

Passive TCP Stream Estimation of RTT and Jitter Parameters

Jason But, Urs Keller*, David Kennedy and Grenville Armitage
Centre for Advanced Internet Architectures
Swinburne University of Technology
Melbourne, Australia
{jbut, dkennedy, garmitage}@swin.edu.au ukeller@gmail.com

Abstract

There exist many tools to passively monitor a link for traffic flows. They are typically used near the edge of the network, but not necessarily at the termination point of data flows – usually within a few hops of end-points. Round Trip Time (RTT) values for individual flows is of interest for network management purposes and can be used to indicate user experienced network delay, and in network design decisions. Determining the RTT when not at an end-point of a data flow is complicated by the fact that packets may be seen out of order and that witnessed packets may not reach their destination. In this paper we present an algorithm to estimate running RTT and Jitter characteristics of TCP streams monitored at the midpoint of a TCP flow.

1. Introduction

Estimating network RTT values between two individual hosts on the network is relatively easy when one is located on one of the two end-points of the data flow. The problem becomes more difficult when we consider the scenario of a passive listener somewhere in the middle of a dataflow between two hosts [1, 2].

If monitoring for network management purposes, it is more useful to listen at a common point rather than at each host. The aggregate traffic of multiple users produce better statistical values for network properties [1, 2]. Typically traffic is monitored near the gateway of a network under investigation, a small number of hops from the end-points of the data flows.

Estimation of RTT and Jitter values of individual data flows can assist in network management and the design and placement of Internet caches to reduce the RTT and wait time experienced by end users.

Passive estimation of RTT while monitoring a data stream from a mid-point is non trivial [1, 2]. In this paper we present an algorithm that can be applied to

TCP packets collected at a mid-point of a data flow to generate a sequence of RTT and Jitter estimates that accurately track the RTT estimate in the end-host TCP stack. The output is superior to existing tools such as **tstat** which produce a single RTT estimate [15, 16].

2. RTT Estimation at a Mid-Point

RTT estimation of individual TCP flows is regularly performed within each end-host TCP stack [3, 4]. The algorithm does not measure network RTT but rather TCP RTT, or the RTT that the application using TCP sees. Where delayed acknowledgements are used, the timeout before the acknowledgement is sent is included in the RTT estimate. The sender cannot detect the use of delayed acknowledgements to compensate when estimating RTT.

When monitoring a TCP Stream at a mid-point (see Figure 1), we do not have access to all data within the end-host TCP stacks. Calculations must be based on packets captured at the measurement point [2, 5].

The general approach reconciles data packets with acknowledgements traversing the measurement point in opposing directions. This is used to infer the state at the end hosts of the TCP flow. Passive estimation of RTT involves the consideration of a number of issues.

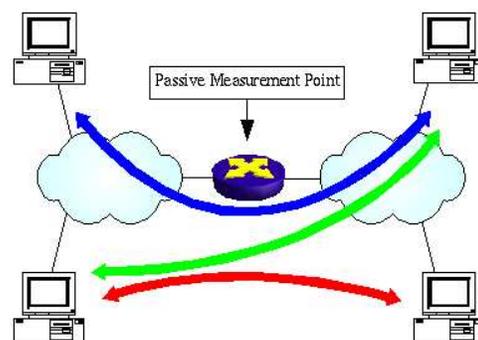


Figure 1. Passive Monitoring of Data Streams at the Midpoint of the Flow

* Urs Keller was a visitor at CAIA from the Swiss Federal Institute in Lausanne EPFL.

2.1. Data Packets Lost Prior to Capture

Data packets can be dropped or lost by the network prior to reaching the measurement point. This results in the retransmission of this packet and also causes the TCP window size to decrease. Lost packets must be detected to properly estimate RTT and Jitter [1, 6, 7].

Problems occur if an entire window of data is lost prior to reaching the measurement point, non-arrival of any packets makes this event impossible to detect.

2.2. Data Packets Lost After Capture

Data packets may be dropped or lost after passing the measurement point but prior to reaching the destination. In this instance, an acknowledgement will not be sent and retransmission will occur [2, 7].

Apart from affecting the estimate of the sender window size, it is also imperative that captured data packets that are later found to have been dropped are not used in estimating stream RTT and Jitter values.

