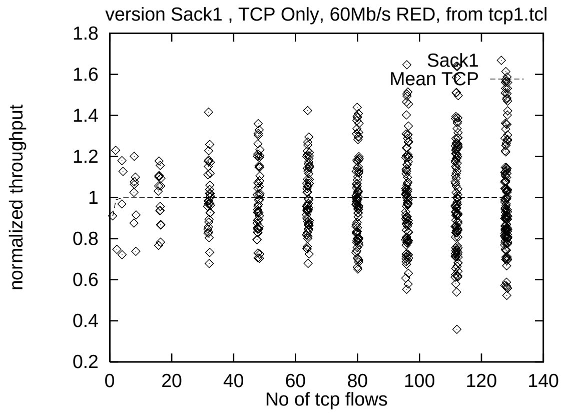




3.2.6 TCP with 60 Mb/s Bottleneck Bandwidth and RED Queuing







3.3 Simulations without slowincr restrictions using TCP Tahoe

3.3.1 1.5 Mb/s Bottleneck Bandwidth

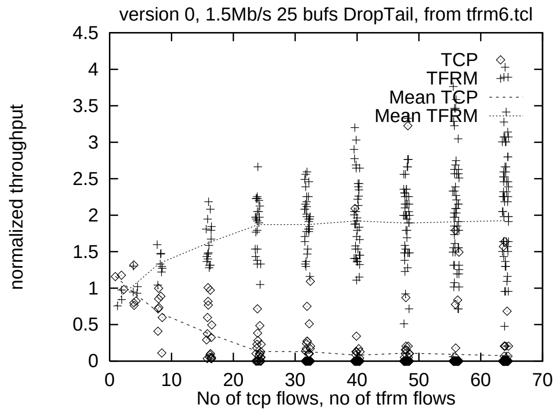


Figure 1:

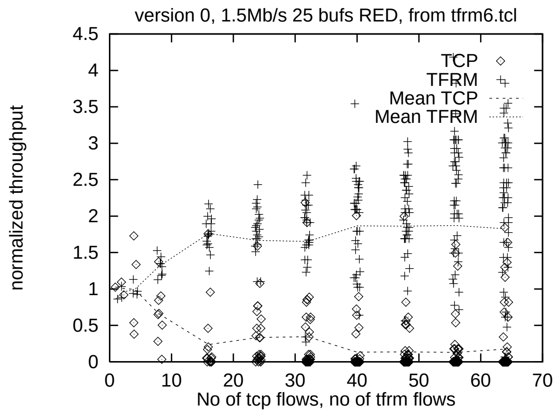


Figure 2:

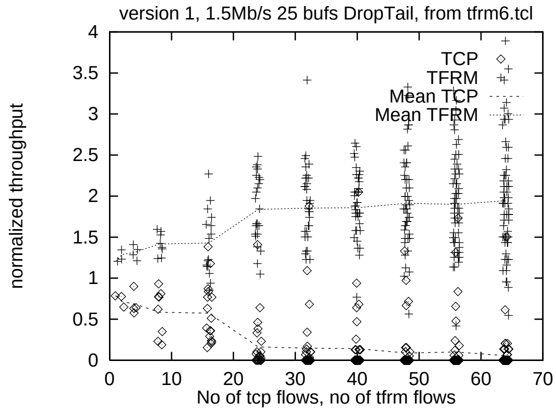


Figure 3:

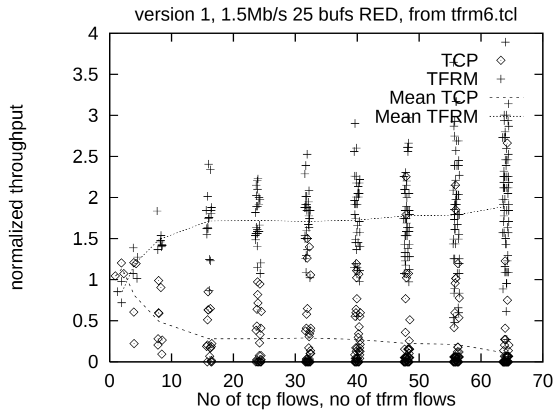


Figure 4:

TFRM is rather too aggressive when there isn't a greate deal of bandwidth. This is probably because the TCP friendly equation used doesn't model timeouts.

3.3.2 15 Mb/s Bottleneck Bandwidth

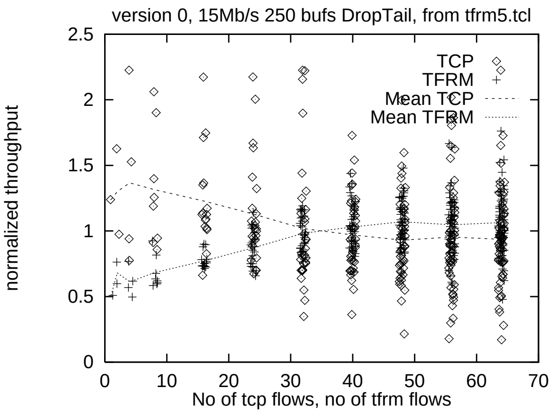


Figure 5:

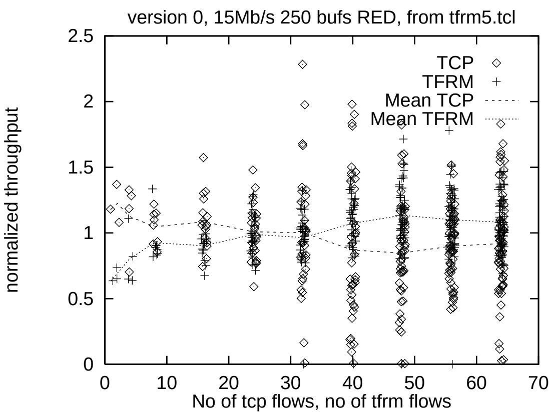


Figure 6:

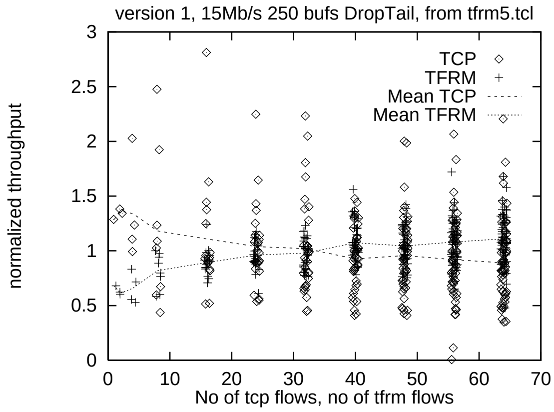


Figure 7:

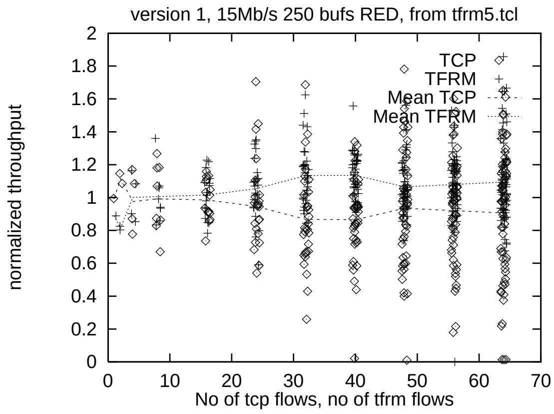


Figure 8:

3.3.3 60 Mb/s Bottleneck Bandwidth

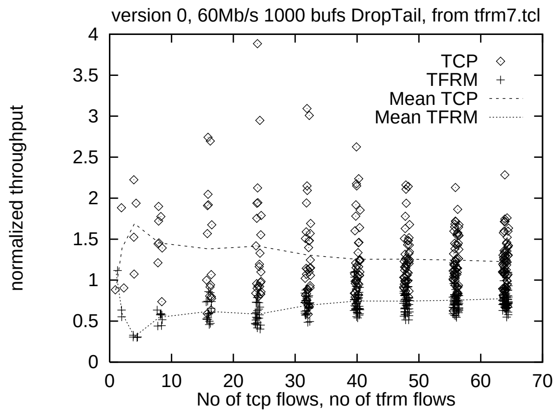


Figure 9:

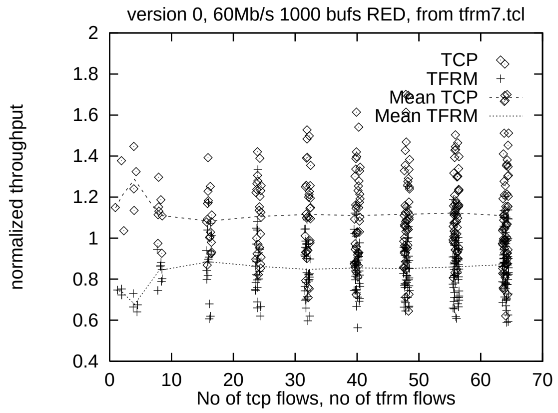


Figure 10:

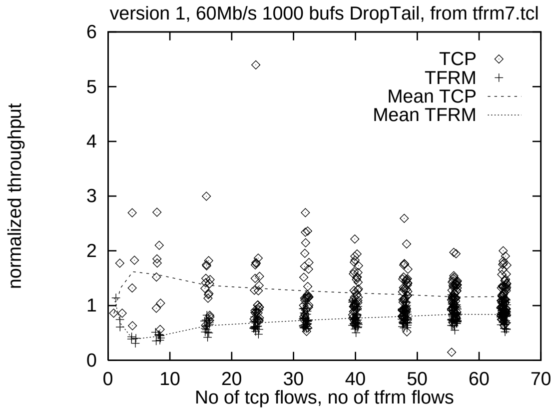


Figure 11:

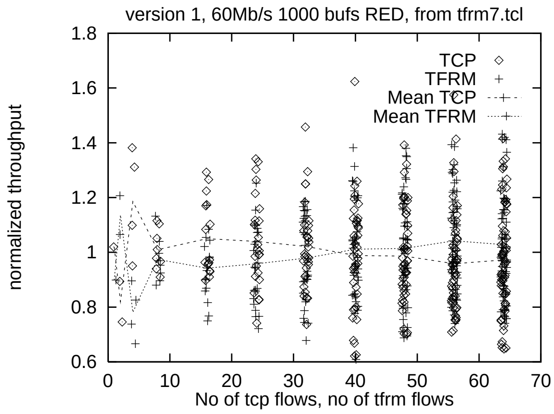


Figure 12:

3.4 Simulations with slowincr restrictions using TCP Tahoe

3.4.1 1.5 Mb/s Bottleneck Bandwidth

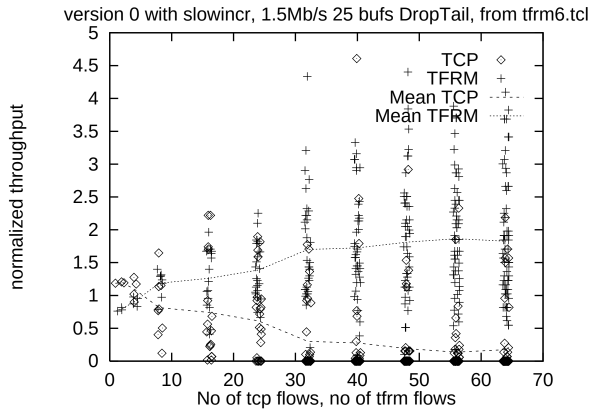


Figure 13:

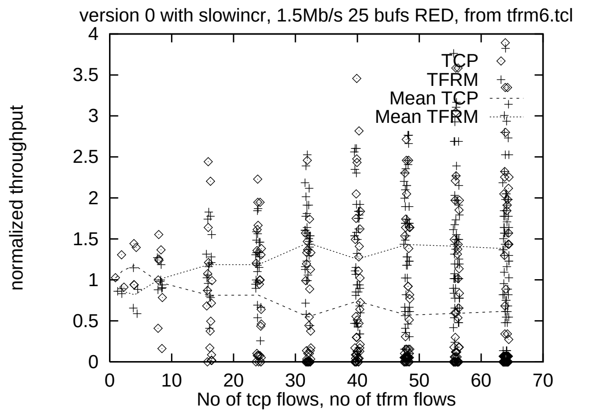


Figure 14:

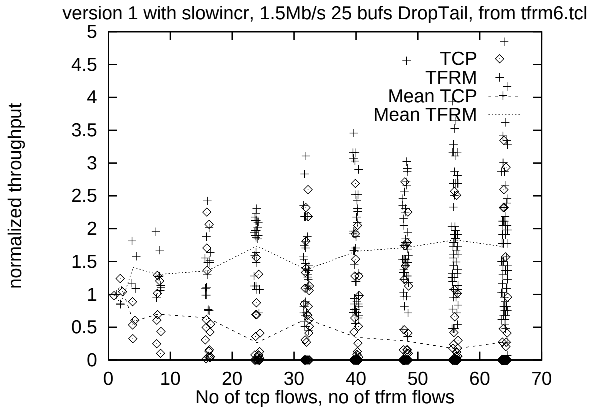


Figure 15:

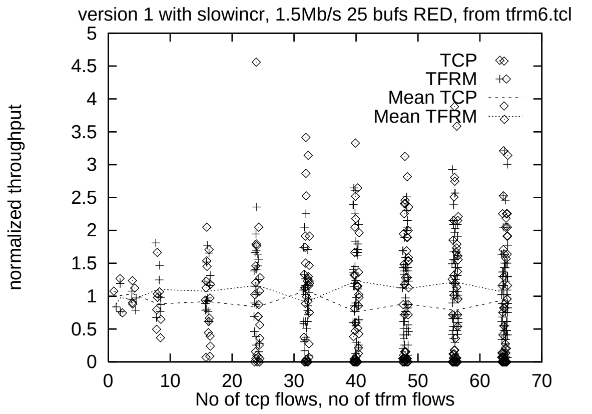


Figure 16:

TFRM is rather too aggressive when there isn't a greate deal of bandwidth. This is probably because the TCP friendly equation used doesn't model timeouts.

3.4.2 15 Mb/s Bottleneck Bandwidth

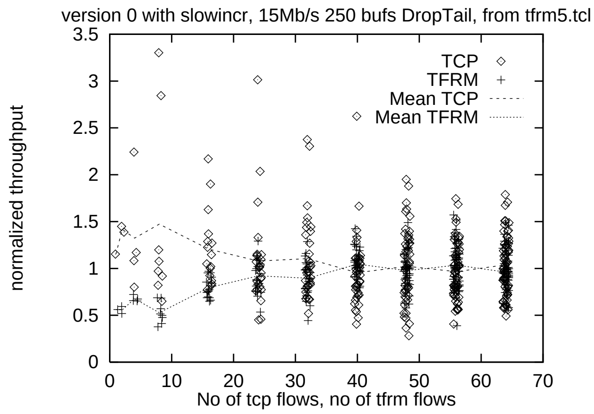


Figure 17:

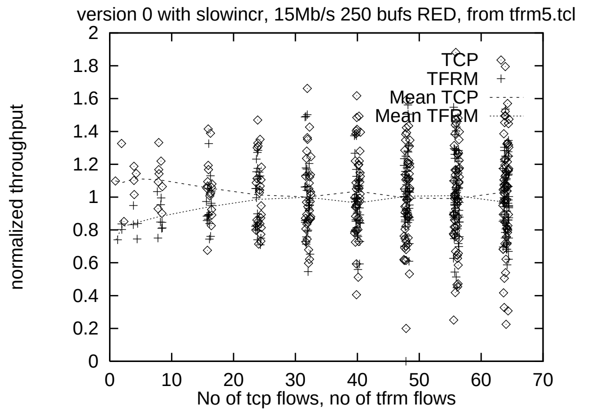


Figure 18:

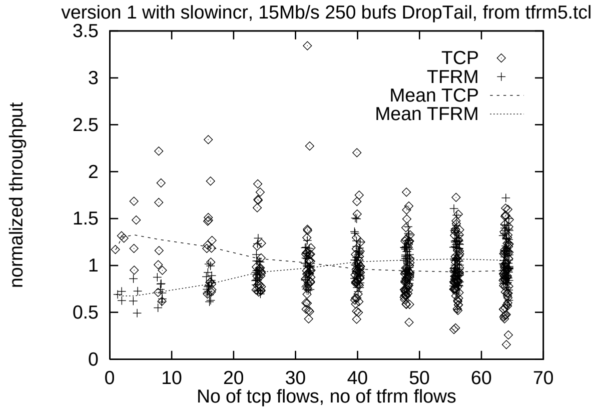


Figure 19:

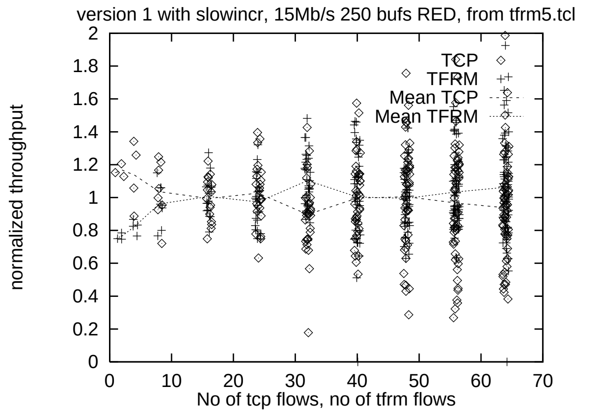


Figure 20:

3.4.3 60 Mb/s Bottleneck Bandwidth

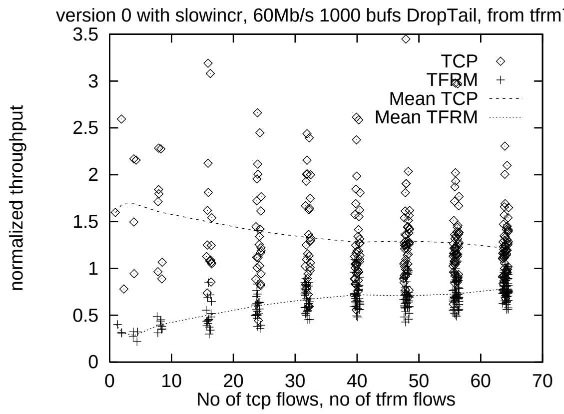


Figure 21:

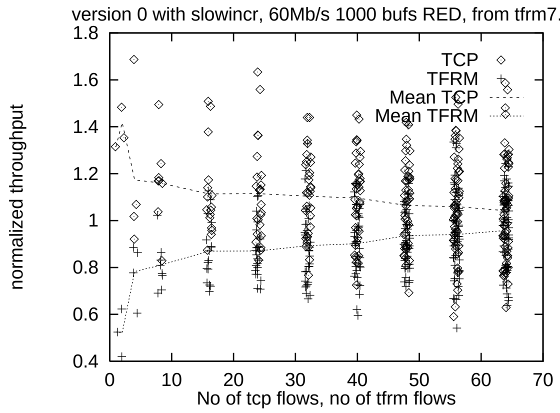


Figure 22:

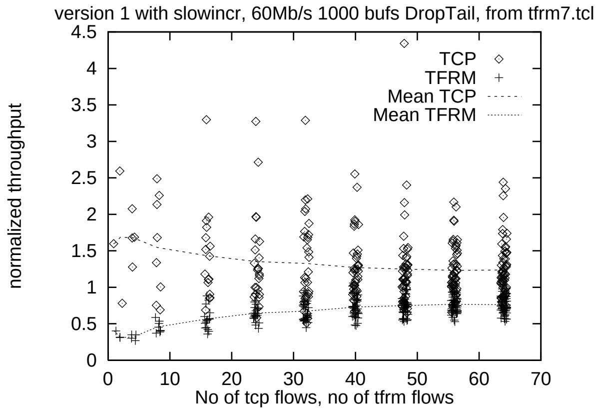


Figure 23:

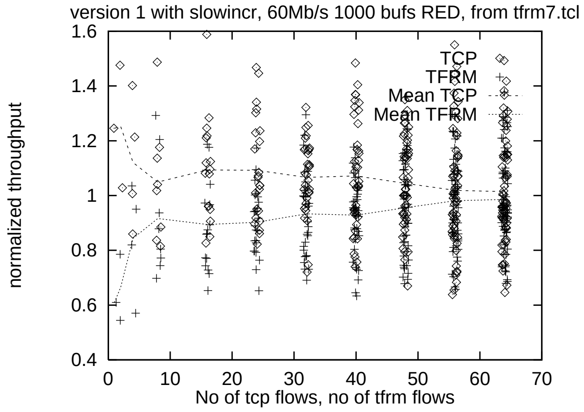


Figure 24:

3.5 Simulations without slowincr restrictions using TCP Reno

3.5.1 1.5 Mb/s Bottleneck Bandwidth

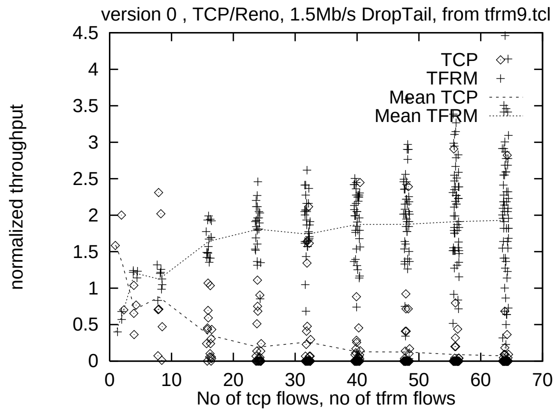


Figure 25:

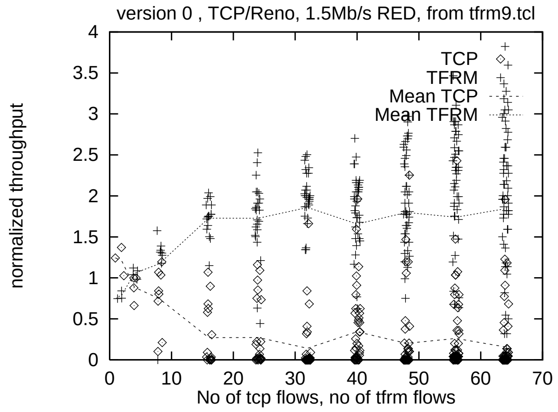


Figure 26:

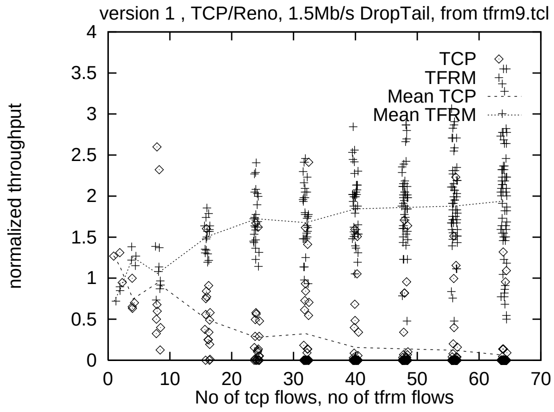


Figure 27:

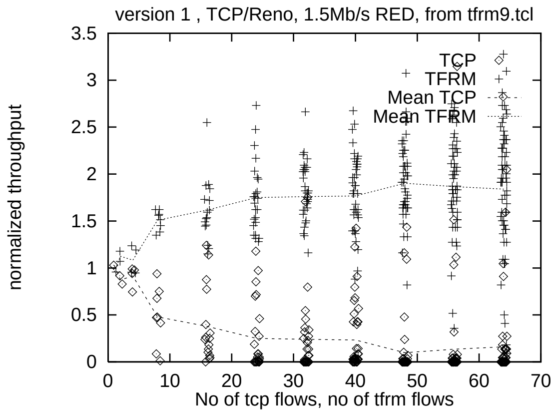


Figure 28:

3.5.2 15 Mb/s Bottleneck Bandwidth

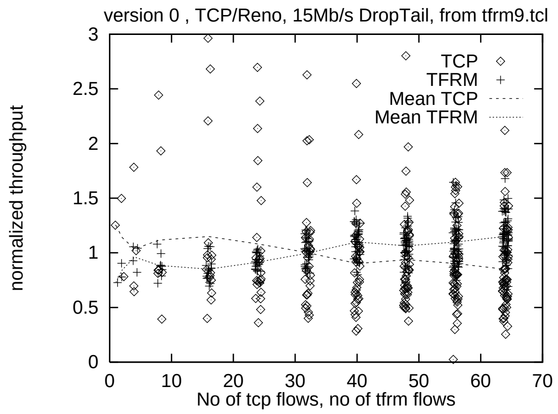


Figure 29:

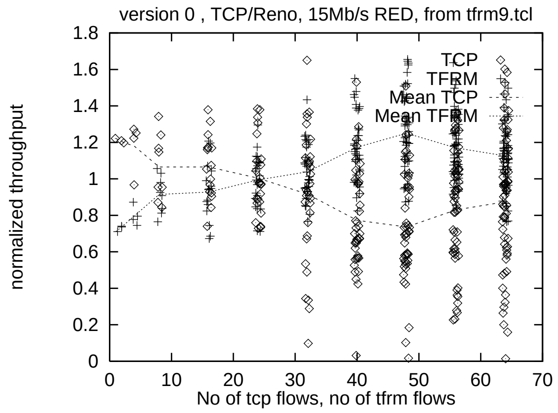


Figure 30:

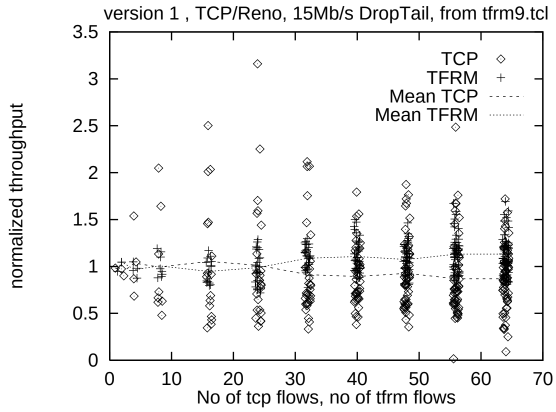


Figure 31:

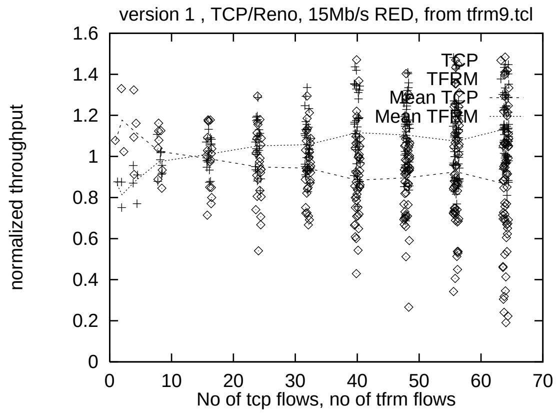


Figure 32:

3.5.3 60 Mb/s Bottleneck Bandwidth

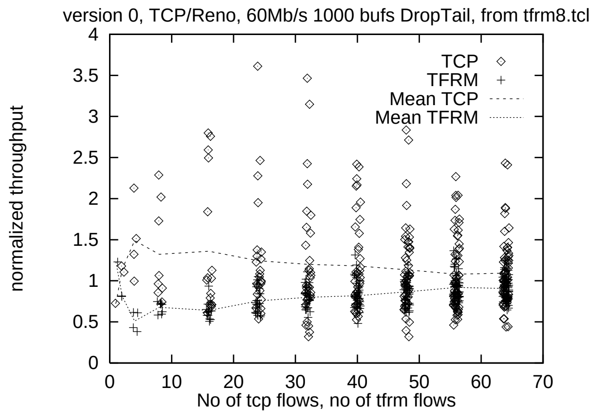


Figure 33:

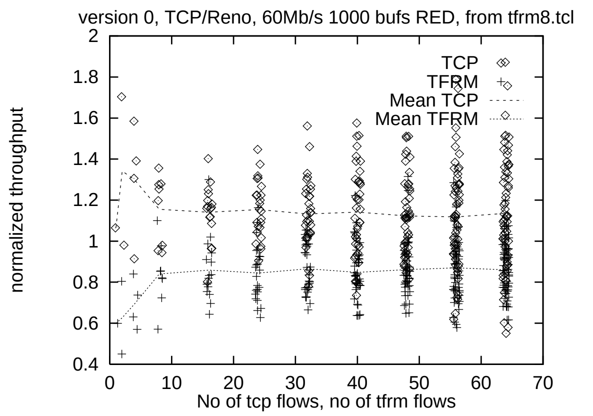


Figure 34:

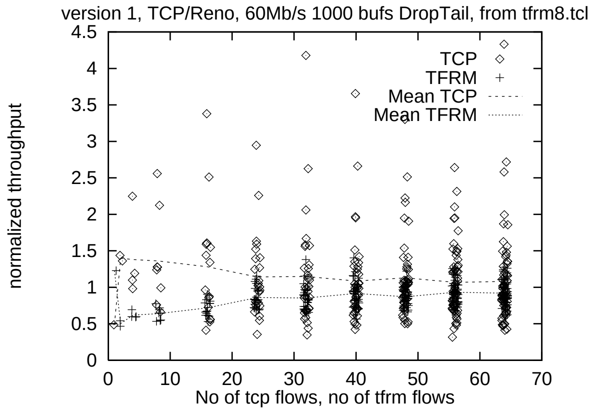


Figure 35:

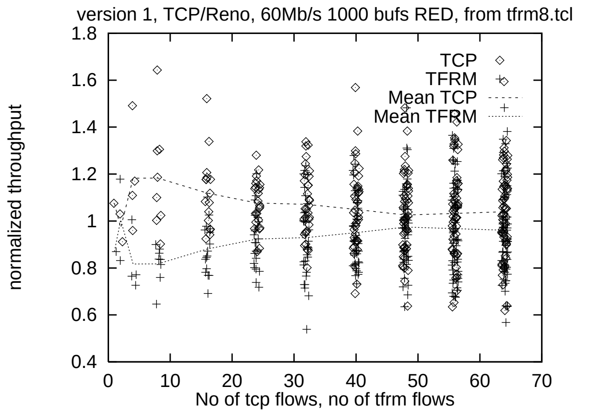


Figure 36:

3.6 Simulations with slowincr restrictions using TCP Reno

3.6.1 1.5 Mb/s Bottleneck Bandwidth

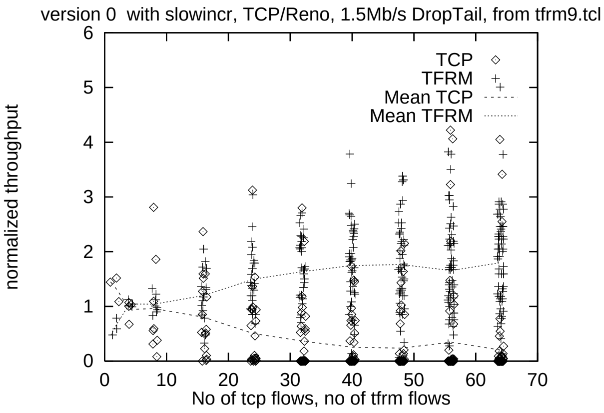


Figure 37:

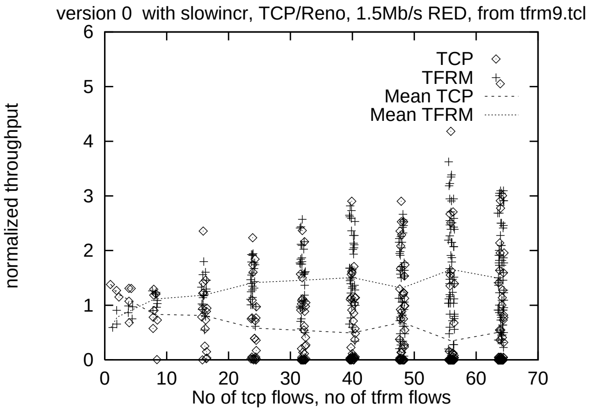


Figure 38:

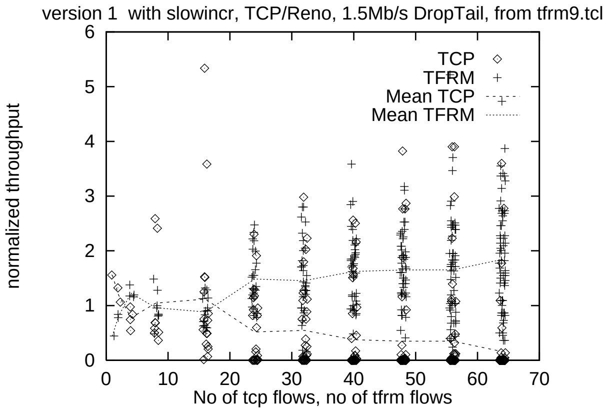


Figure 39:

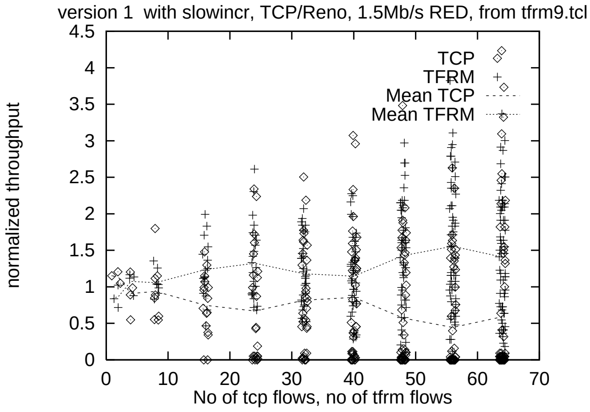


Figure 40:

3.6.2 15 Mb/s Bottleneck Bandwidth

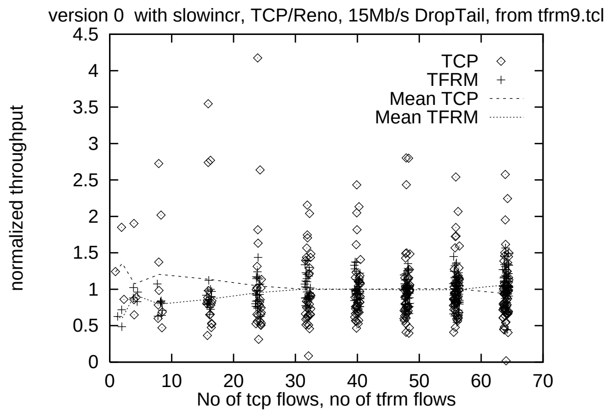


Figure 41:

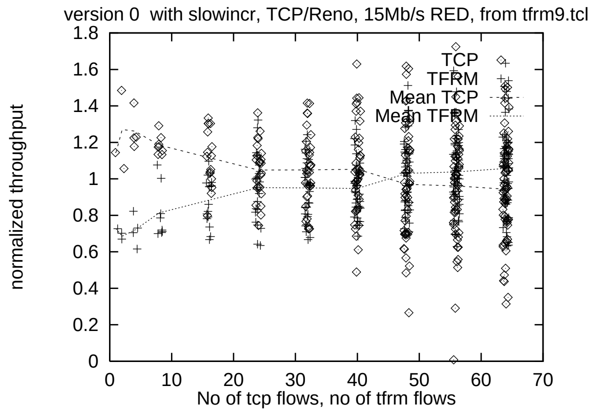


Figure 42:

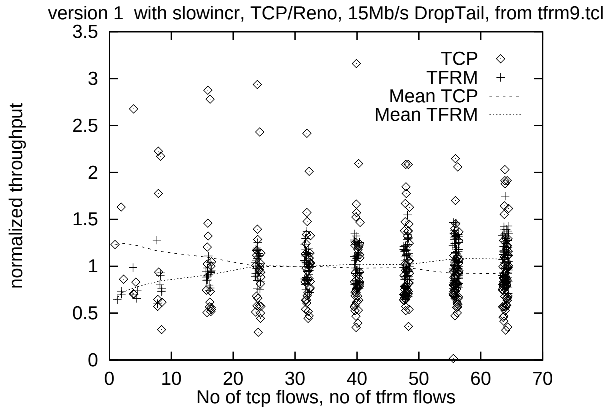


Figure 43:

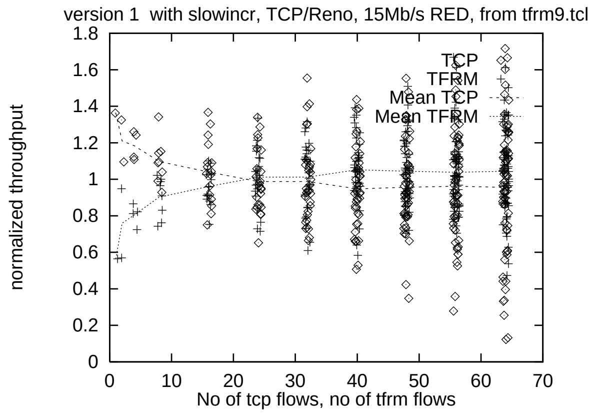


Figure 44:

3.6.3 60 Mb/s Bottleneck Bandwidth

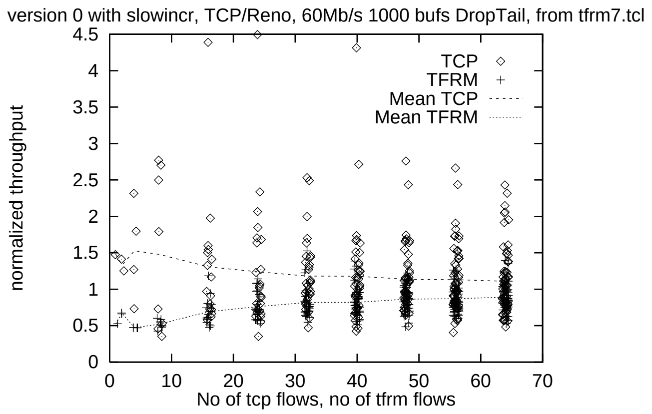


Figure 45:

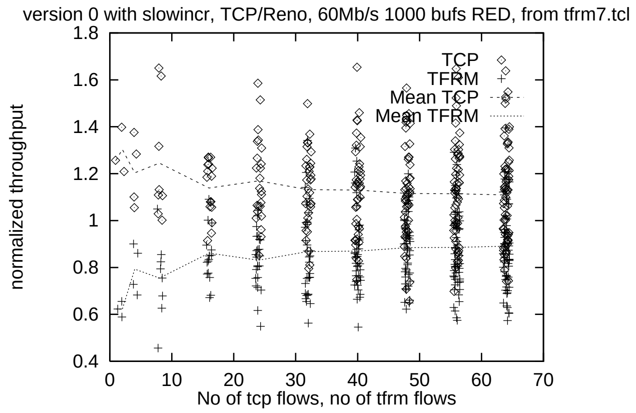


Figure 46:

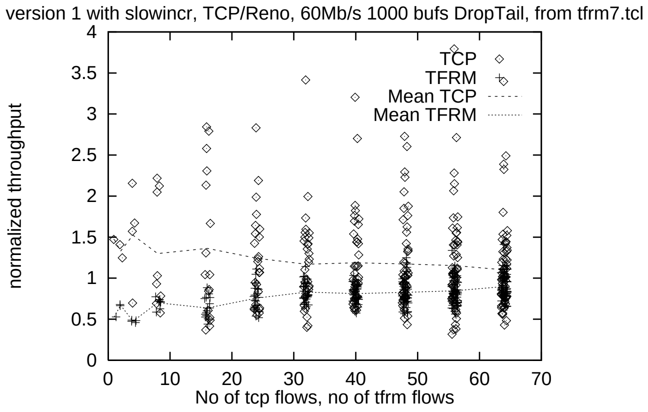


Figure 47:

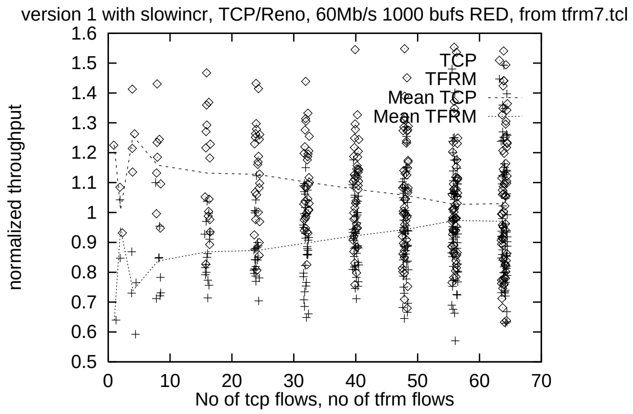


Figure 48:

3.7 Simulations without slowincr restrictions using TCP Sack

3.7.1 1.5 Mb/s Bottleneck Bandwidth

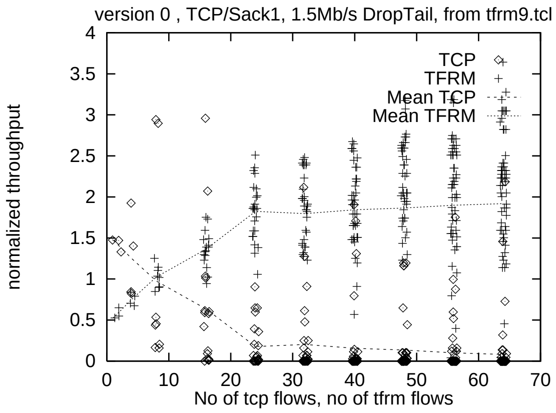


Figure 49:

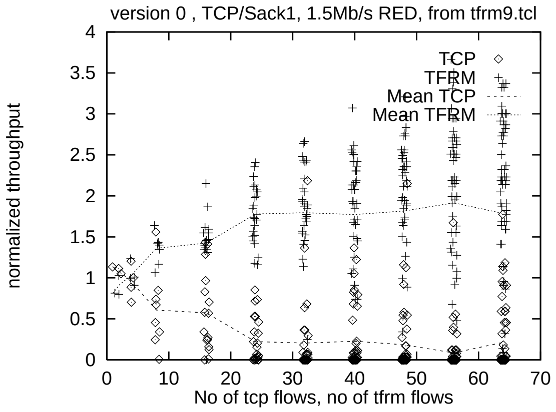


Figure 50:

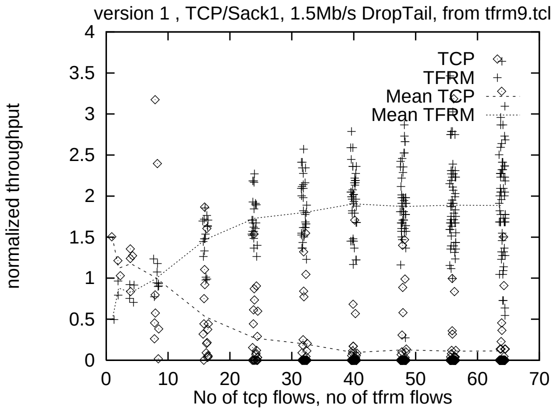


Figure 51:

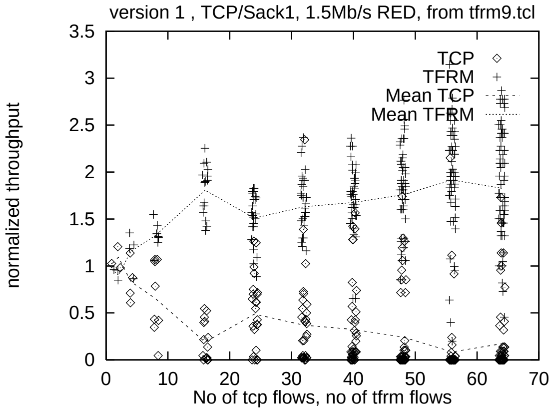


Figure 52:

3.7.2 15 Mb/s Bottleneck Bandwidth

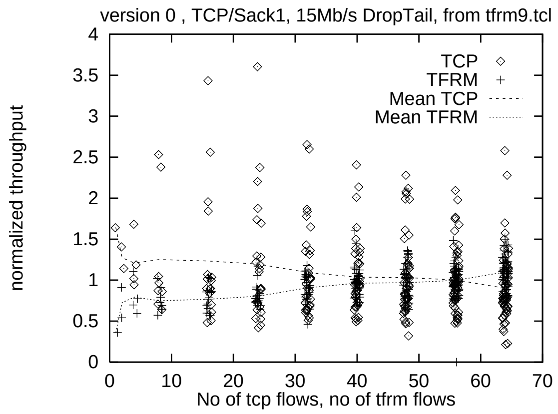


Figure 53:

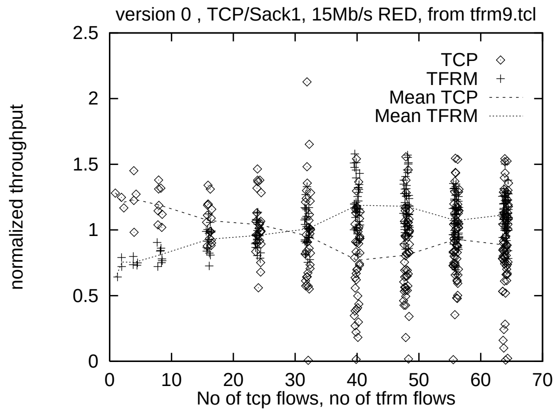


Figure 54:

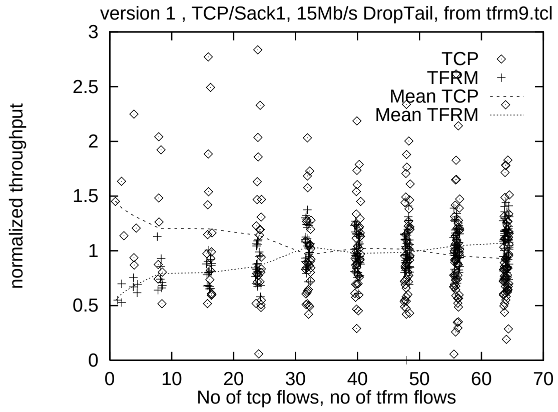


Figure 55:

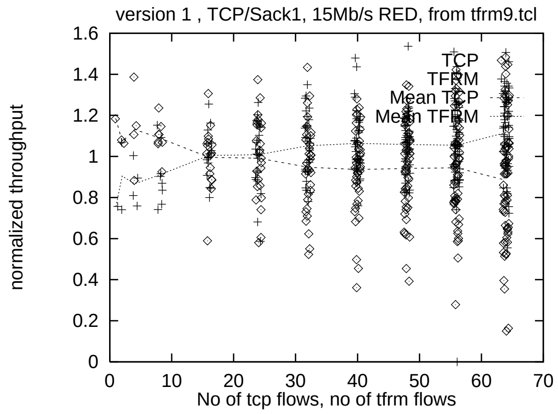


Figure 56:

3.7.3 60 Mb/s Bottleneck Bandwidth

Simulations not run because NS TCP SACK implementation is buggy.