2.3. Acknowledgements Lost Prior to Capture

A data packet may reach the destination, however the subsequent acknowledgement may be dropped or lost prior to reaching the measurement point. This has the same effect as if the data packet was lost, resulting in packet retransmission. This event should be treated as a data packet lost after the measurement point [7].

2.4. Acknowledgements Lost After Capture

Acknowledgements may be dropped or lost by the network after passing the measurement point – we know that a piece of data has reached the destination but the source host may not learn this fact [2, 7].

In this case a data packet that has reached the destination may yet be retransmitted. This fact must be taken into account by any RTT estimation algorithm.

2.5. Timeout Retransmissions

Even if all packets successfully traverse the network, this may take longer than the current sender RTT estimate, resulting in packet retransmission. When the acknowledgements arrive, the sender will stop retransmission and adjust its RTT estimate, maximising the *(dropped_packet):(long network delay)* ratio [1, 4-6, 8, 9]. RTT estimation must consider possible packet retransmission in the case of no loss.

3. Existing Approaches

Other algorithms that estimate the RTT parameters of TCP streams monitored at the midpoint include:

3.1. SYN-ACK Estimation

Jiang and Dovrolis [1] propose the SYN-ACK technique. RTT is estimated during the handshake phase of the TCP connection as the time difference between detection of the SYN at the measurement point and the subsequent ACK from the same source.

This approach is extended to detect losses of the first SYN packet, in which case no estimation is made [1]. The authors also describe detection of a delayed acknowledgement, invalidating the RTT estimate [1].

This approach yields a single estimate for each TCP flow. However, as shown by Jaiswal et al. [2, 5], the RTT as estimated during the handshake phase is not necessarily applicable for the duration of the TCP flow. Also, this approach occasionally provides no estimate.

3.2. Slow-Start Estimation

Jiang and Dovrolis [1] propose a second technique. The Slow-Start algorithm estimates RTT as the time difference between the first packet of subsequent window transmissions – defined as a burst of at least four packets of the maximum transmission unit size – detected at the measurement point. The algorithm is applied to the first two valid data bursts of a flow [1].

The authors propose detection of lost packets to invalidate the RTT estimate [1]. As for SYN-ACK, this also yields either one or no RTT estimate, which may not be indicative for the entire TCP stream.

3.3. TSTAT

The publically available **tstat** application [16] can operate on both live traffic streams and captured traffic dump files. It matches up each data packet travelling past the measurement point with a corresponding acknowledgement travelling in the opposite direction. The RTT between the measurement point and the data packet destination is estimated as the difference in the capture time of these two packets. The same operation performed on data flows in the reverse direction yields a RTT estimate for the second portion of the TCP flow.

Tstat keeps a running sum of these estimates and uses this to calculate a mean and standard deviation RTT from the measurement point to each end point in a TCP flow. These values are summed to produce a mean and standard deviation TCP RTT estimate.

The reported RTT is indicative of the average RTT experienced by the TCP flow, but the single estimation may not be indicative for the duration of the stream.

3.4. Inferring RTT via cwnd Estimation

Jaiswal et al. [2, 5] propose a technique for estimating a sequence of RTT values for the duration of the TCP flow, which is more reflective of changing

network conditions. The technique uses a Finite State Machine to estimate TCP parameters at the sender based on packets captured at the measurement point.

The algorithm estimates the window size at the sender based on data and acknowledgement packets [5, 9]. Using the window size, two values are calculated:

1. The time (Δt_1) between seeing a data packet and its matching acknowledgement – estimates the RTT between the measurement point and TCP receiver.
2. The time (Δt_2) between seeing an acknowledgement and the next data packet that is triggered as a result of the acknowledgement – estimates the RTT between the measurement point and the TCP sender.

The RTT estimate is the sum of these two values, with one estimate being calculated for each transmission of a window of data, see Figure 2a.

The algorithm also classifies each data packet as normal, retransmitted, duplicated, or re-ordered. When a packet is retransmitted, the TCP sender enters recovery mode, causing a decrease in the window size. During recovery, no RTT estimations are made [2, 5].

The authors note potential errors in estimation, typically due to incorrect window size estimates, and the subsequent over-estimation of the RTT [2, 5].