3.8 Simulations with slowincr restrictions using TCP Sack

3.8.1 1.5 Mb/s Bottleneck Bandwidth

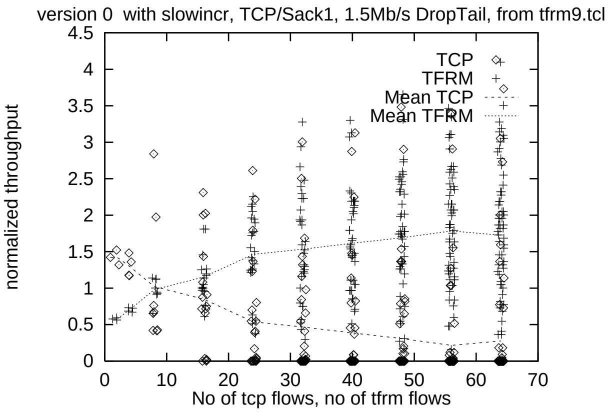


Figure 57:

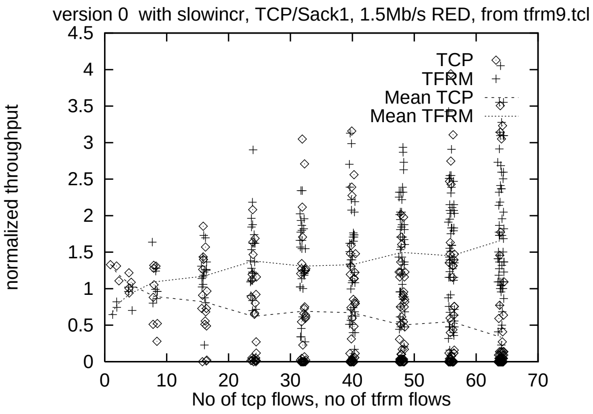


Figure 58:

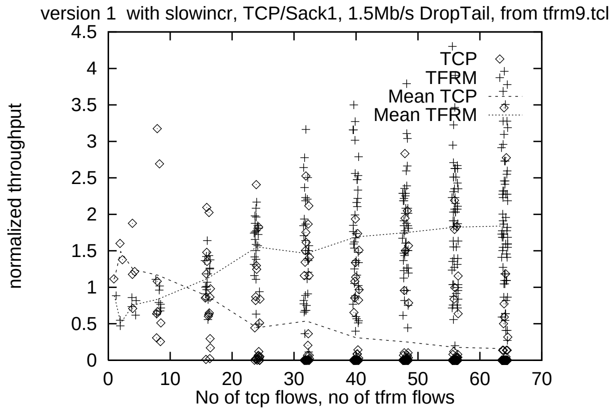


Figure 59:

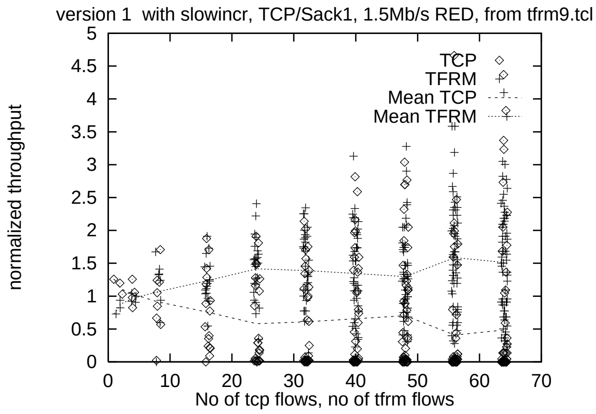


Figure 60:

3.8.2 15 Mb/s Bottleneck Bandwidth

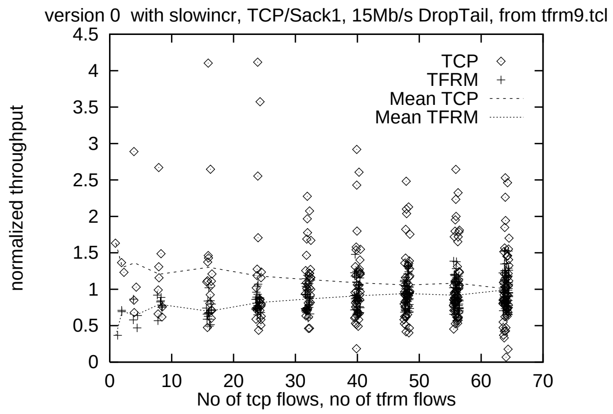


Figure 61:

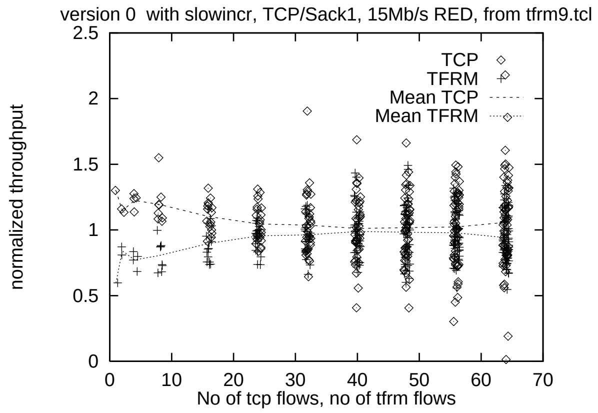


Figure 62:

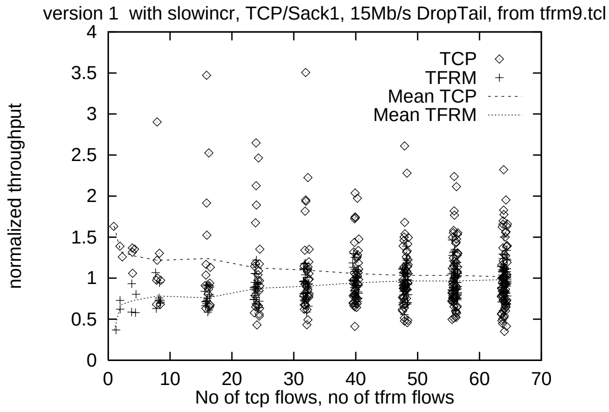


Figure 63:

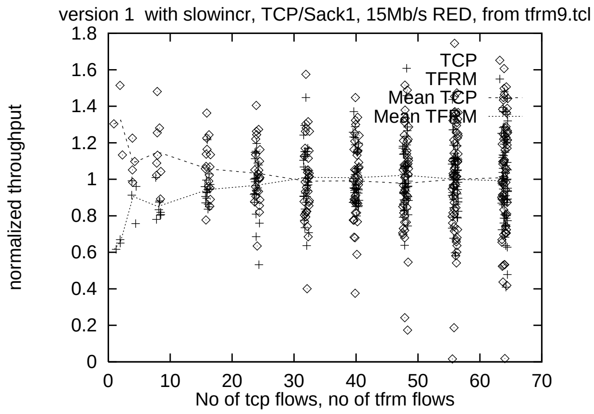


Figure 64:

3.8.3 60 Mb/s Bottleneck Bandwidth

Simulations not run because NS TCP SACK implementation is buggy.

3.9 Simulations of TFRM running alone

The procedure for these simulations was the same as for competing with TCP except that TFRM is running alone. The purpose of the simulations is to examine intra-protocol fairness.

Each simulation is run for 30 seconds of simulated time. The throughput of each flow through the bottleneck is measured in the last 15 seconds to allow flows to get up to speed.

DropTail and RED buffering parameters are as in section 3.1, as are toplogies and round trip times.

3.9.1 1.5 Mb/s Bottleneck Bandwidth, without slow-incr

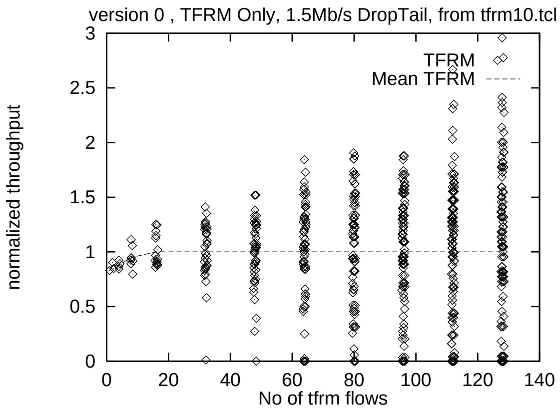


Figure 65:

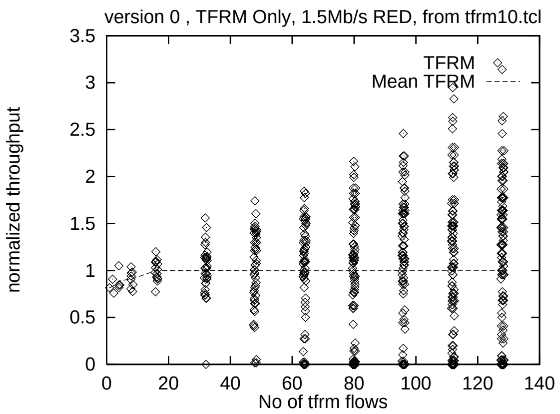


Figure 66:

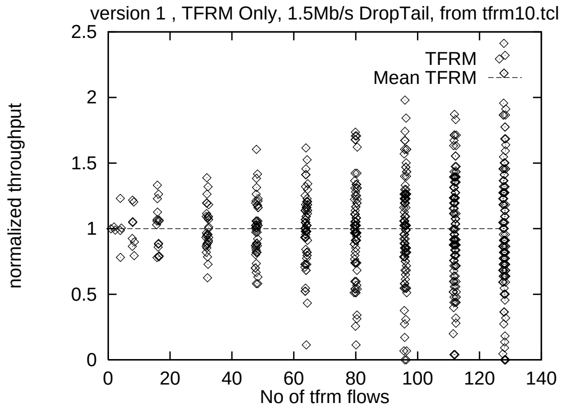


Figure 67:

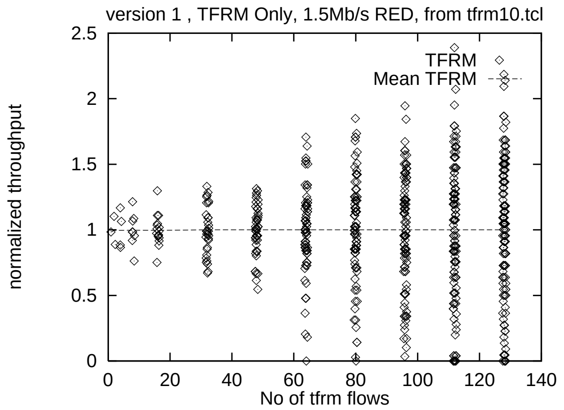


Figure 68:

Note: all simulations suffer cases where some flows achieve zero throughput. RED does not help this problem. Version 1 is less bad than version 0.

3.9.2 1.5 Mb/s Bottleneck Bandwidth, with slowincr

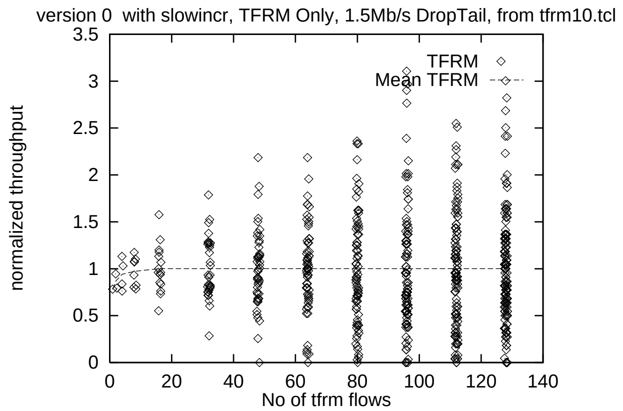


Figure 69:

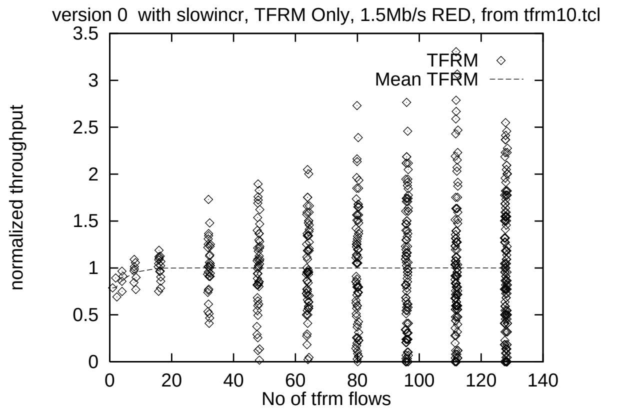


Figure 70:

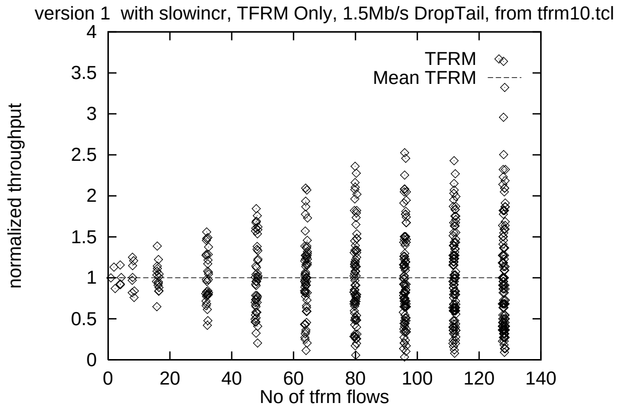


Figure 71:

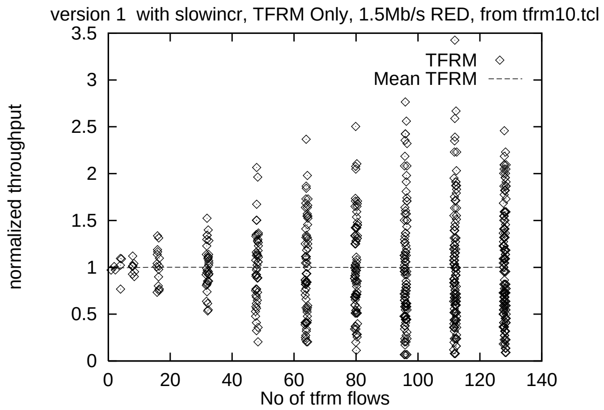


Figure 72:

Note: all simulations suffer cases where some flows achieve zero throughput. RED does not help this problem. Slowincr helps this problem slightly. Version 1 is less bad than version 0.

3.9.3 15 Mb/s Bottleneck Bandwidth, without slow-incr

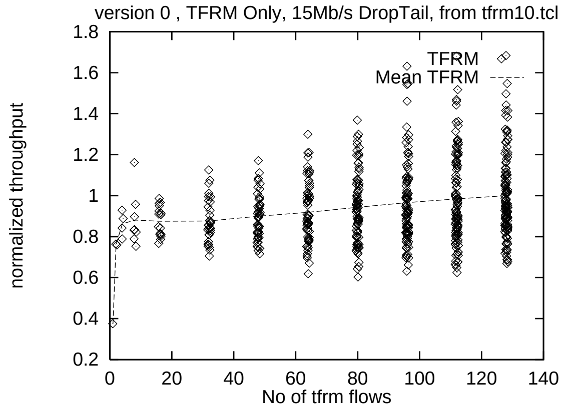


Figure 73:

Note: TFRM Version 0 alone does not achieve 100% utilization except with very large number of flows.

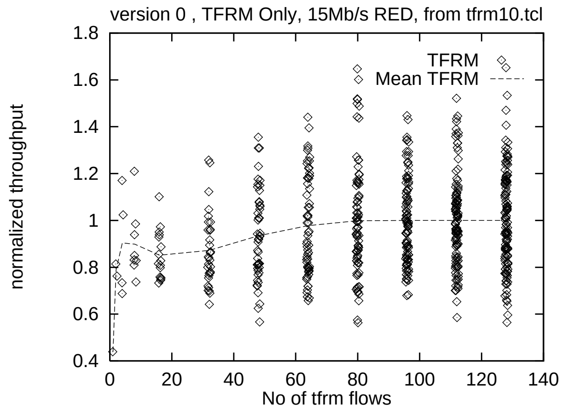


Figure 74:

Note: TFRM Version 0 alone does not achieve 100% utilization except with very large number of flows. RED does not help this problem.

Thus under-utilization is caused by strong oscillatory behaviour:

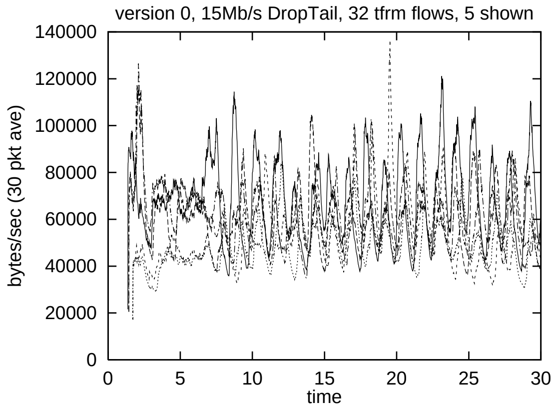


Figure 75:

RED is not sufficient to de-synchonize these flows and prevent the oscillation:

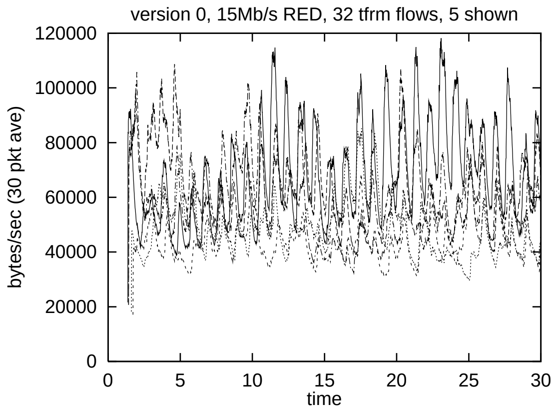


Figure 76:

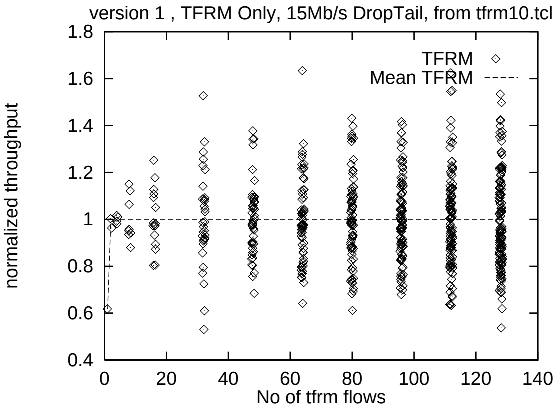


Figure 77:

Note: TFRM Version 1 does not achieve 100% utilization with a single flow.

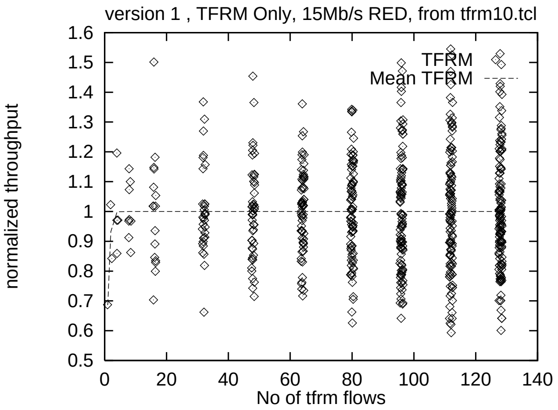


Figure 78:

Note: TFRM Version 1 does not achieve 100% utilization with a single flow. RED marginally improves this problem.

Version 1 does not appear to oscillate:

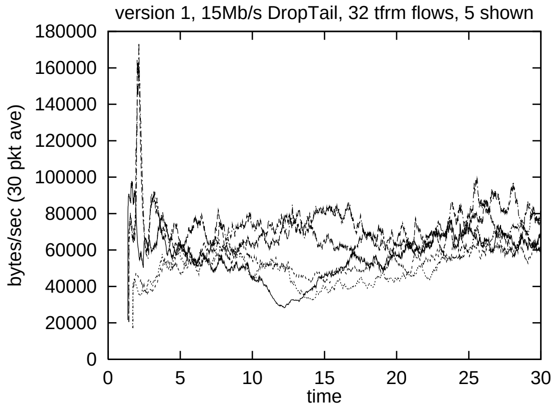


Figure 79:

The poor utilization with a small number of flows is caused by poor startup behaviour:

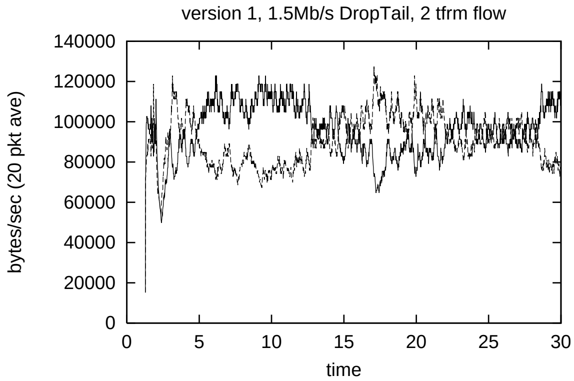


Figure 80:

3.9.4 15 Mb/s Bottleneck Bandwidth, with slowincr

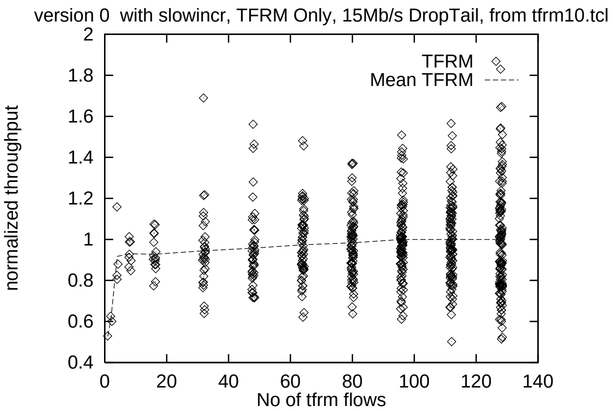


Figure 81:

Note: TFRM Version 0 alone does not achieve 100% utilization except with very large number of flows. Slowincr only marginally helps this problem.

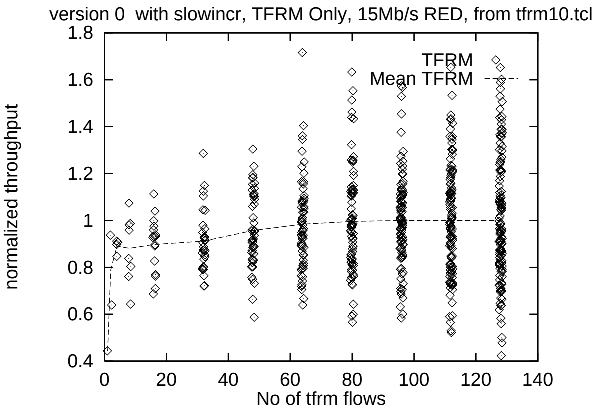


Figure 82:

Note: TFRM Version 0 alone does not achieve 100% utilization except with very large number of flows. Slowincr only marginally helps this problem. RED does not help this problem.

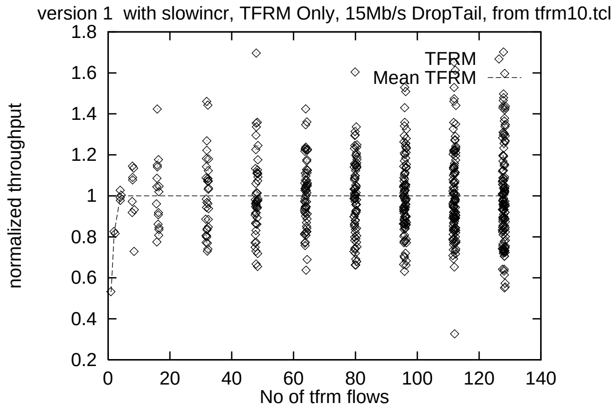


Figure 83:

Note: TFRM Version 1 does not achieve 100% utilization with a single flow. Slowincr makes this problem marginally worse.

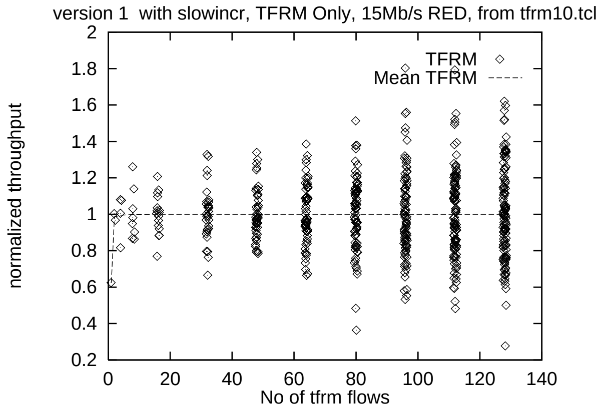


Figure 84:

Note: TFRM Version 1 does not achieve 100% utilization with a single flow. Slowincr makes this problem marginally worse. RED marginally improves this problem.

3.9.5 60 Mb/s Bottleneck Bandwidth, without slow-incr

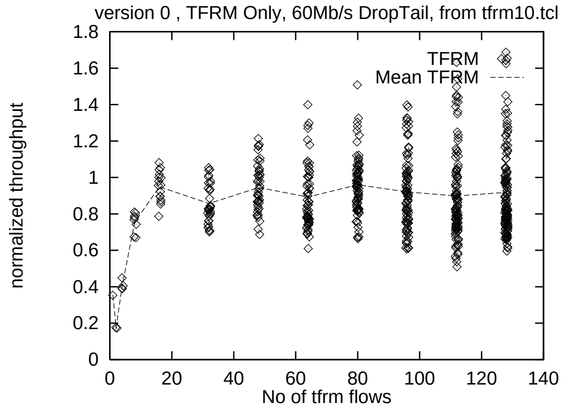


Figure 85:

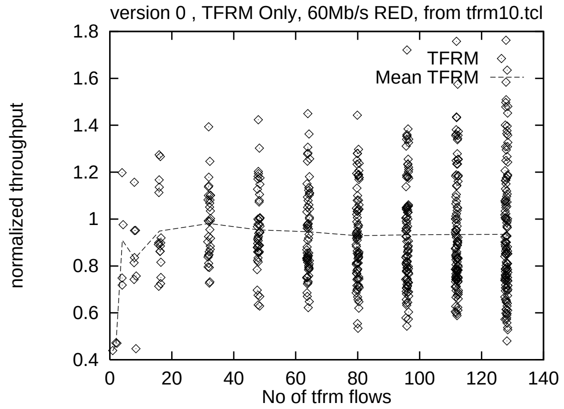


Figure 86:

Note: link does not reach 100% utilization with version 0.

Examining the cause for bottleneck underutilization reveals that version 0 is oscillating:

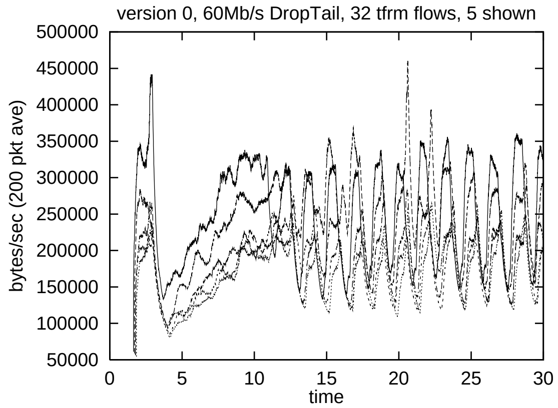


Figure 87:

With RED queuing, this oscillation is not nearly as pronounced, but short term oscillations still occur:

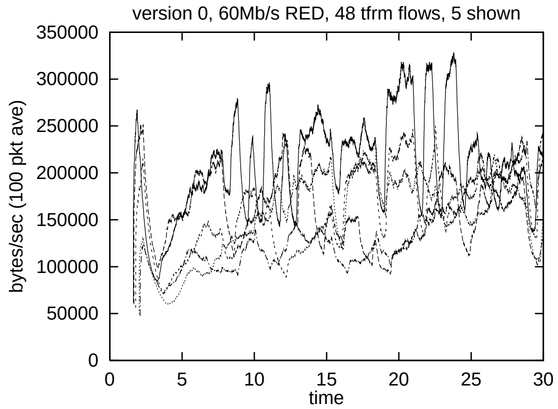


Figure 88:

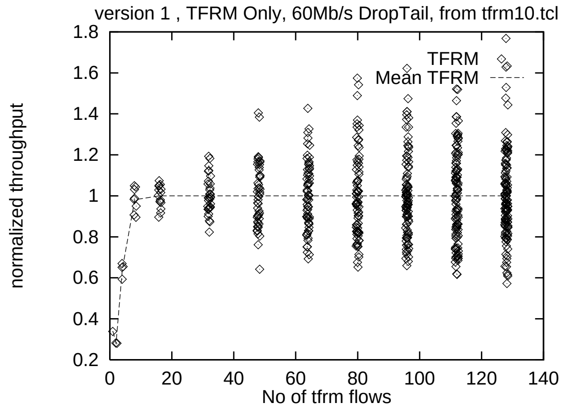


Figure 89:

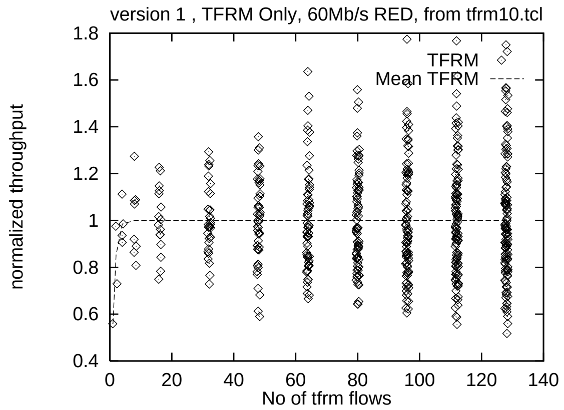


Figure 90:

With version 1, link is only underutilized with small numbers of flows.

It appears that TFRM needs some attention paid to startup behaviour. It takes a long time to fill an empty link, and then when it causes loss, takes even longer to reach a steady operating point:

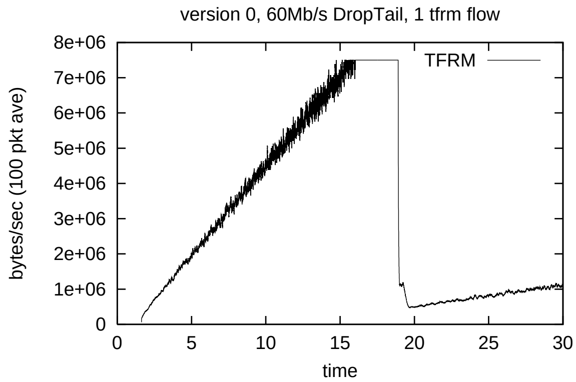


Figure 91:

Note that the partial oscillations seem with version 0 are not present with version 1:

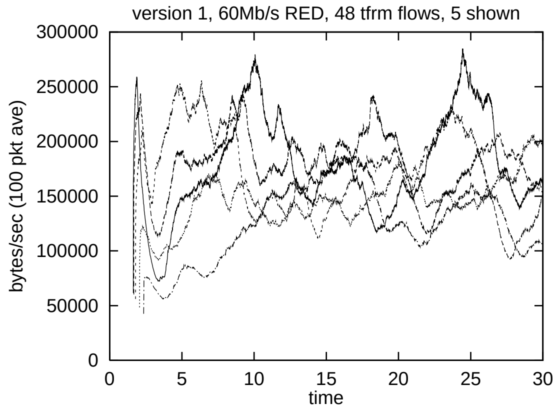


Figure 92:

3.9.6 60 Mb/s Bottleneck Bandwidth, with slowincr

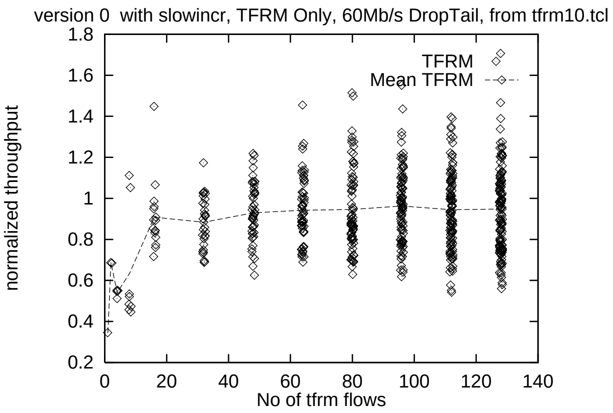


Figure 93:

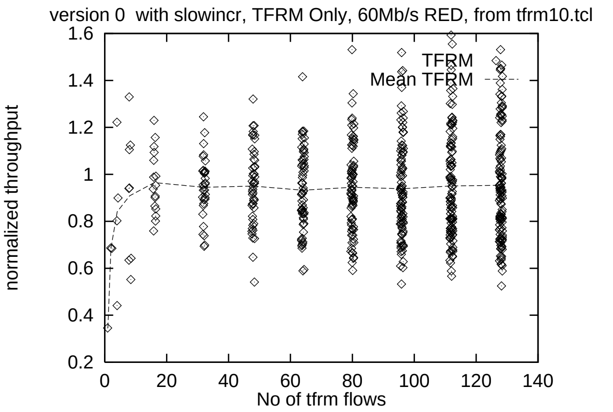


Figure 94:

Note: link does not reach 100% utilization with version 0.

Examining why this might be the case, we see that slowincr has failed to completely prevent oscillation:

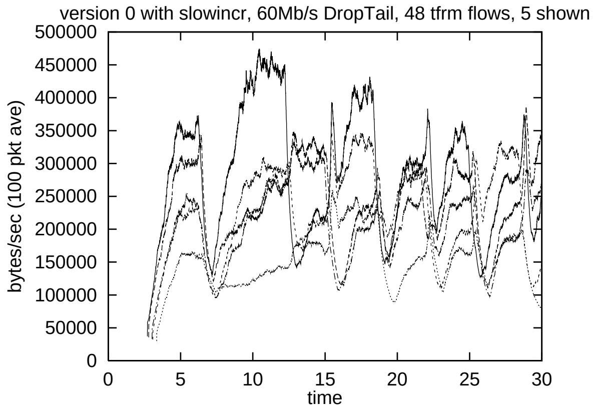


Figure 95:

Even with slowincr and RED we still see partial short term-oscillation:

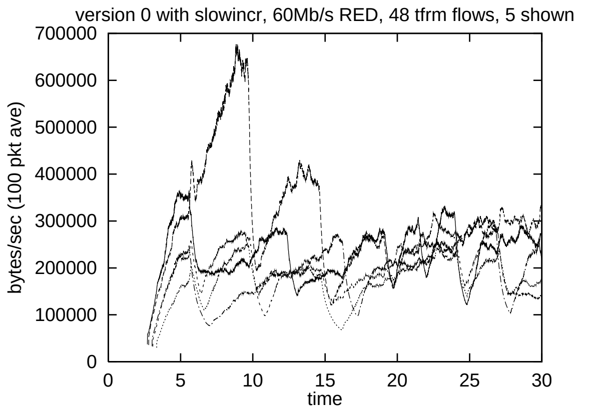


Figure 96:

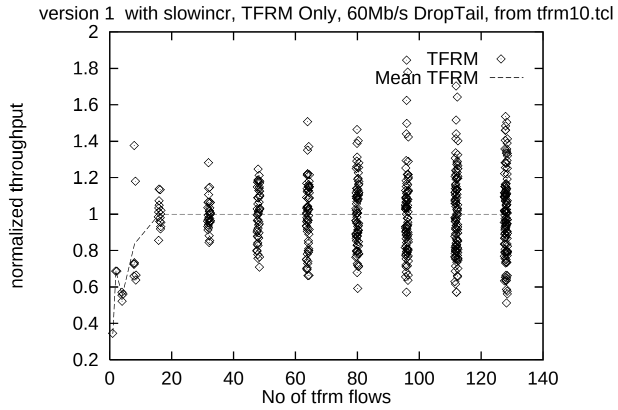


Figure 97:

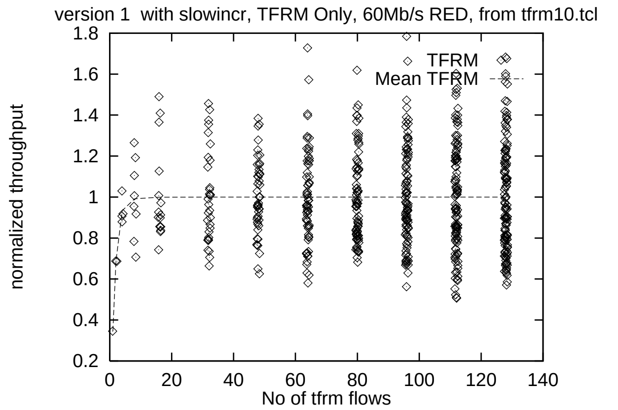


Figure 98:

With version 1, link is only underutilized with small numbers of flows. The reason is as before - poor startup behaviour. Slowincr does not make a significant difference.

4 Using the Padhye/Firoiu/Towsley/Kurose Equation

In [1], a new TCP bandwdith equation is described loosely modelling Tahoe TCP including the retransmit timeouts. The original Floyd/Mathis/Mahdavi equation does not model retransmit timeouts and so does not work well at los bandwidths as shown in the 1.5Mb/s equations in sections 3.3.1 and 3.4.1.

We choose the approximate solution from [1] and ignore the term due to receiver window limiting - this is a self-imposed handicap unrelated to network congestion, so we can safely ignore it. The equation given is:

$$B(l) = \frac{s}{t_{RTT}\sqrt{\frac{2bl}{3}} + t_0 min\left(1, 3\sqrt{\frac{3bl}{8}}\right) l(1+32l^2)} \quad (2)$$

As before, l is the loss fraction, $B(1)$ is the data rate in bytes/sec, and s is the packet size in bytes. In addition, b is the number of packets acknowledged by each TCP ACK, and t_0 is the TCP restransmission timeout value.

At the receiver, to obtain the number of sample packets to examine the loss fraction over, we need the inverse of this function, i.e., given B(l), we need l. In these simulations, we solve this numerically using a binary search technique.

Given a loss fraction across k/l_{exp} sample packets, we then simulation both versions of the algorithm as before. Version 1 now converts the feedback from the receiver to expected bandwidth using equation 2 rather than 1 which we previously used.

To apply this equation, we need a reasonable estimate of t_0 at both the sender and receiver. The sender calculates this value using the TCP Tahoe code, which essentially implements the following:

$$t_{SRTT,i+1} = \frac{7}{8}t_{SRTT,i} + \frac{1}{8}t_{RTT}$$

$$t_{VAR,i+1} = \frac{3}{4}t_{VAR,i} + \frac{1}{4}|t_{RTT} - t_{SRTT,i}|$$

However in TCP, this calculation is performed with a course grain clock. The precise code fragment we use is taken from the ns TCP Tahoe implementation:

```
t_rtt_ =int((now-nck->timestamp_echo) /tcp_tick_ + 0.5);
if (t_rtt_==0) t_rtt_=1;
if (t_srtt_ != 0) {
   register short delta;
   delta = t_rtt_ - (t_srtt_ >> T_SRTT_BITS);
   if ((t_srtt_ += delta) <= 0)
       t_srtt_ = 1;
   if (delta < 0)
       delta = -delta;
   delta -= (t_rttvar_ >> T_RTTVAR_BITS);
   if ((t_rttvar_ += delta) <= 0)
       t_rttvar_ = 1;
} else {
   t_srtt_ = t_rtt_ << T_SRTT_BITS;
   t_rttvar_ = t_rtt_ << (T_RTTVAR_BITS-1);
}
t_rtxcur_ = (((t_rttvar_ << (rttvar_exp_ +
       (T_SRTT_BITS - T_RTTVAR_BITS))) +
       t_srtt_) >> T_SRTT_BITS ) * tcp_tick_;
tzero_=t_rtxcur_;
```

T_SRTT_BITS is 3, T_RTTVAR_BITS is 2, tcp_tick_ is 0.1, rttvar_exp_ is 2.

One cause for concern is that different TCP implementations use different clock granularities to calculate t_0. There is concern that the fairness may be very dependent on accurately modelling t_0.

4.1 Simulations using TCP Reno

This simulations do not use TCP delayed ACK and use a value of $b = 1$.

4.1.1 1.5Mb/s Bottleneck Bandwidth without slow-incr

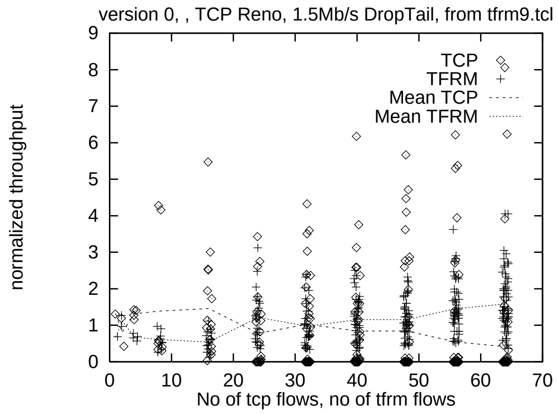


Figure 99:

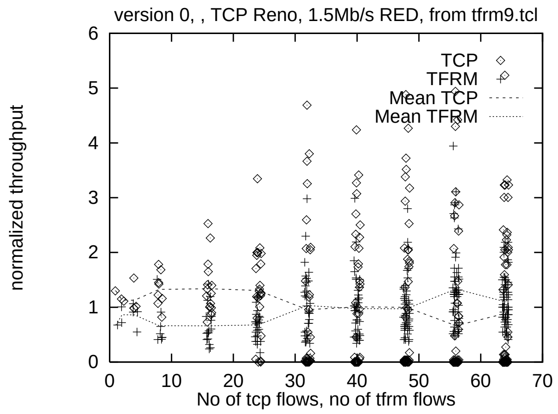


Figure 100:

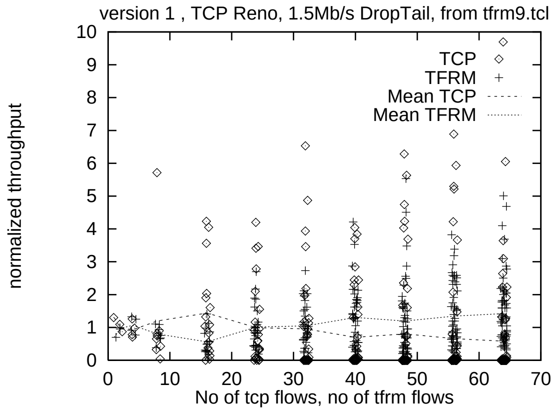


Figure 101:

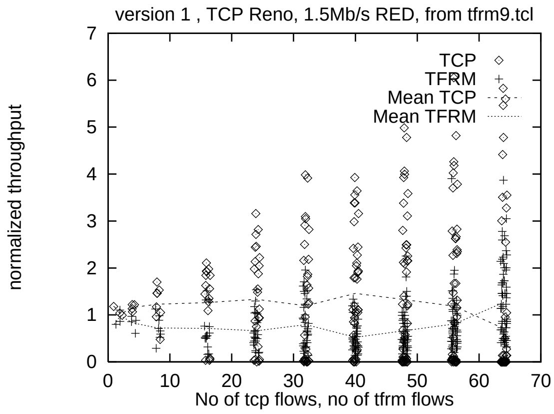


Figure 102:

4.1.2 1.5Mb/s Bottleneck Bandwidth with slowincr

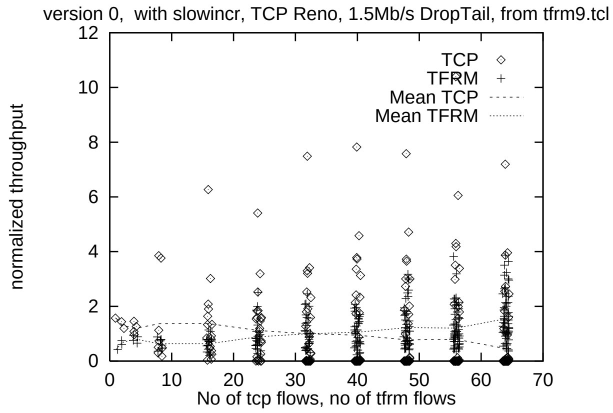


Figure 103:

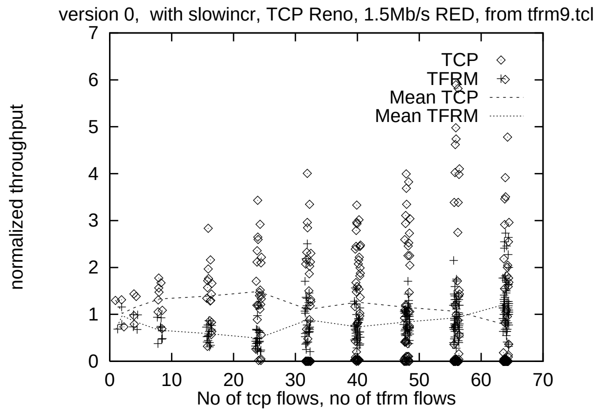


Figure 104:

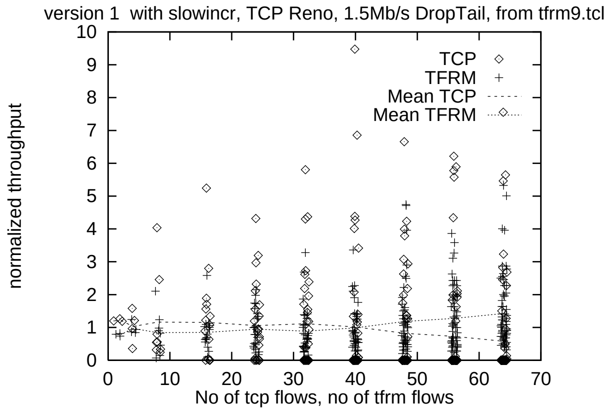


Figure 105:

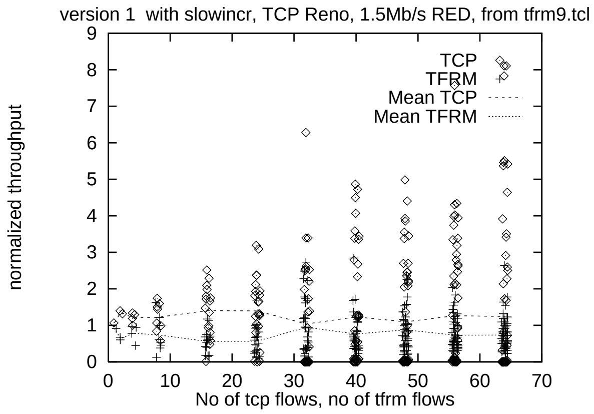


Figure 106:

4.1.3 15Mb/s Bottleneck Bandwidth without slow-incr

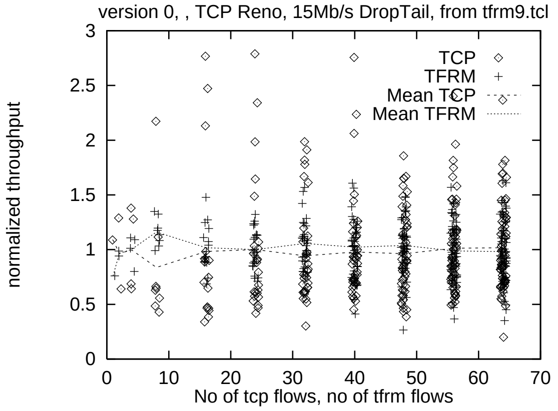


Figure 107:

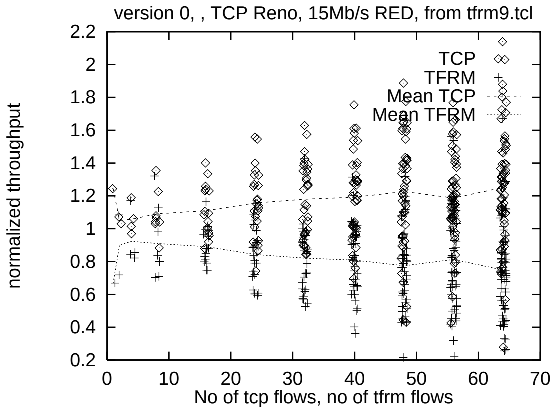


Figure 108:

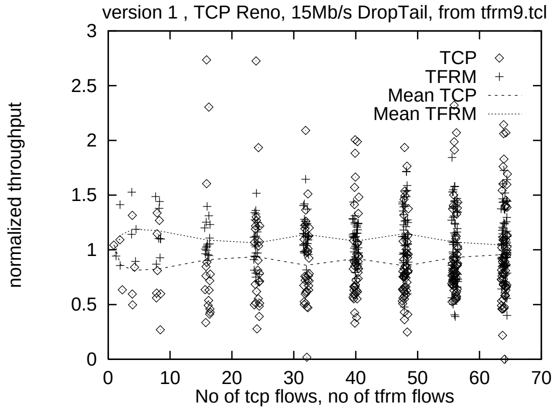


Figure 109:

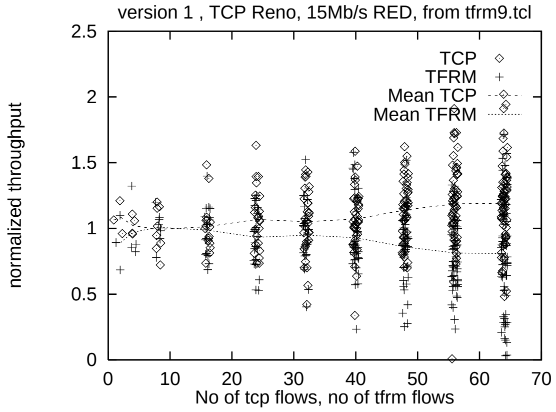


Figure 110:

4.1.4 15Mb/s Bottleneck Bandwidth with slowincr

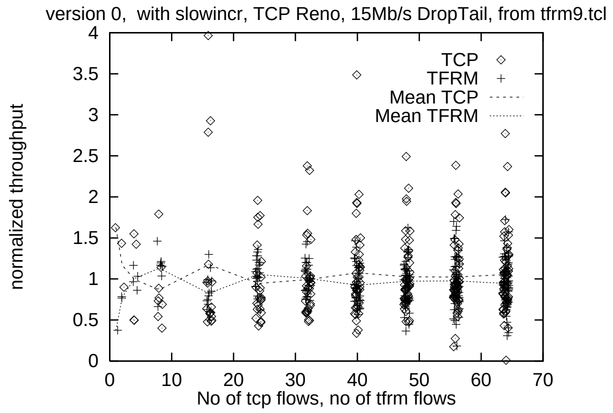


Figure 111:

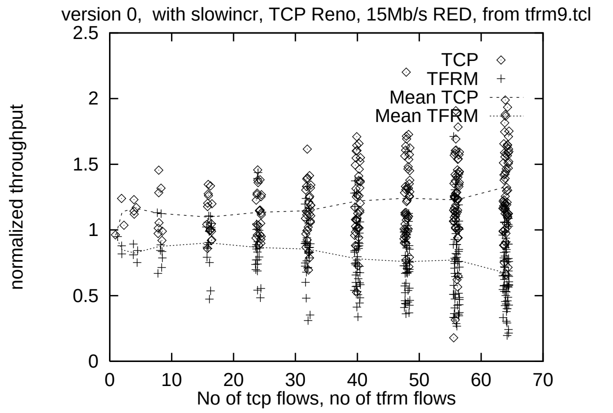


Figure 112:

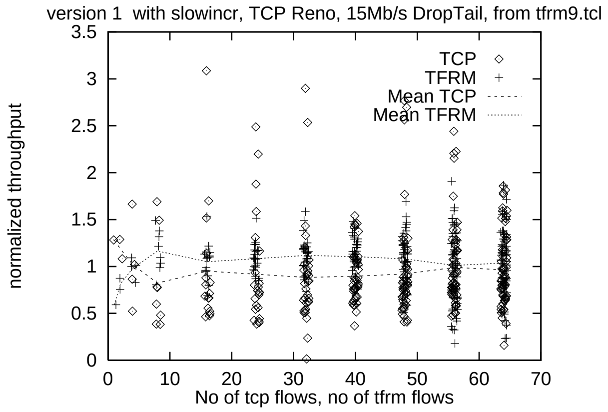


Figure 113:

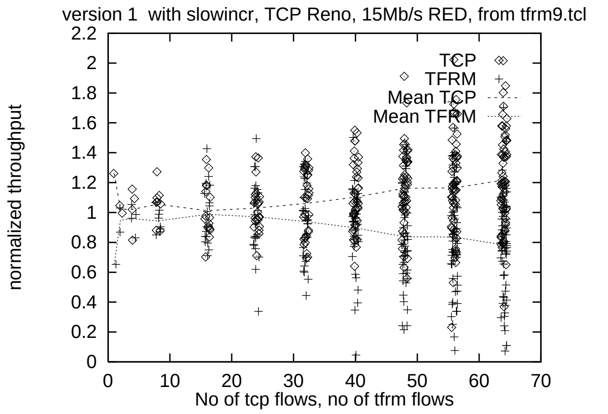


Figure 114:

4.1.5 60Mb/s Bottleneck Bandwidth without slow-incr

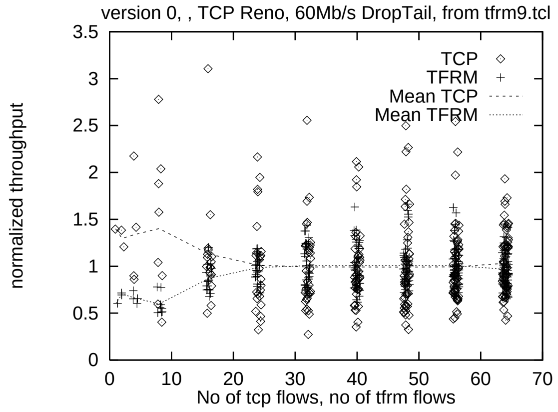


Figure 115:

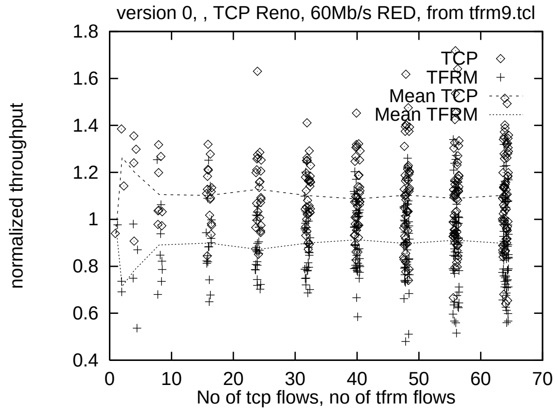


Figure 116:

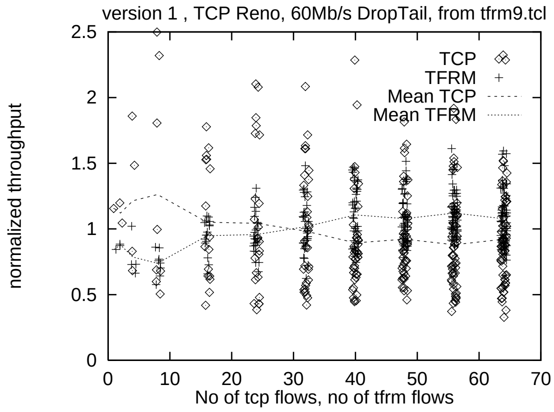


Figure 117:

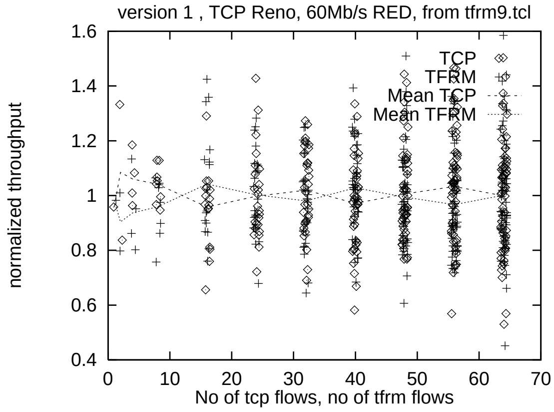


Figure 118:

4.1.6 60Mb/s Bottleneck Bandwidth with slowincr

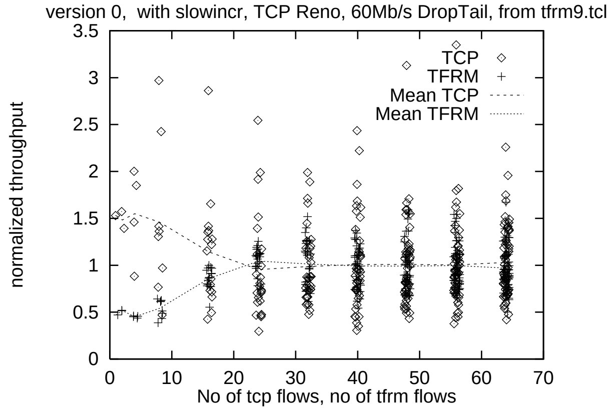


Figure 119:

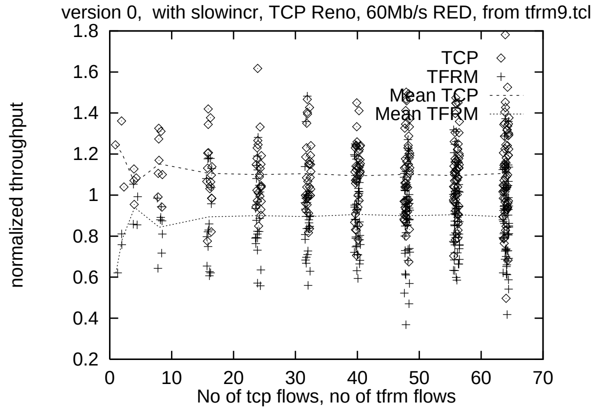


Figure 120:

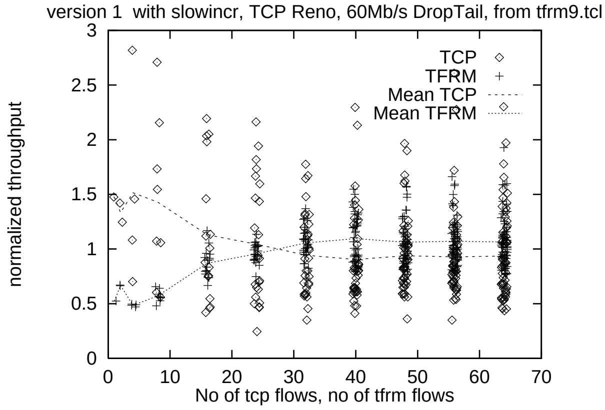


Figure 121:

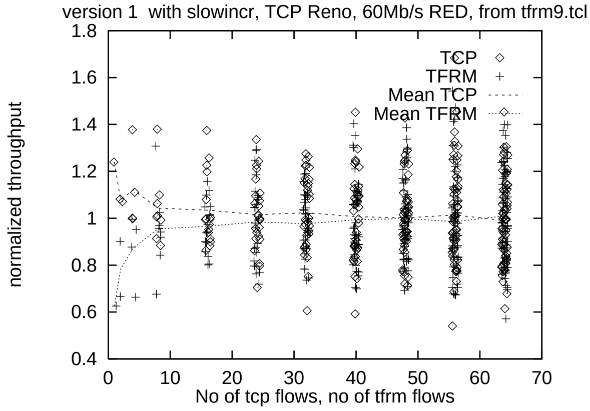


Figure 122:

4.2 Simulations using TCP Reno

This simulations do not use TCP delayed ACK and use a value of $b = 1$.

4.2.1 1.5Mb/s Bottleneck Bandwidth without slow-incr

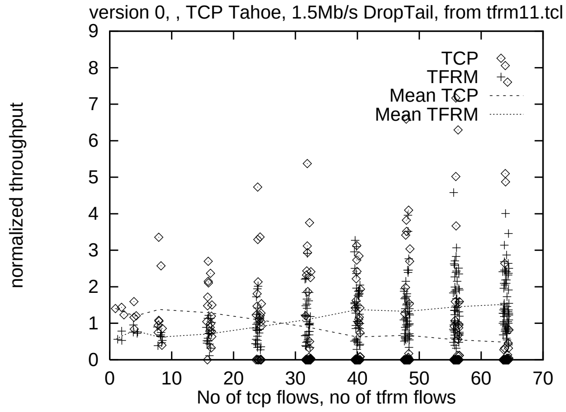


Figure 123:

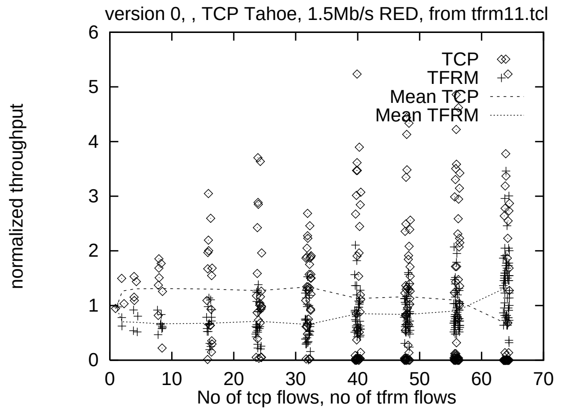


Figure 124:

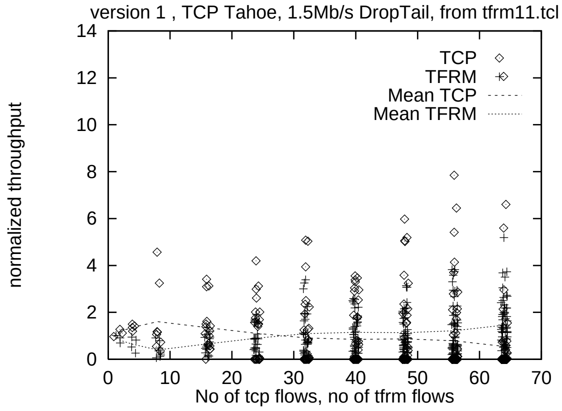


Figure 125:

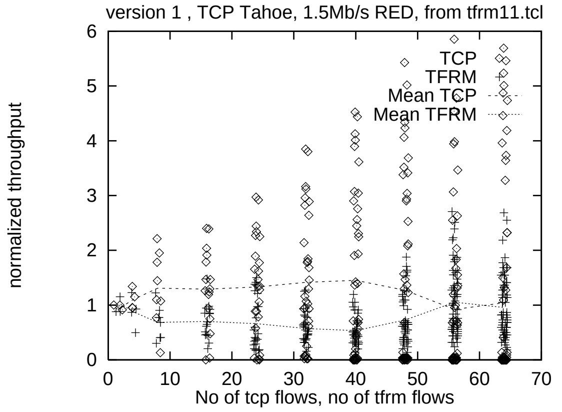


Figure 126:

4.2.2 1.5Mb/s Bottleneck Bandwidth with slowincr

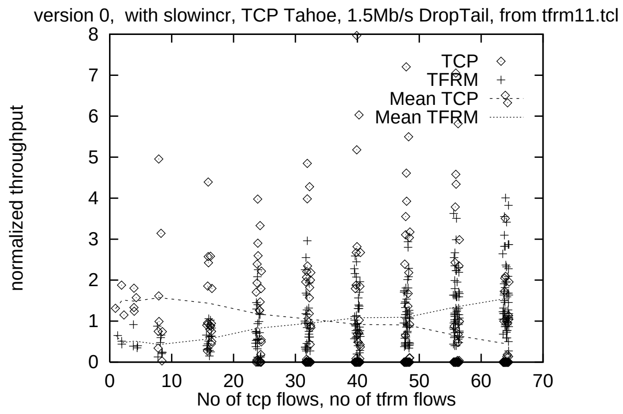


Figure 127:

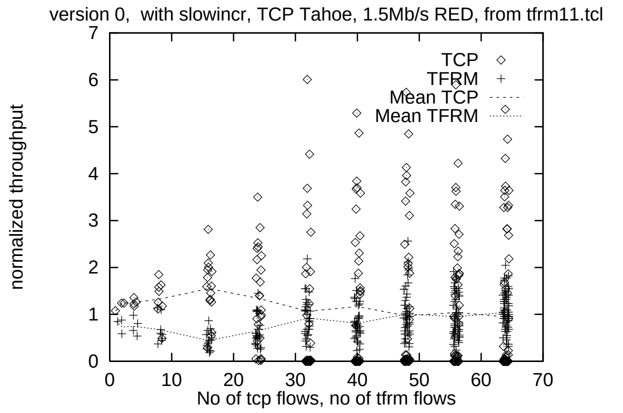


Figure 128:

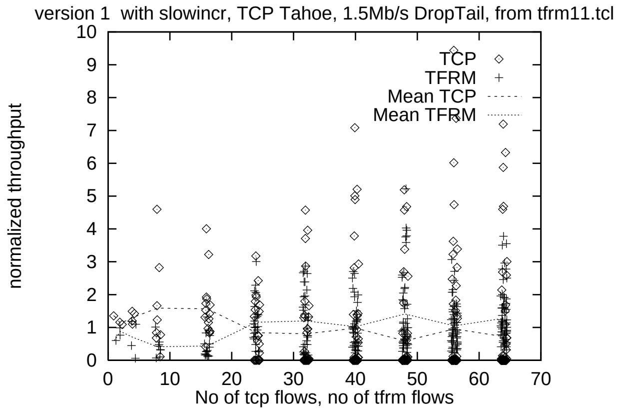


Figure 129:

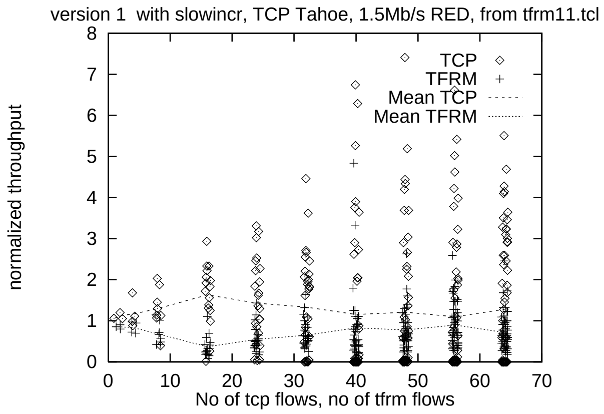


Figure 130:

4.2.3 15Mb/s Bottleneck Bandwidth without slow-incr

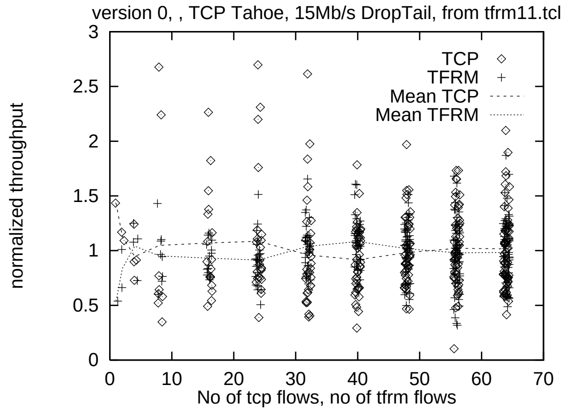


Figure 131:

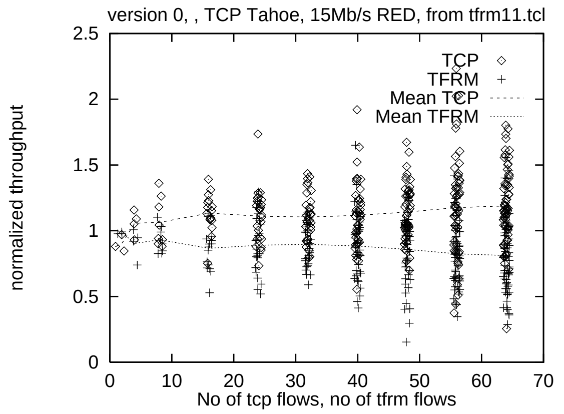


Figure 132:

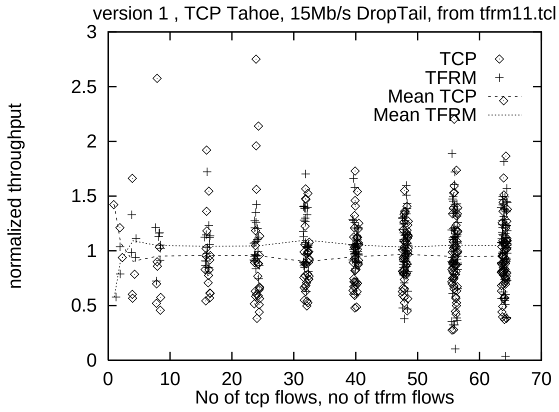


Figure 133:

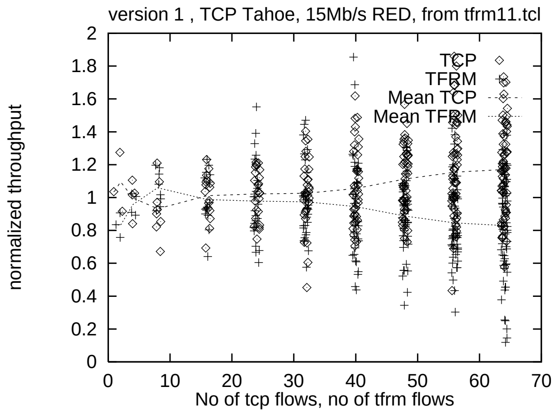


Figure 134:

4.2.4 15Mb/s Bottleneck Bandwidth with slowincr

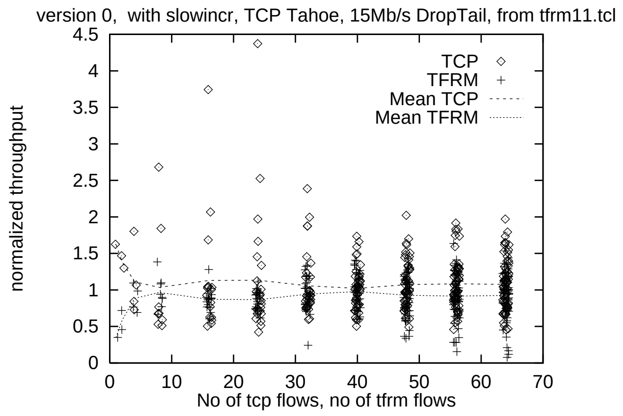


Figure 135:

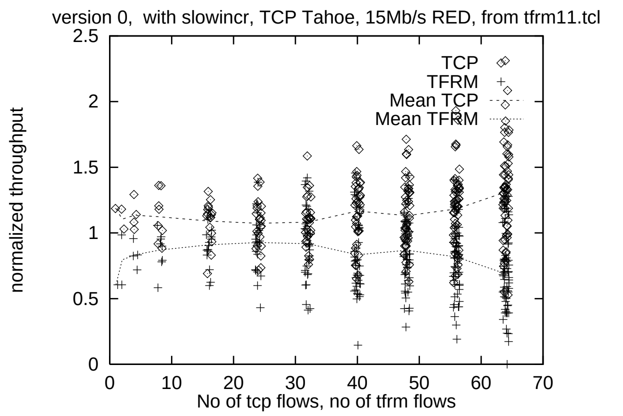


Figure 136:

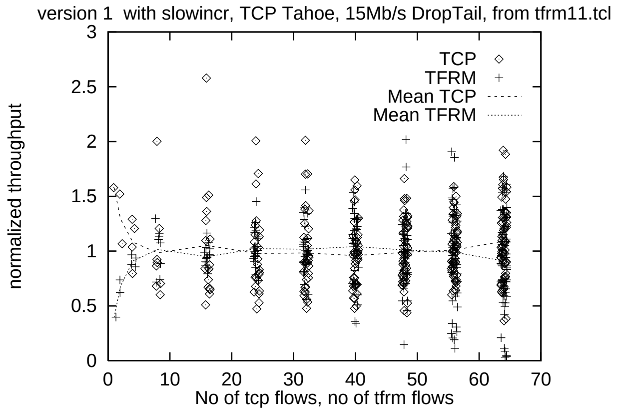


Figure 137:

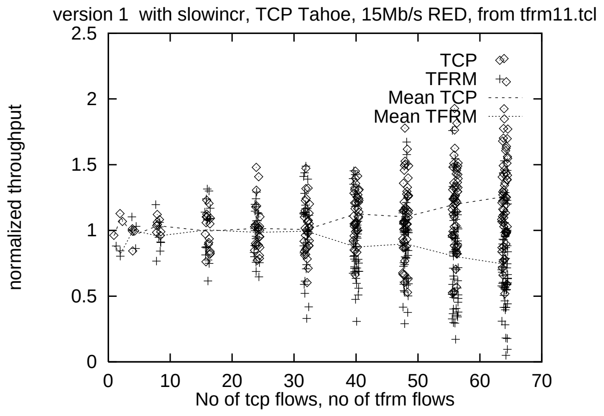


Figure 138:

4.2.5 60Mb/s Bottleneck Bandwidth without slow-incr

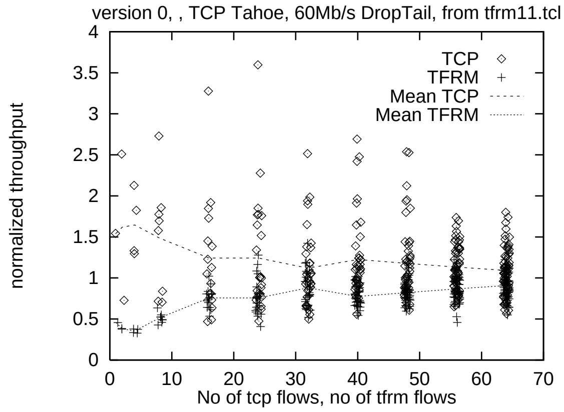


Figure 139:

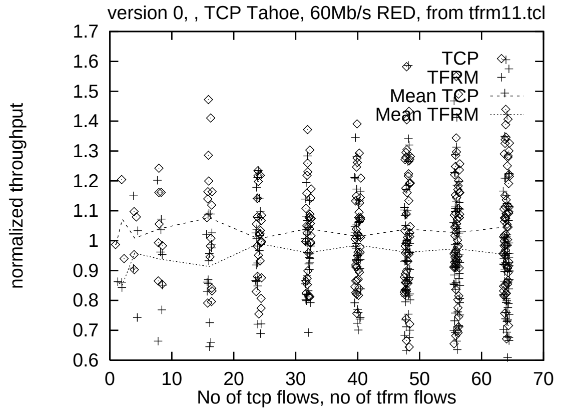


Figure 140:

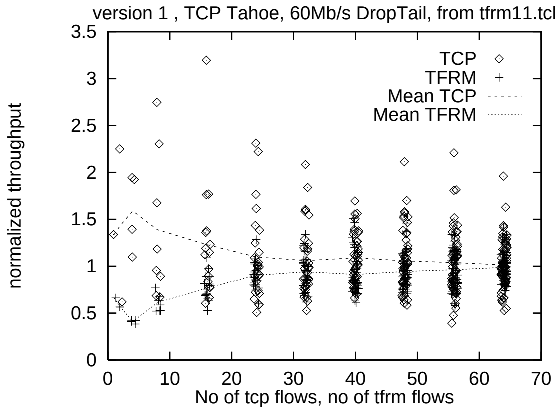


Figure 141:

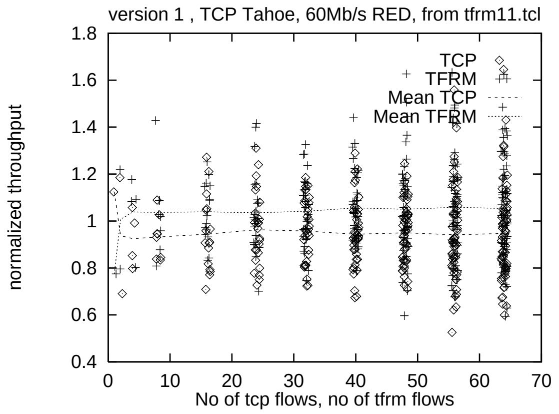


Figure 142:

4.2.6 60Mb/s Bottleneck Bandwidth with slowincr

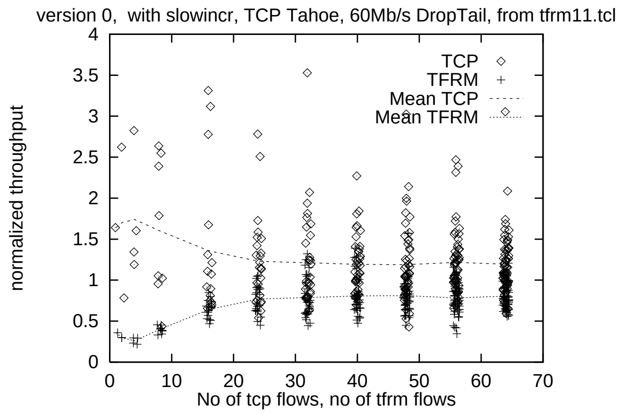


Figure 143:

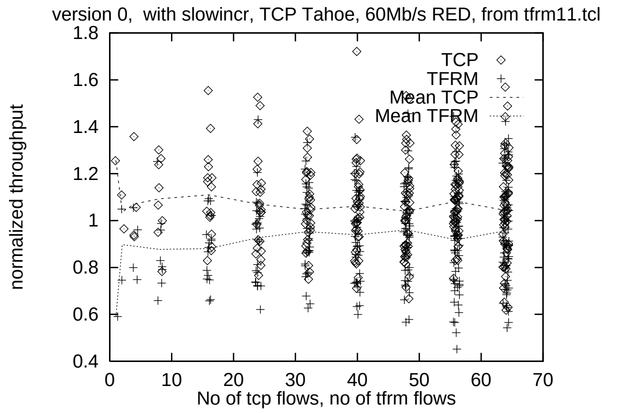


Figure 144:

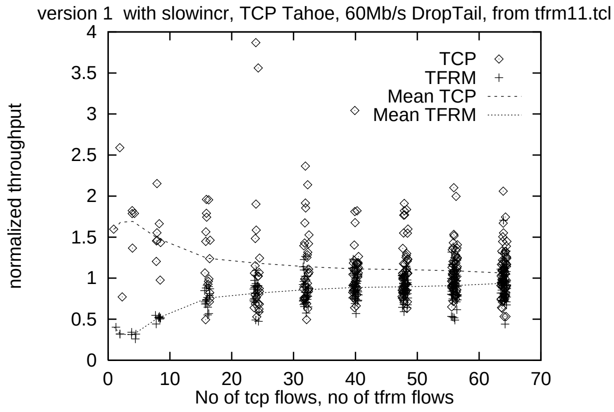


Figure 145:

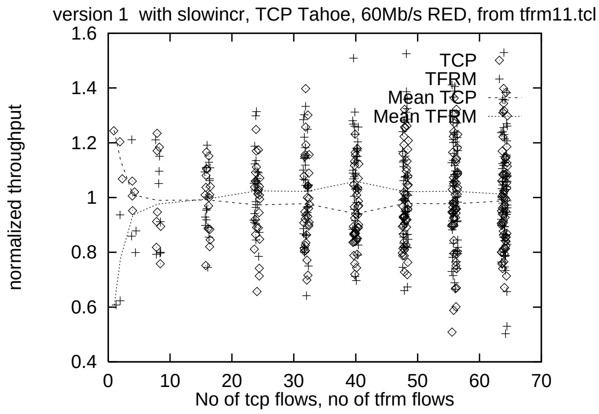


Figure 146:

4.3 Simulations using TCP Sack

This simulations do not use TCP delayed ACK and use a value of $b = 1$.

4.3.1 1.5Mb/s Bottleneck Bandwidth without slow-incr

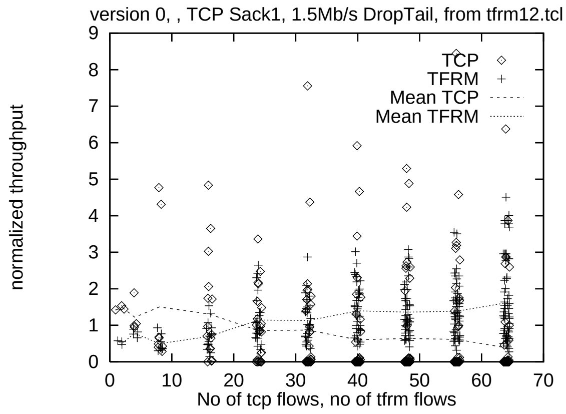


Figure 147:

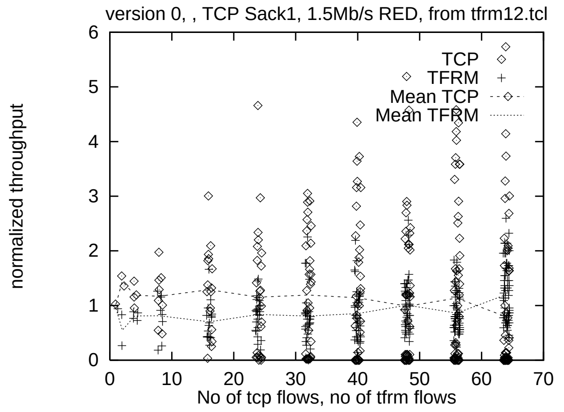


Figure 148:

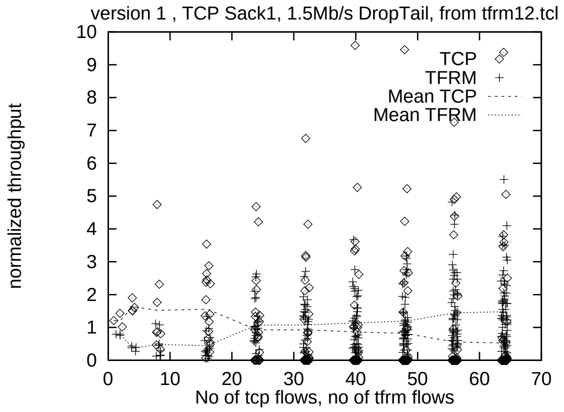


Figure 149:

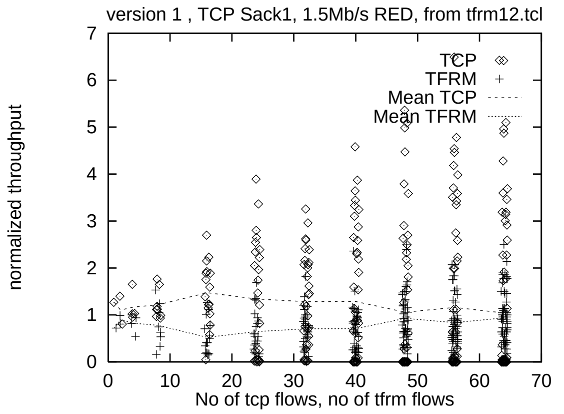


Figure 150:

4.3.2 1.5Mb/s Bottleneck Bandwidth with slowincr

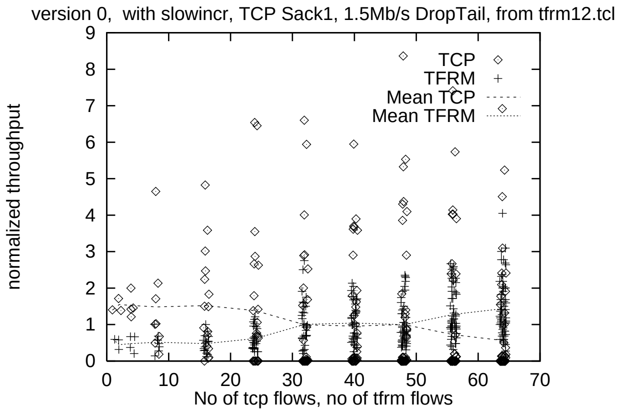


Figure 151:

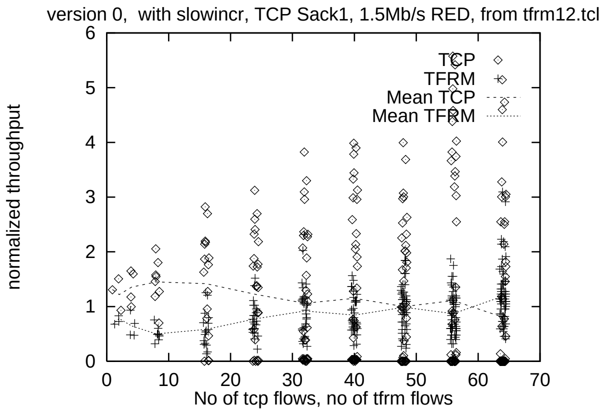


Figure 152:

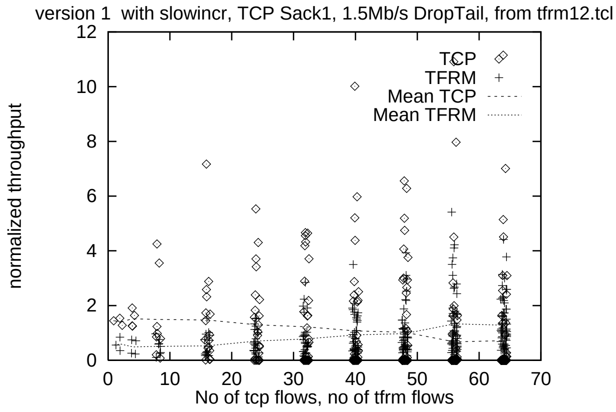


Figure 153:

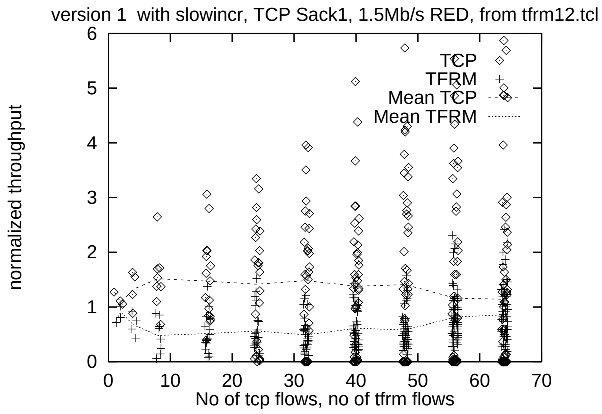


Figure 154:

4.3.3 15Mb/s Bottleneck Bandwidth without slow-incr

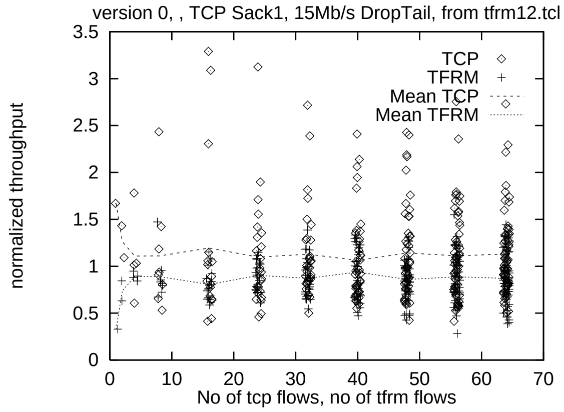


Figure 155:

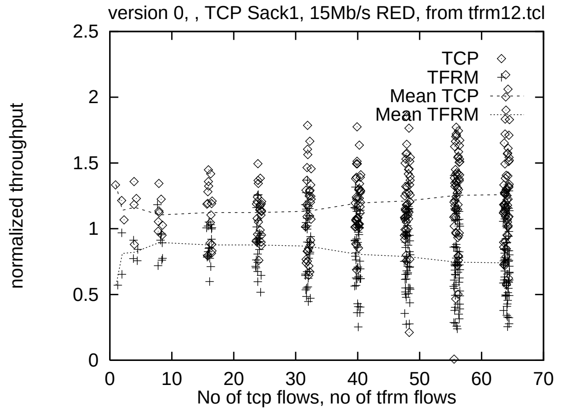


Figure 156:

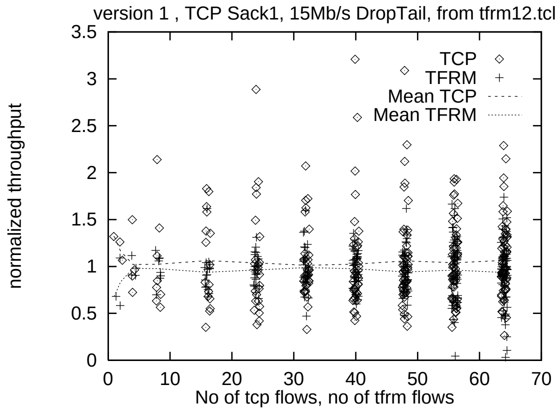


Figure 157:

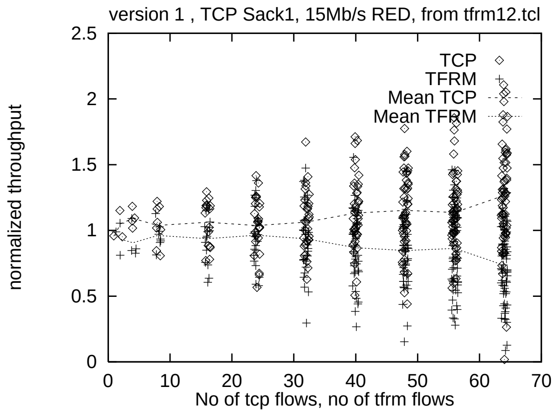


Figure 158:

4.3.4 15Mb/s Bottleneck Bandwidth with slowincr

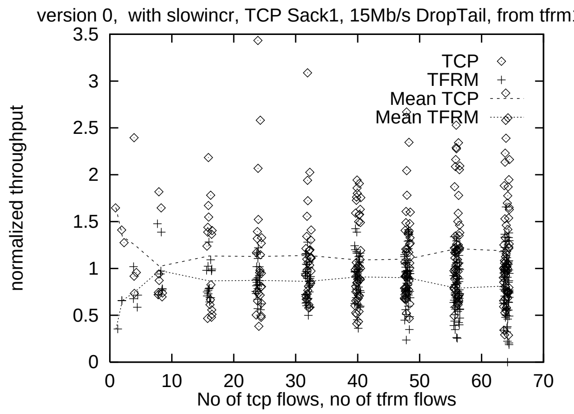


Figure 159:

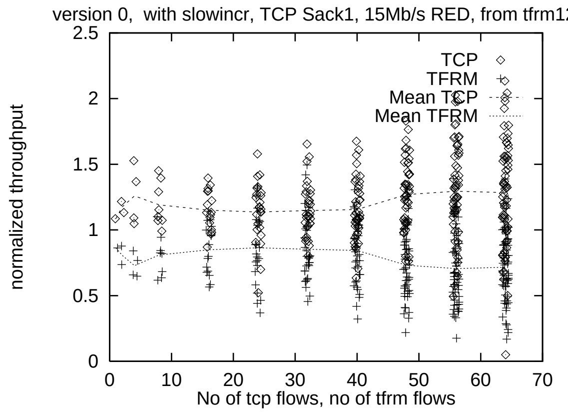


Figure 160:

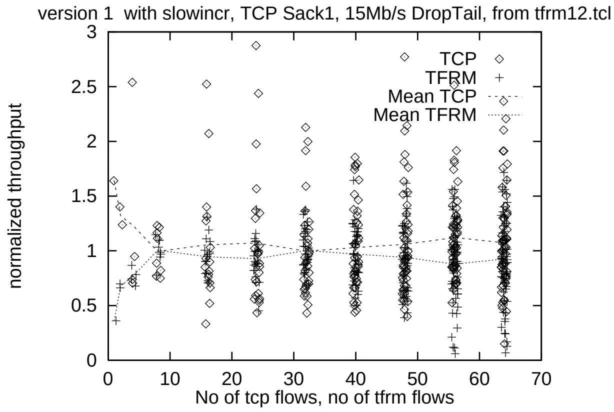


Figure 161:

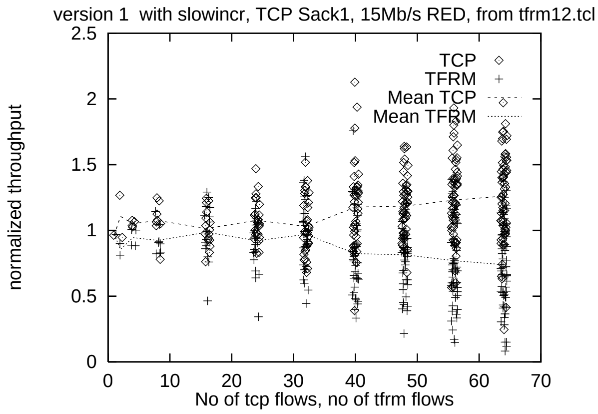


Figure 162:

4.4 Parameter Sensitivity

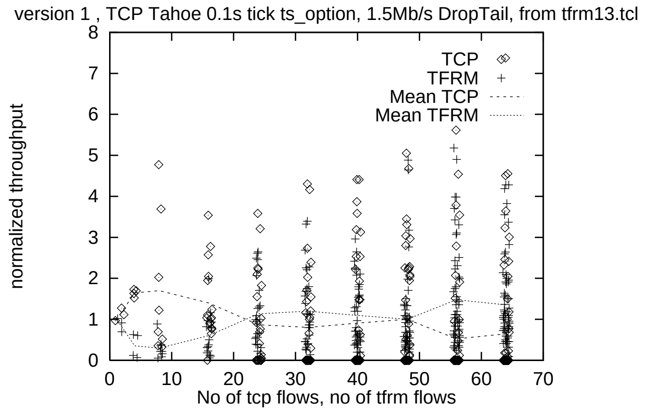
How sensitive is TFRM to t_0? We can change the algorithm we use to set t_0, and we can use versions of TCP that use different algorithms.

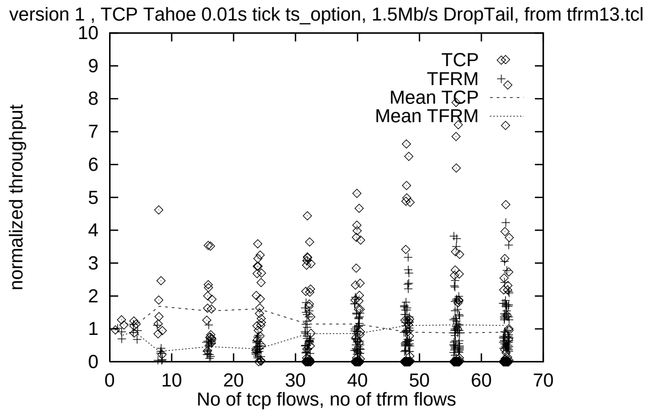
Firstly we'll leave TFRM with 100ms timer granularity, but run TCP with the timestamp option and 1ms and 10ms timer granularity.