A further problem occurs when the TCP sender is not-greedy (the overlaying application does not always have data to send). In Figure 2a there is always data to send, upon receipt of an acknowledgement a new data packet is sent, allowing accurate estimation of Δt_2 .

Figure 2b illustrates a non-greedy sender, when an acknowledgement is received, there is no data to send. Some time later, data is available, this extra think-time, t_{think} , is included in the estimate of Δt_2 . The algorithm places a loose upper bound on RTT estimation – the actual RTT is typically not higher than the estimate.

4. A Different Technique

The aim is to generate a sequence of RTT estimates that are representative of the instantaneous network state. We propose a new algorithm which calculates a lower bound on the RTT. The algorithm is an extension of the one used by **tstat**. We also present a Jitter – or RTT variation – estimation algorithm.

4.1. RTT Estimation Algorithm

Jaiswal et. als [2, 5] RTT estimation algorithm uses two values – Δt_1 and Δt_2 . The majority of the error occurs in estimating Δt_2 . Δt_1 estimation is more accurate, since we are effectively calculating the RTT for a TCP stream sourced from the measurement point.

Rather than estimating Δt_1 and Δt_2 for each flow, our approach seeks to minimise error by estimating Δt_1 for the TCP flow in each direction and summing these estimates, the same approach used by **tstat**.

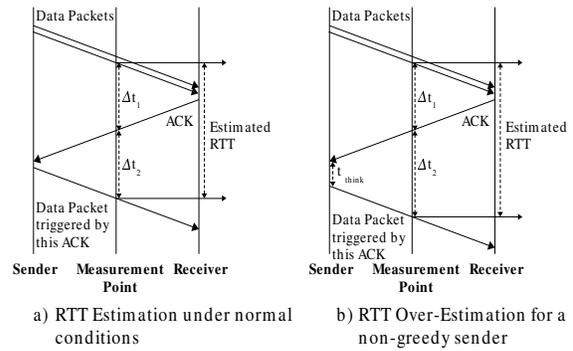


Figure 2. TCP Sample Based RTT Estimation

The algorithm is shown in full in Figure 3. For every data packet, we record both the sequence number and timestamp (obtained using the local clock on the measurement device). For every acknowledgement, we locate the most recently seen data packet with a lower sequence number. The RTT sample is the difference in the two timestamps.

Estimation is suspended during TCP recovery – where packet causality is problematic – and resumes when all retransmitted packets are acknowledged.

The RTT samples are smoothed using Jacobsons Algorithm [4, 6] (1) to generate a running estimate.

$$RTT_{t+1} = \frac{7}{8} \cdot RTT_t + \frac{1}{8} \cdot RTT_{\text{sample}} \quad (1)$$

Smoothing is applied for the same reasons that it is applied in the TCP stack at the end-hosts – to keep the RTT estimate relevant to current network conditions, while minimising the effect of outliers (one-off large or small RTT samples) on the RTT estimate.

We define the mean TCP stream RTT as being the mean of all RTT estimates for that stream (2), as per **tstat**. This value cannot be considered representative for the duration of the flow, but can be useful if network conditions are stable for the stream duration.

$$RTT_{\mu} = \frac{1}{n} \cdot \sum^n RTT_t \quad (2)$$

4.2. Estimating TCP Jitter Characteristics

Jitter is a measure of RTT variability. The aim is to generate a sequence of values that are representative of the current network jitter. Jitter samples are calculated as the absolute difference between the current RTT sample and estimation (3).

$$Jitter_{\text{sample}} = |RTT_{\text{sample}} - RTT_t| \quad (3)$$

Running Jitter estimates are produced through application of Jacobsons algorithm [4, 6] (4).

ProcessDataPacket:

input:

timestamp, seqno

if seen_before seqno calcRTT = *false*
store(timestamp, packet.seqno)

ProcessAcknowledgement:

input:

timestamp, ackno

if no stored values **return**

if no stored sequence numbers < ackno **return**

ts = max_stored_timestamp(stored.seqno < ackno)
remove_stored_info(stored.seqno < ackno)
RTTsample = timestamp - ts;

if calcRTT = *false*

if no_retransmitted_packets_stored calcRTT = *true*
return

if RTT > 0

RTT = (7 * RTT + RTTsample) / 8
Jittersample = Abs(RTT - RTTsample)
if Jitter > 0
Jitter = (7 * Jitter + Jittersample) / 8
else
Jitter = Jittersample

else

RTT = RTTsample

Figure 3. Running RTT Estimation Algorithm

$$Jitter_{t+1} = \frac{7}{8} \cdot Jitter_t + \frac{1}{8} \cdot Jitter_{sample} \quad (4)$$

As per RTT estimation, Jitter is estimated from the measuring point to each end-host, these are summed to obtain Jitter for the TCP flow. We define mean TCP Jitter as the average of all estimates for that stream (5).

$$Jitter_{\mu} = \frac{1}{n} \cdot \sum^n Jitter_t \quad (5)$$

These two algorithms produce running RTT and Jitter estimates for the duration of the TCP flow. The mean of these estimates should only be used under consideration of variability of network conditions.

4.3. Sources of Estimation Error

Under normal conditions, each acknowledgement passing the measurement point is matched to the corresponding data packet and a RTT sample is calculated. Under non-normal TCP conditions:

- **Packet retransmission** – Detected as a lost packet. RTT estimation is suspended until all retransmitted packets are acknowledged. This ensures that an

acknowledgement for the original packet is not matched with the duplicate packet.

- **Network re-ordering of packets** – Acknowledgements are matched with the most recently seen (data packet) sequence number. Imagine two data packets are reordered such that packet 2 arrives at the destination first. An acknowledgement is only returned after packet 1 arrives. This is correctly matched with packet 1 to obtain the RTT sample.
- **Network Duplication of packets** – If this occurs prior to the measurement point, we see two data packets with the same sequence number. This is treated as a retransmission, and a sample will not be calculated. If it occurs after the measurement point, the acknowledgement will (correctly) be for the first of the duplicate packets to reach the receiver.

Further, the algorithm takes into account the situation of a non-greedy sender and, as per TCP stack implementation, the RTT estimate includes the effects of delayed acknowledgements at the receiver, estimating TCP RTT rather than network RTT.

There are some problems using this algorithm to calculate the RTT of a TCP stream:

- **Unidirectional TCP Stream** – If data is only flowing in one direction, the RTT estimates are accurate only from the measurement point to the TCP receiver. The other half of the RTT estimation can only be performed based on the TCP handshake packets. This limits the accuracy of the estimation for this half of the TCP stream due to minimal data.
- **Half-Duplex TCP Streams** – If the TCP stream consists of bursts of data flowing in one direction followed by bursts in the opposite direction, RTT samples for each half of the TCP stream will not be generated concurrently. Rapid changes in network conditions may not be reflected in RTT estimates.
- **Location of Measurement Point relative to Sender** – The relative location of a heavy sender to the measurement point can affect RTT estimation. If the measurement point is close to the receiver and there is little reverse data flow, the error in estimation between the measurement point and the sender is high. Jaiswal's algorithm [2, 5] suffers from a similar problem.

Jitter calculation is based on the RTT estimates and is subject to similar sources of error.

5. Experimental Results

Two FreeBSD workstations are configured with static IP addresses 192.168.0.1 and 192.168.0.2, and are connected to each other through a third FreeBSD machine configured as an Ethernet bridge running DummyNet, enabling manual configuration of network parameters to all packets traversing the bridge [11].

DummyNet allows specification of delay, packet loss rate and throughput [11]. After setting these network characteristics to known parameters, we can compare them against the estimated RTT and Jitter.

5.1. RTT Estimation

DummyNet was configured with a network delay between 0 and 200ms in regularly increasing intervals of 20ms, and with a packet loss rate of between 0 and 5% in regularly increasing intervals of 1%. A TCP stream is generated using `scp` to copy a 10MB file between the two hosts.

In order to check the estimated RTT against the configured delay, delayed acknowledgements are disabled. Delayed acknowledgements incur an extra delay at the receiver prior to being sent, particularly in the case of lost or dropped packets.

In all cases, the algorithm correctly estimated the RTT of the link as the DummyNet configured delay. The running estimate was consistent for the duration of the stream and for all configured packet loss rates.

5.2. Change in Network Delay Mid-Stream

The network delay parameter is changed midstream – when the file transfer is approximately half complete. The test conditions are listed in Table 1.