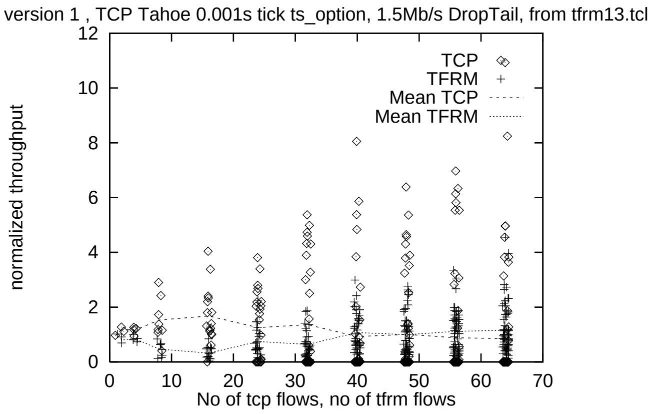
Then we'll run TFRM with 1ms and 10ms timer granularity and TCP with 100ms timer granularity.

Given that version 0 and version 1 give broadly similar results, and slowincr has minimal effect, we'll just run these experiments with version 1 without slowincr. Also 60Mb/s is not dominated by t_0, so we'll omit these runs too.

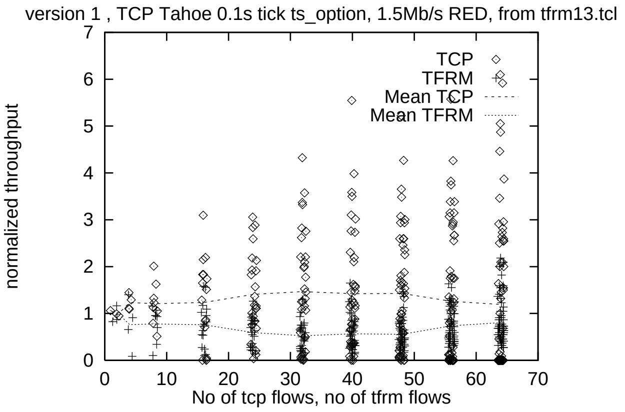
4.4.1 1.5Mb/s bottleneck, DropTail Queuing, Tahoe TCP with timestamp option

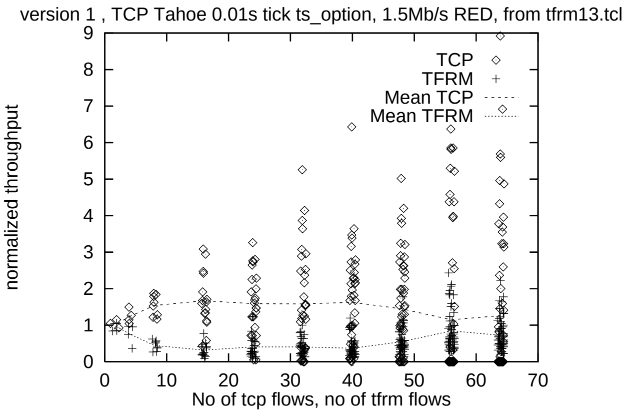


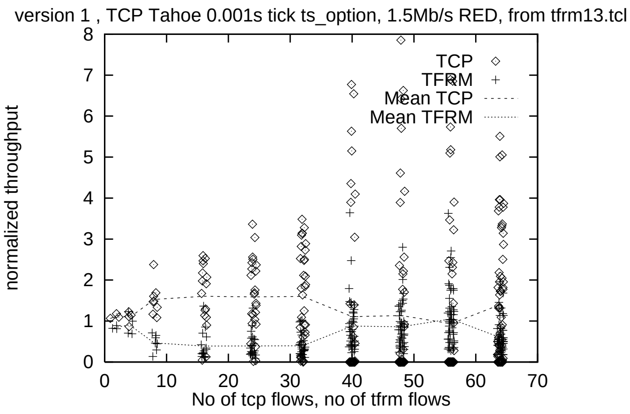




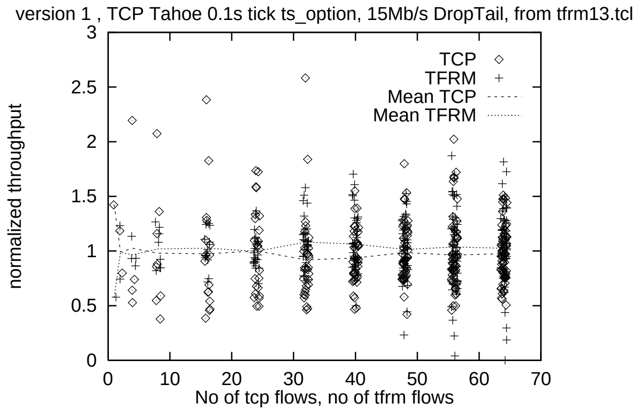
4.4.2 1.5Mb/s bottleneck, RED Queuing, Tahoe TCP with timestamp option

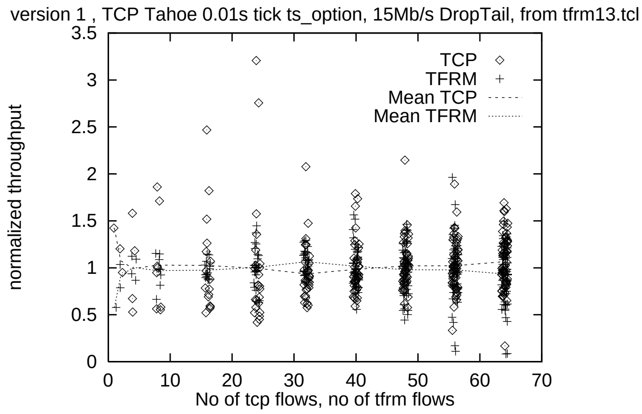


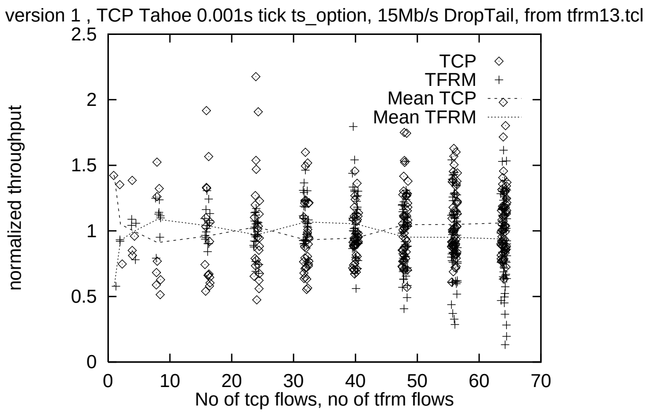




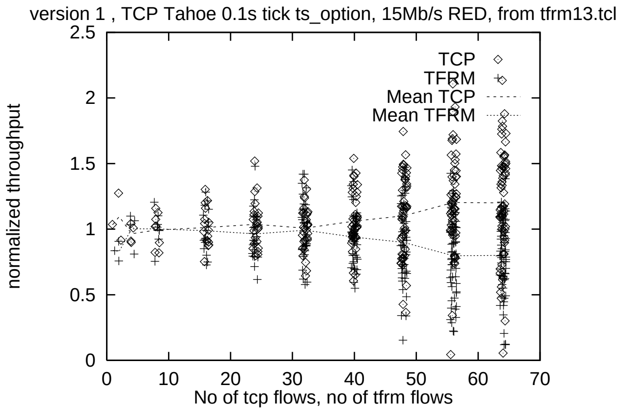
4.4.3 15Mb/s bottleneck, DropTail Queuing, Tahoe TCP with timestamp option

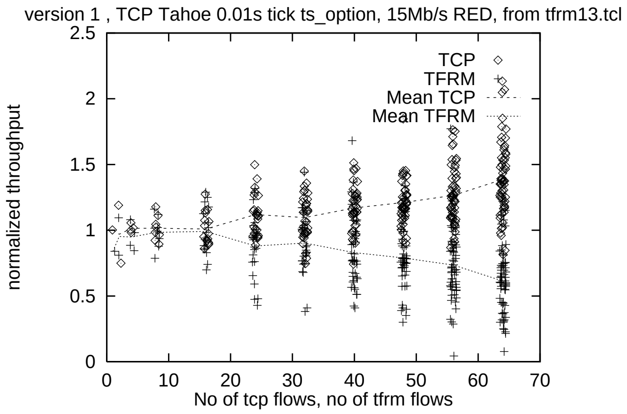


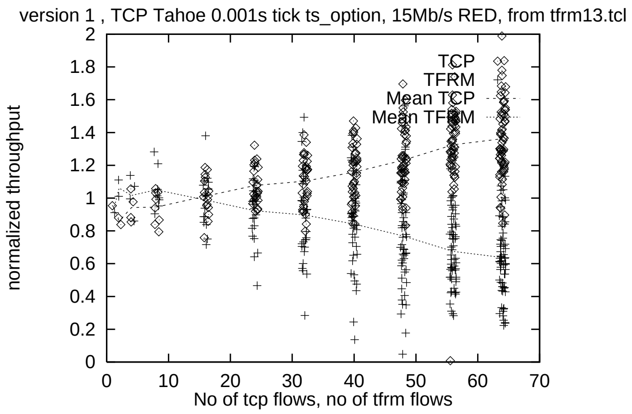




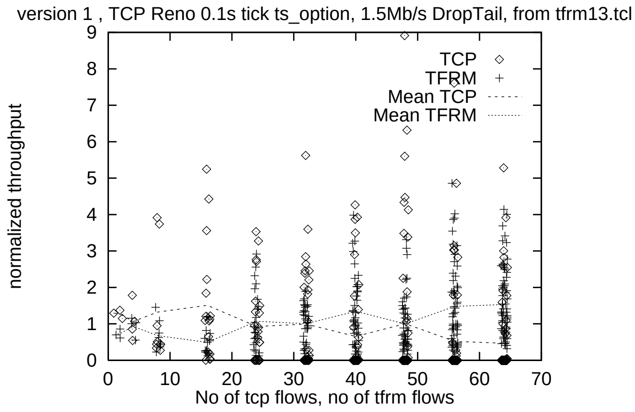
4.4.4 15Mb/s bottleneck, RED Queuing, Tahoe TCP with timestamp option

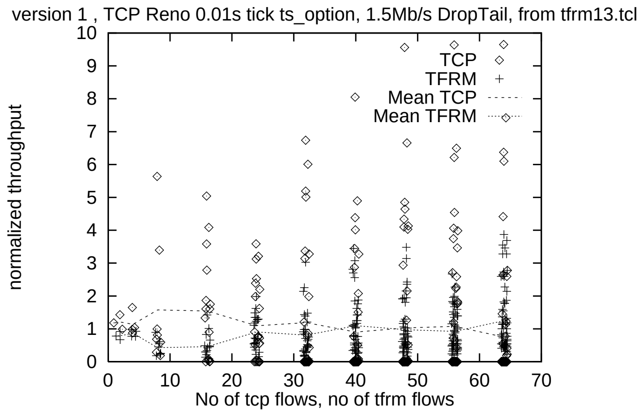


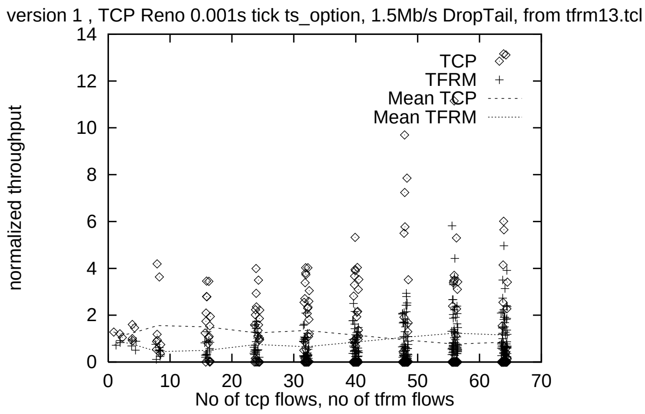




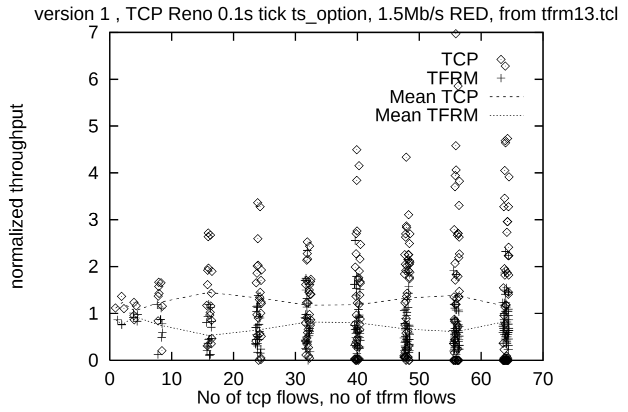
4.4.5 1.5Mb/s bottleneck, DropTail Queuing, Reno TCP with timestamp option

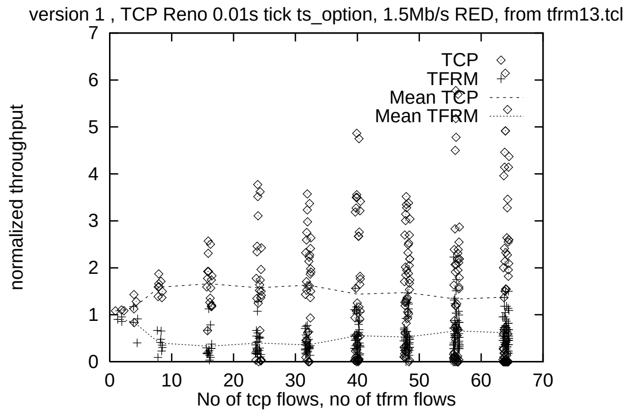


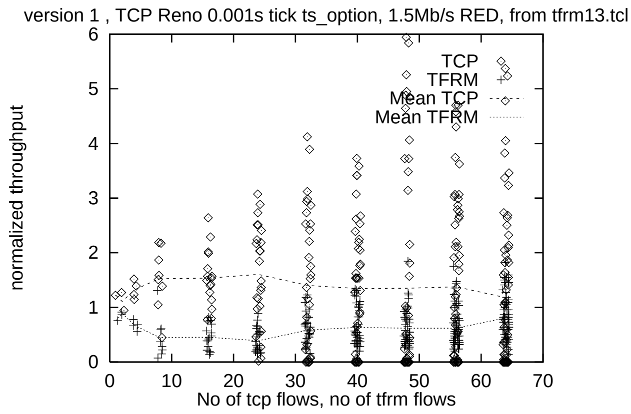




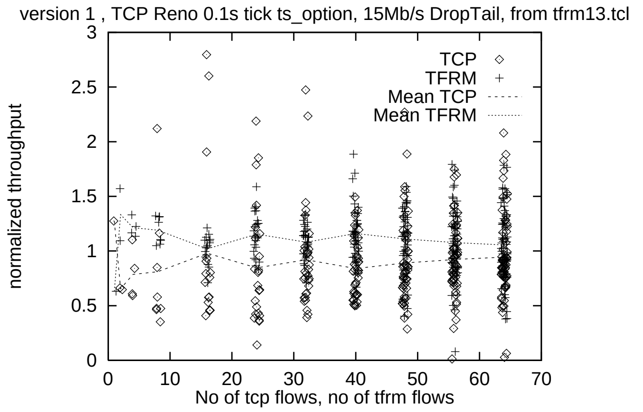
4.4.6 1.5Mb/s bottleneck, RED Queuing, Reno TCP with timestamp option

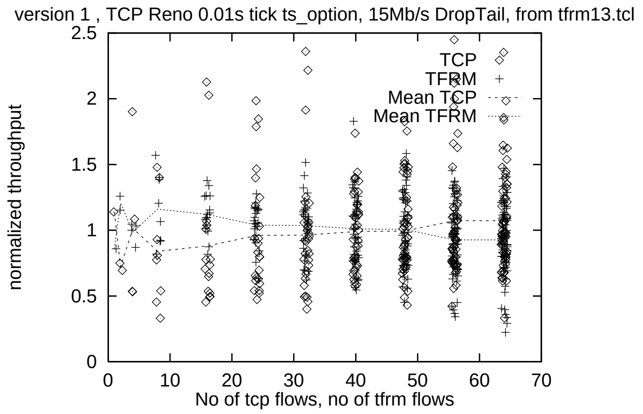


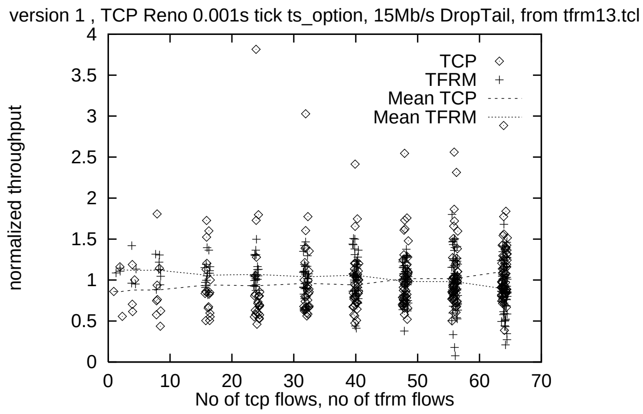




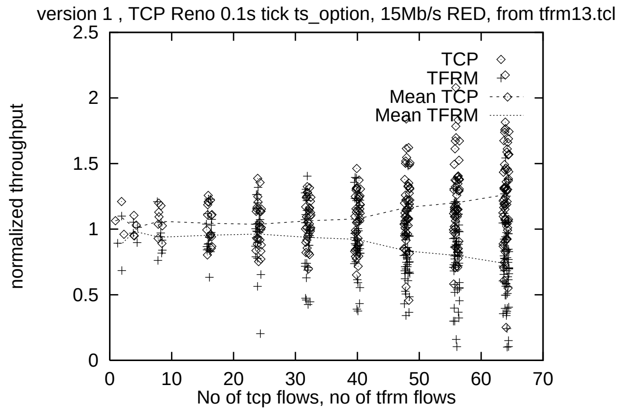
4.4.7 15Mb/s bottleneck, DropTail Queuing, Reno TCP with timestamp option

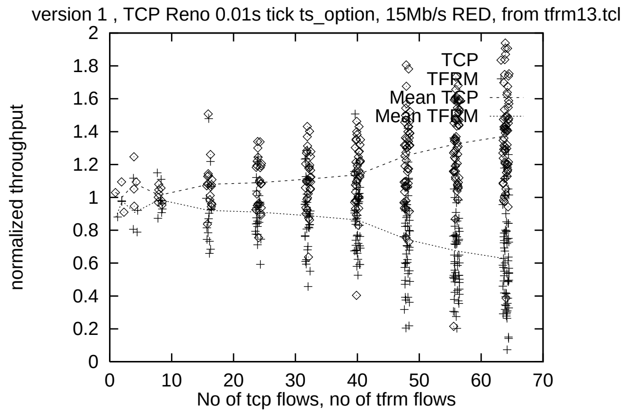


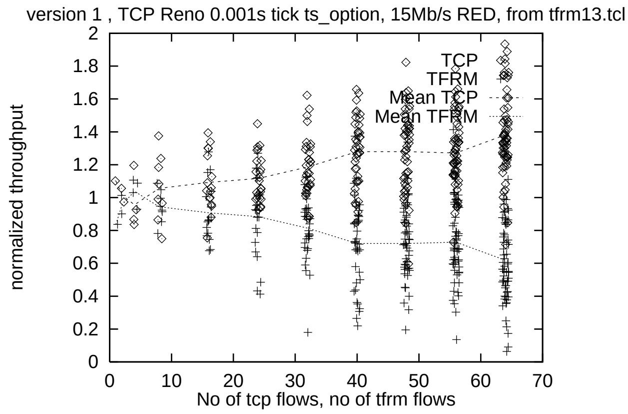




4.4.8 15Mb/s bottleneck, RED Queuing, Reno TCP with timestamp option







4.5 Examining TFRM using the Padhye equation competing against itself

What we're concerned about here, as with the Mathis/Mahdavi/Floyd equation is stable behaviour. We'd rather see that no oscillation occurs and that we get 100% bottleneck bandwidth utilization.

4.5.1 1.5Mb/s bottleneck bandwidth, without slow-incr

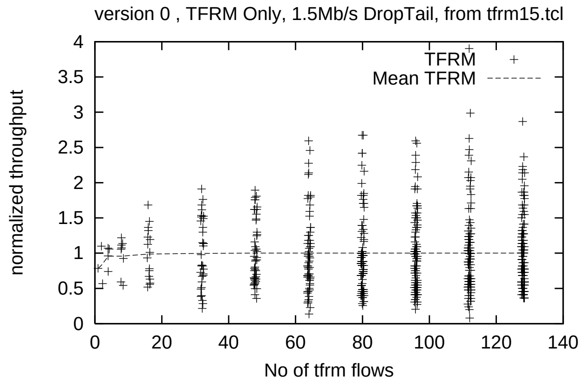


Figure 163:

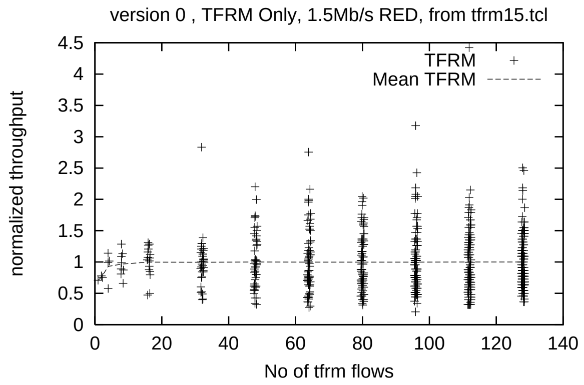


Figure 164:

Note: version 0 fails to achieve 100% utilisation with small numbers of flows.

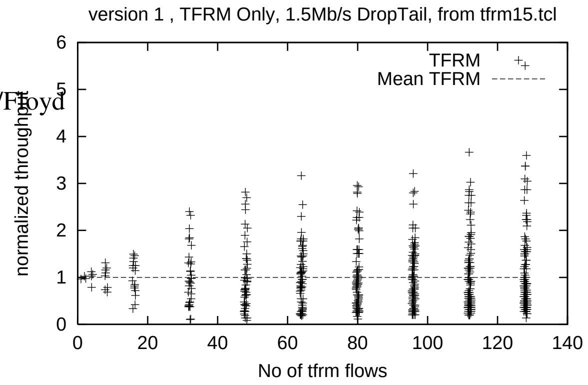


Figure 165:

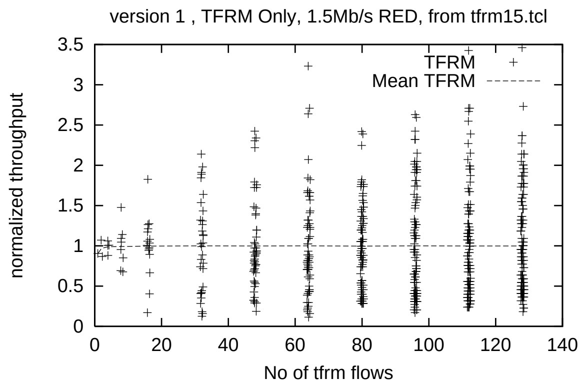


Figure 166:

To discover why version 0 fails to achieve 100% utilisation with small numbers of flows, we graph bandwidth against time from the simulations with one and two flows for both version 0 and version 1. Bandwidth is a windowed average over a fixed number of packets. Slowincr is not used.