Table 1. Change in DummyNet Delay

<i>Packet Loss Rate</i>	<i>Initial Network Delay</i>	<i>Final Network Delay</i>
0.00%	150ms	200ms
0.00%	150ms	120ms
2.00%	40ms	80ms
2.00%	200ms	100ms
4.00%	100ms	120ms
4.00%	40ms	0ms

Of interest is the progressive estimated RTT values. We would like to confirm the initial RTT estimate and how quickly the estimate converges to the new network conditions. The results are presented in Figure 4 and 5 and are centred on the transition period during the change in network delay.

The RTT estimate converges to the new value quickly, after approximately 90 RTT samples. There is a small period during the changeover period when the Jitter estimate goes up, but this returns to zero once the RTT estimate has reacquired a steady state value.

5.3. Jitter Estimation

DummyNet does not directly support configurable Jitter [11, 12], and while regular random changes of the

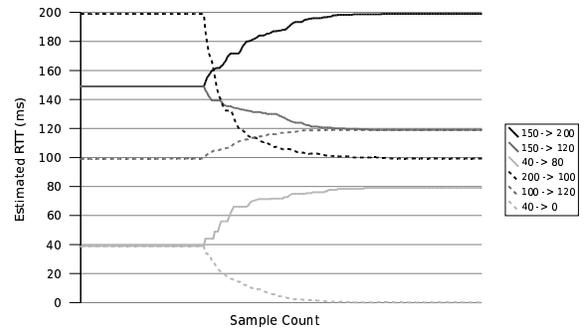


Figure 4. Estimated RTT – Delay Change

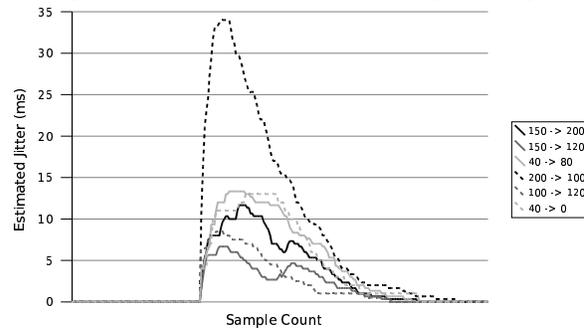


Figure 5. Estimated Jitter – Delay Change

delay characteristic can be used to emulate jitter [12], this is not applicable for TCP streams containing bursts of data packets. In this case, each data packet burst is delayed by the current DummyNet delay setting, emulating a change in RTT as per the previous section rather than actual Jitter to network packet arrival times.

We replace the FreeBSD based DummyNet bridge with a Linux based NISTNet router [13], under which jitter (or RTT standard deviation) is configurable.

Use of NISTNet results in large scale reordering of packets, each packet is delayed by a statistically determined period of time, there is a high probability that a packet will pass through NISTNet more quickly than an earlier packet – this can be verified via a simple ping test. To minimise packet reordering, NISTNet must run using a correlation factor, the documentation recommends a “realistic” value of 0.8 [14].

NISTNet delay correlation ensures that incoming packets are delayed by a similar time period to the previous packet [13, 14]. This minimises but does not exclude the possibility of packet reordering.

The NISTNet correlation method changes the mean and standard deviation of the delay variation. NISTNet compensates by calculating a new mean and standard deviation that, in conjunction with the correlation factor, produces correct delays [14].

However, compensation only functions correctly for small values of standard deviation when compared to the mean. A ping test will verify that generated delays are incorrect when the standard deviation is larger than 20% of the mean, the measured delays are:

- **0 milliseconds** – The compensation sets a negative mean delay in the NISTNet core.
- **4x the configured delay** – The compensation sets an overly large delay in the NISTNet core.

This problem with NISTNet can be circumvented by choosing a delay standard deviation less than 20% of the configured mean, as per the settings in Table 2.

Table 2. NISTNet Delay and Jitter Settings

<i>Packet Loss Rate</i>	<i>Mean Network Delay</i>	<i>Std. Dev. Network Delay</i>
0.00%	150ms	10ms
0.00%	80ms	11ms
2.00%	100ms	7ms
2.00%	40ms	5ms
4.00%	180ms	13ms
4.00%	120ms	16ms

Since the delay is configured in both directions, the NISTNet delay is calculated using (6), and the standard deviation is calculated using (7).

$$\mu_{link} = \frac{\mu_{total}}{2} \quad (6)$$

$$\sigma_{link}^2 + \sigma_{link}^2 = \sigma_{total}^2 \quad (7)$$

Again we transfer the 10MB test file for each test case. The mean estimated RTT and Jitter is presented in Table 3. NISTNet delays each packet by a certain mean + standard deviation delay, while the estimated values are biased toward the most recent packets. The RTT estimate tracks minor variations in delay. As jitter estimation is based on the moving RTT estimate, its mean may be smaller than the actual network jitter.

Table 3. Configured vs. Mean Estimated RTT and Jitter

<i>Configured Parameters</i>	<i>Estimated Parameters</i>
150 ± 10ms	150.42 ± 6.81ms
80 ± 11ms	83.15 ± 7.67ms
100 ± 7ms	100.17 ± 4.95ms
40 ± 5ms	40.71 ± 3.55ms
180 ± 13ms	181.84 ± 9.35ms
120 ± 16ms	123.68 ± 11.46ms

We plot the progressive RTT estimate for all six test cases in Figure 6. While the mean RTT estimate is accurate, there is a degree of fluctuation with an error about equal to the configured jitter. Because the estimate is a weighted average of recent RTT samples, a sequence of lower or higher RTT samples will produce a temporary variation in the estimated RTT.

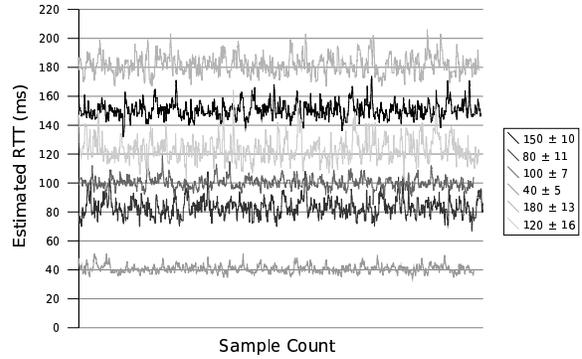


Figure 6. Estimated RTT in a Jittered Network

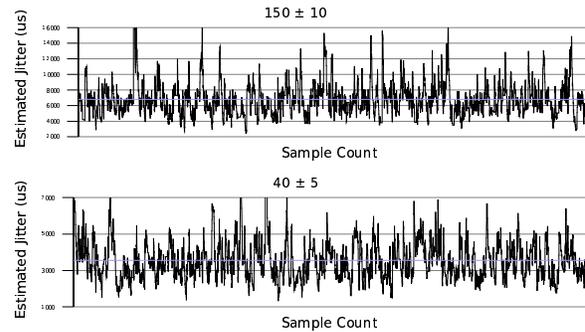


Figure 7. Estimated Jitter in a Jittered Network

Running jitter estimates are plotted in Figure 7 – graphs are shown for two test cases, and are indicative of the remaining test cases. We see greater variation which also appears to be related to the configured jitter. This is primarily due to the estimate being a weighted average of recent measurements, the estimate tracks immediate rather than average network conditions.

5.4. Performance under Network Congestion

We next evaluate the performance of RTT and Jitter estimation in the presence of network congestion. Three clients are connected to the bridge via a switch. Each client is configured to insert the TCP stack RTT estimate into the TCP header. The bridge is bandwidth limited to 500kbps to ensure network congestion.

Delayed acknowledgements were disabled on all machines. Each client was configured to initiate a TCP stream with a duration of thirty seconds, starting transmission at ten second intervals. Traffic was captured and analysed on the Ethernet bridge.

DummyNet was configured with a bi-directional delay from 0ms to 125ms in regularly increasing intervals of 25ms. Results are displayed for a delay of 75 ms, the results being indicative of all tests. Figure 8 shows the estimated RTT for each client, the **tstat** estimated RTT and the ping estimated RTT. Figure 9 shows the corresponding RTT as estimated by the TCP stack in each client. Figure 10 shows Jitter estimates.