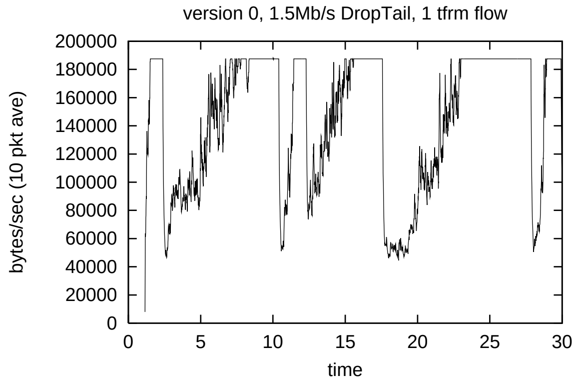


Figure 167: Version 0

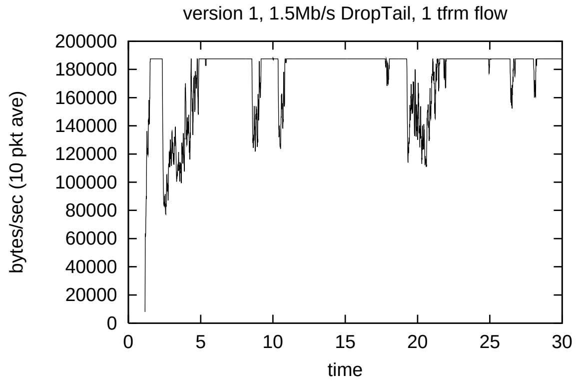


Figure 168: Version 1

Version 0 is exhibiting relatively unstable behaviour, although it is not actually oscillatory in these simulations. Version 1's behaviour is much more stable.

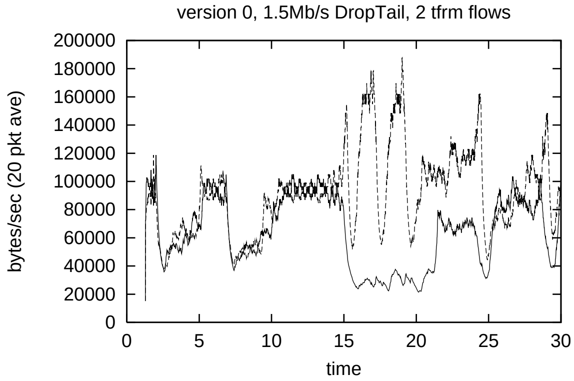


Figure 169: Version 0, two flows

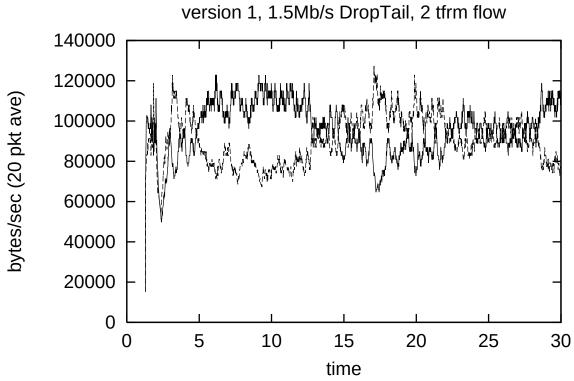


Figure 170: Version 1, two flows

4.5.2 1.5Mb/s bottleneck bandwidth, with slowincr

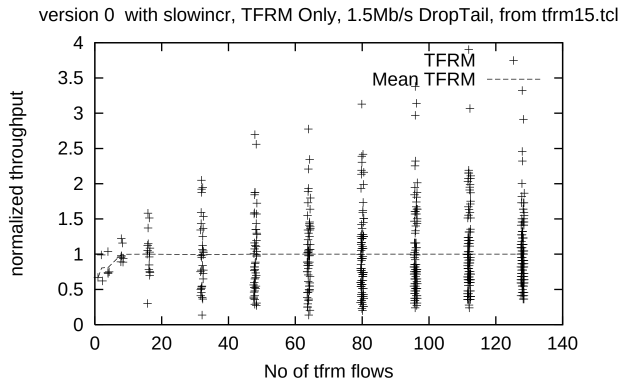


Figure 171:

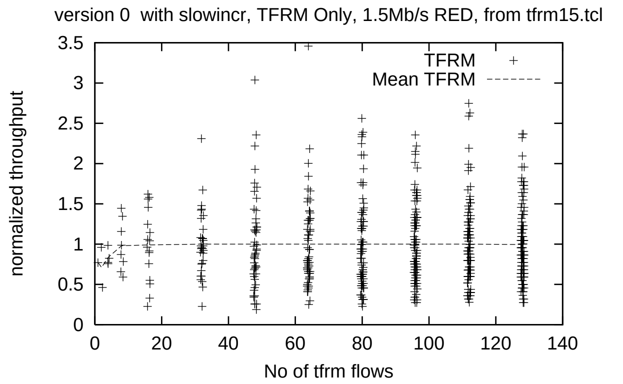


Figure 172:

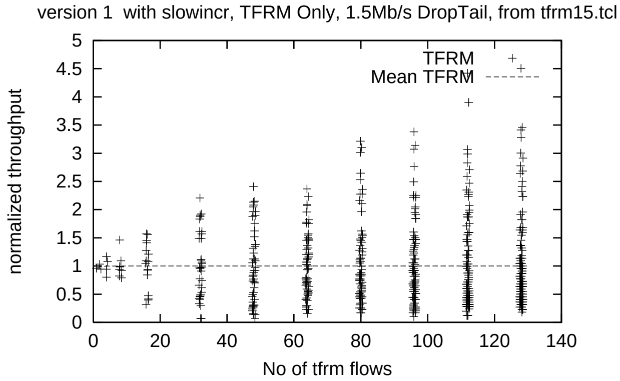


Figure 173:

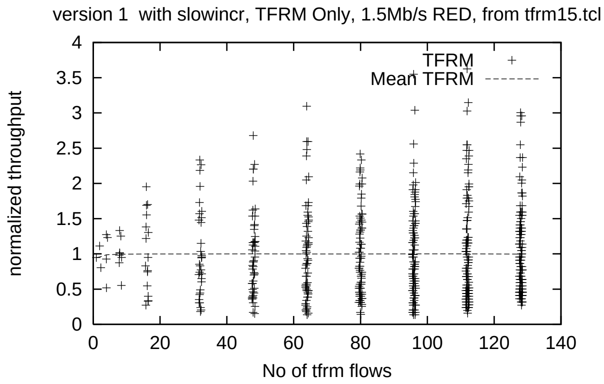


Figure 174:

4.5.3 15Mb/s bottleneck bandwidth, without slow-incr

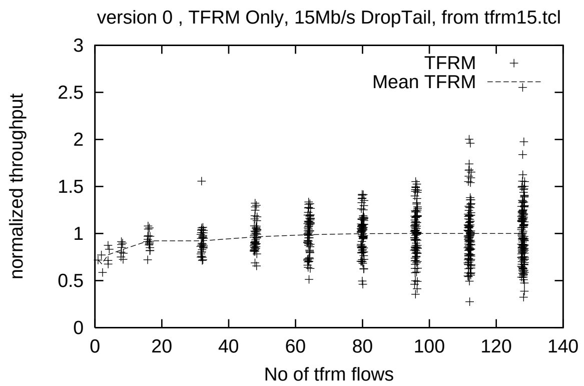


Figure 175:

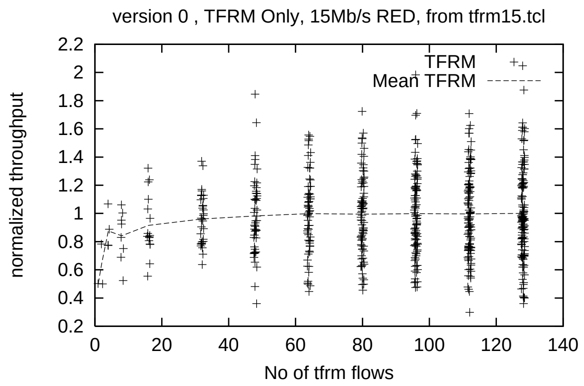


Figure 176:

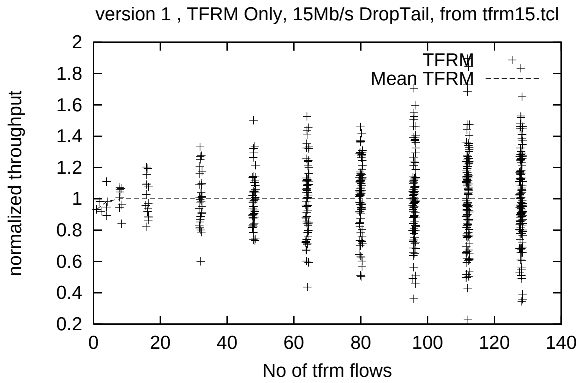


Figure 177:

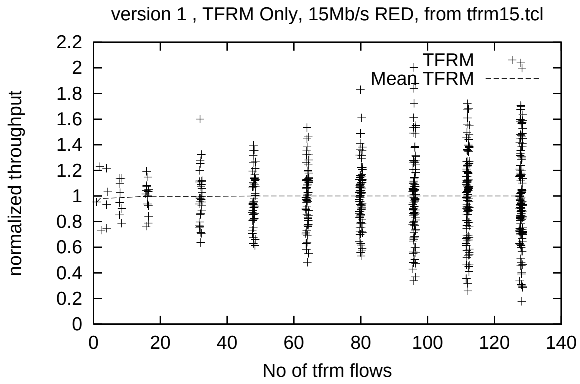


Figure 178:

In figures 175 and 176, TFRM fails to achieve 100% utilization with less that 64 flows. To understand why this is the case we examine bandwidth/time plots from these simulations for the 8 flow case:

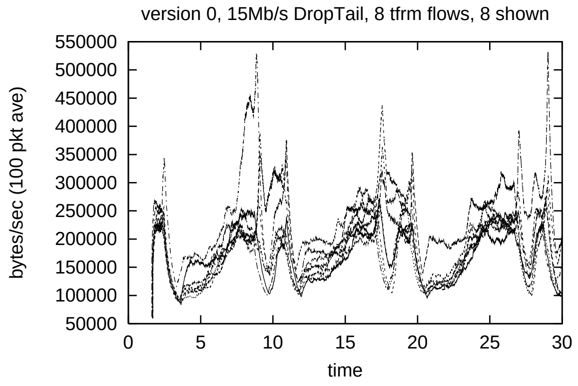


Figure 179:

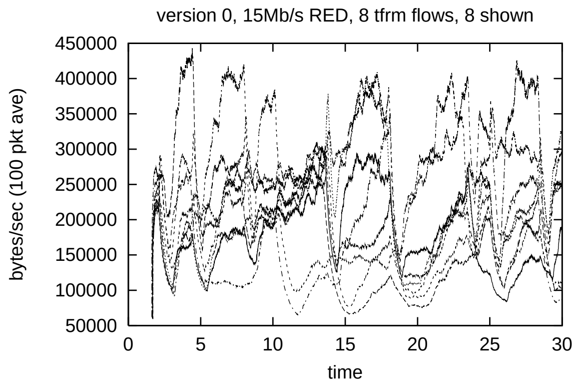


Figure 180:

The cause of this appears to be oscillatory behaviour with a period of between 4 and 7 seconds. RED serves to de-phase these flows somewhat but does not prevent the oscillatory behaviour.

In contrast, figures 177 and 178 indicate version 1 does manage 100% utilization. Examining the bandwidth/time plots from these simulations does not show similar oscillations:

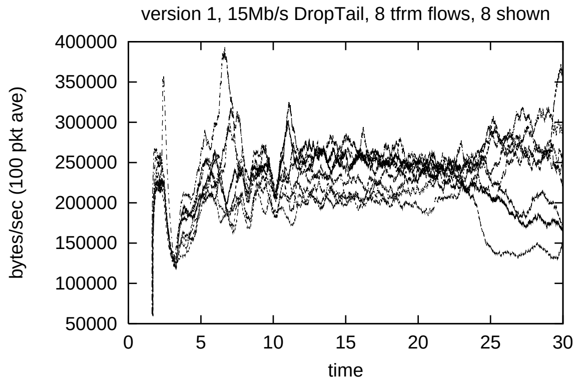


Figure 181:

However, it appears as if the behaviour is becoming unstable after t=30 seconds, so we extended this plot out to 60 seconds:

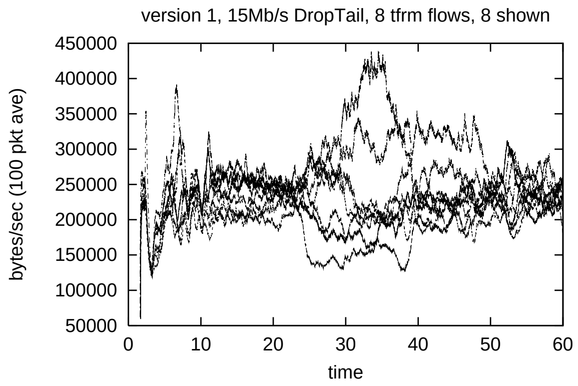


Figure 182:

The apparent instability eventually damps itself out. We believe this to be an example of a phase effect similar to that found with TCP in simulation environments. We had attempted to prevent phase effects by randomly delaying packets by +/- 30% of the inter-packet gap, but it appears this is insufficient to truely de-phase flows with DropTail queuing.

In contrast, with RED queuing and version 1, there is a much wider spread of bandwidths seen at any one time, but still no oscillation:

.

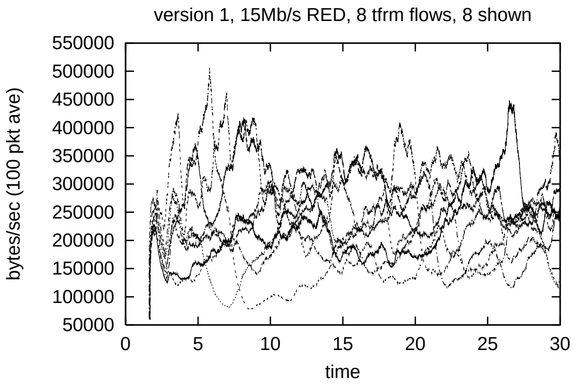


Figure 183:

4.5.4 15Mb/s bottleneck bandwidth, with slowincr

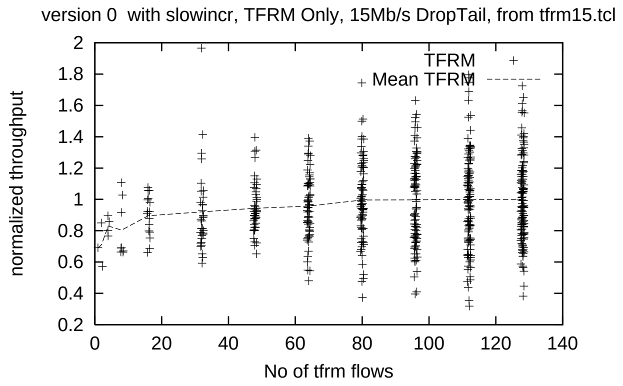


Figure 184:

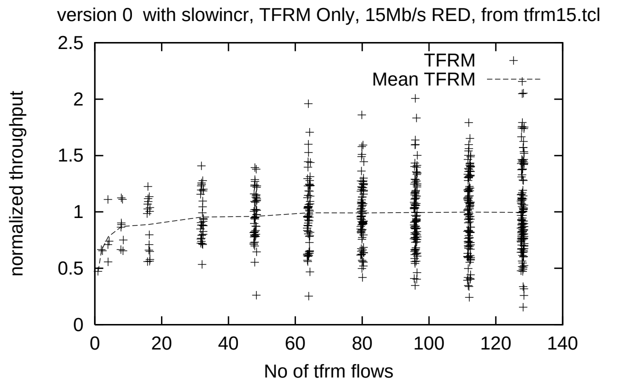


Figure 185:

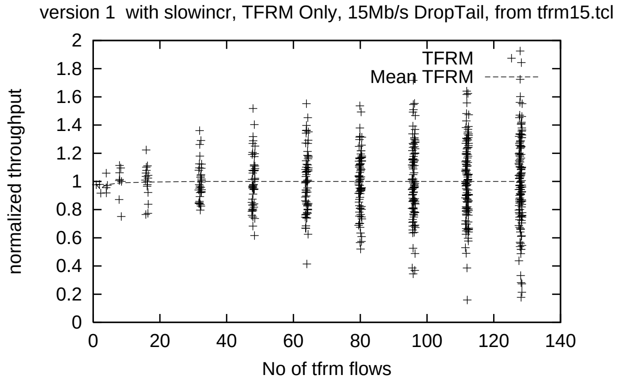


Figure 186:

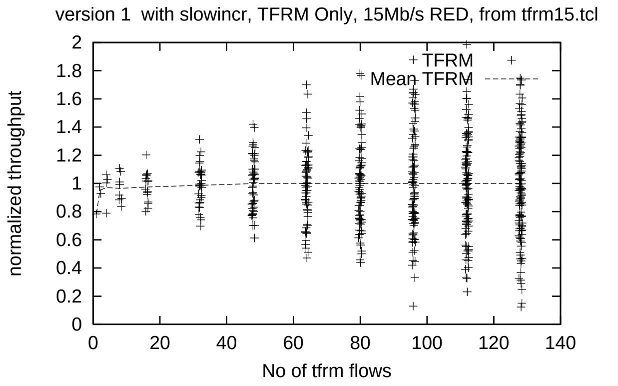


Figure 187:

4.5.5 60Mb/s bottleneck bandwidth, without slow-incr

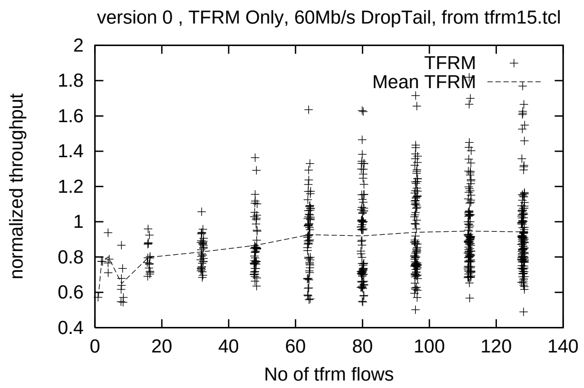


Figure 188:

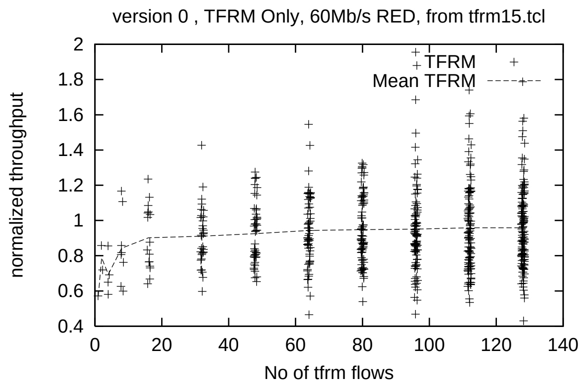


Figure 189:

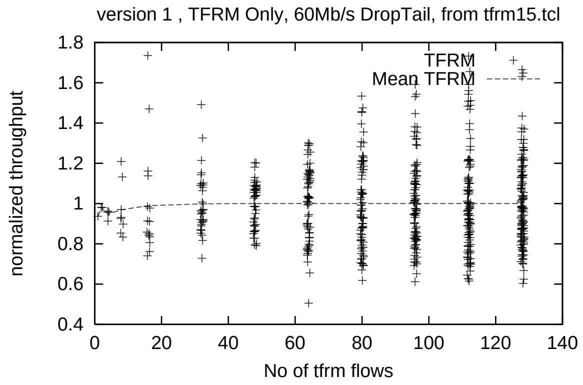


Figure 190:

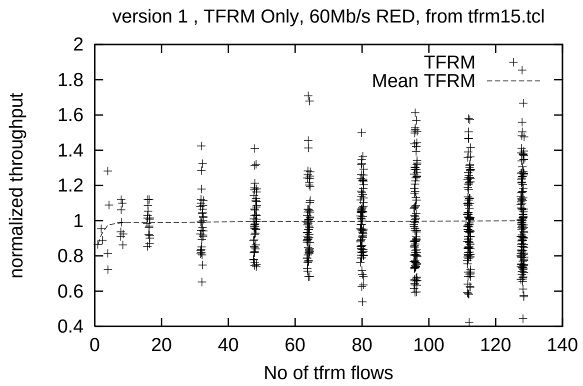


Figure 191:

4.5.6 60Mb/s bottleneck bandwidth, with slowincr

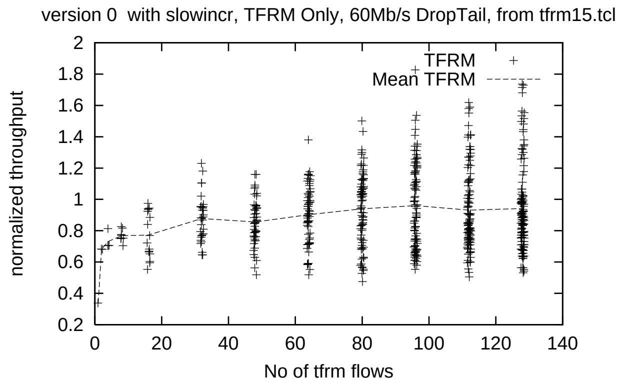


Figure 192:

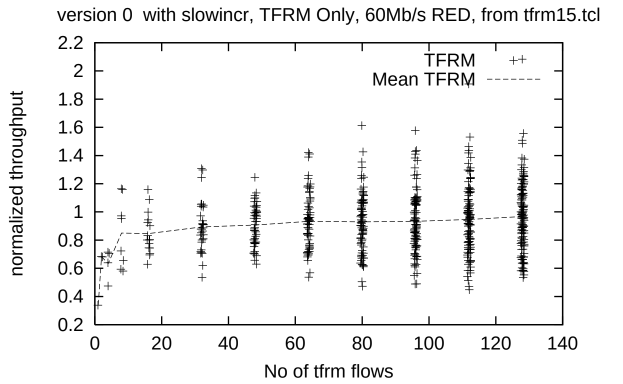


Figure 193:

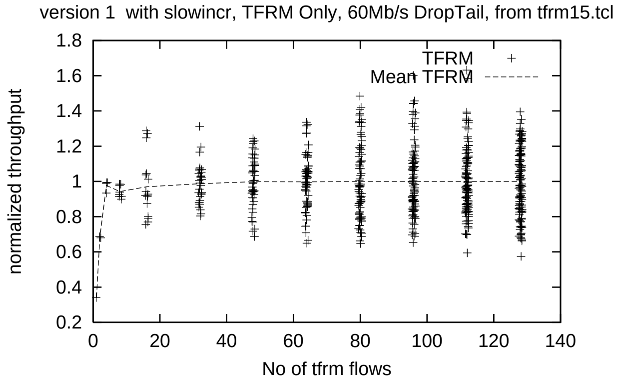


Figure 194:

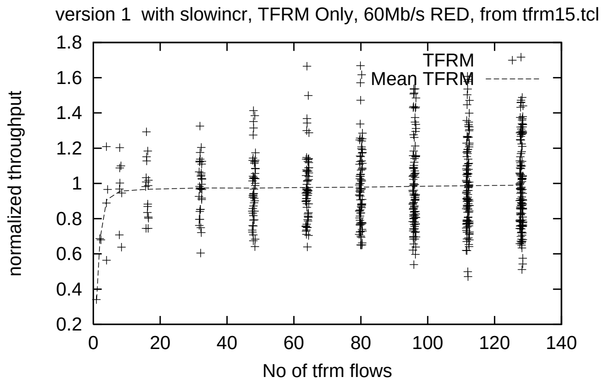


Figure 195:

References

[1] J. Padhye, V. Firoiu, D. Towsley, J. Kurose, "Modeling TCP Throughput: A Simple Model and its Empirical Validation", Proc. ACM Sigcomm, 1998, Vancouver, Canada.