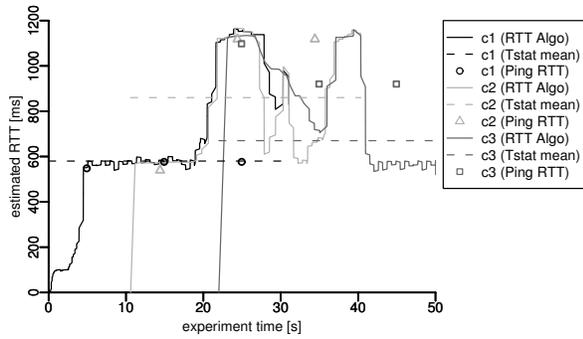


Figure 8. Estimated RTT - no Delayed ACKs

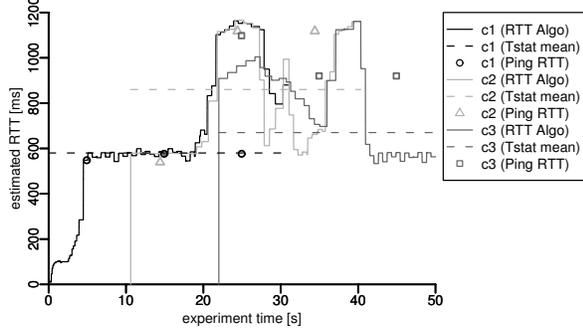


Figure 9. TCP Stack – no Delayed ACKs

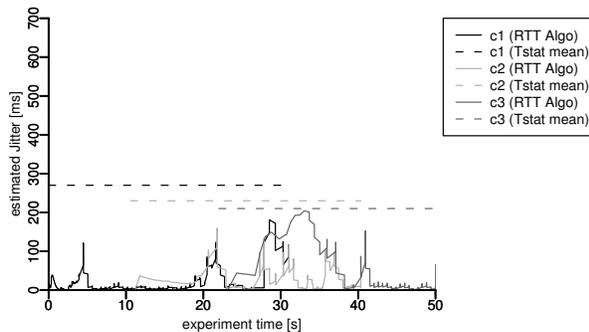


Figure 10. Estimated Jitter – no Delayed ACKs

The estimated RTT tracks the RTT estimate in the TCP stack extremely well, often converging to a steady state more quickly. The graphs also highlight how a single estimate does not reflect varying conditions.

The same experiments were repeated with delayed acknowledgements, Figure 11, 12 and 13 show the results for a delay of 75ms. Again the passive RTT estimate tracks the estimate in the stack well.

Given network congestion, we expect packet retransmissions and the subsequent impact of delayed acknowledgements. This occasional extra delay will lead to a greater variability in the RTT estimates, which is echoed in the higher Jitter estimates.

Finally we repeat the above tests with the Ethernet bridge providing a different delay for each client – client 1(100ms), client 2(150ms) and client 3(200ms).

Figure 14, 15 and 16 plot the results for the case of no delayed acknowledgements. Results for delayed acknowledgements reflect those in Figure 11, 12 and

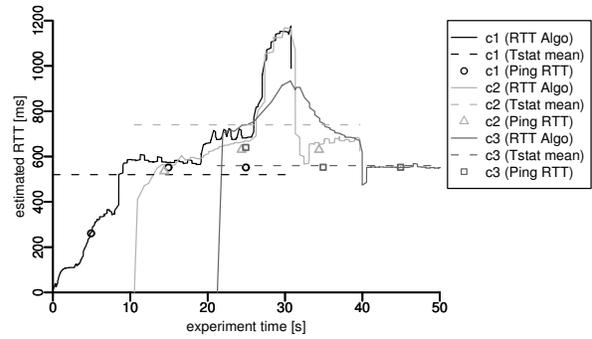


Figure 11. Estimated RTT - Delayed ACKs

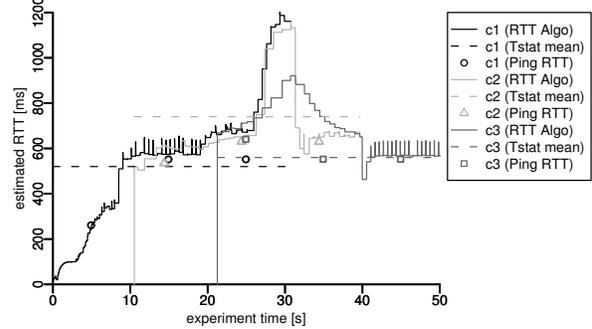


Figure 12. TCP Stack – Delayed ACKs

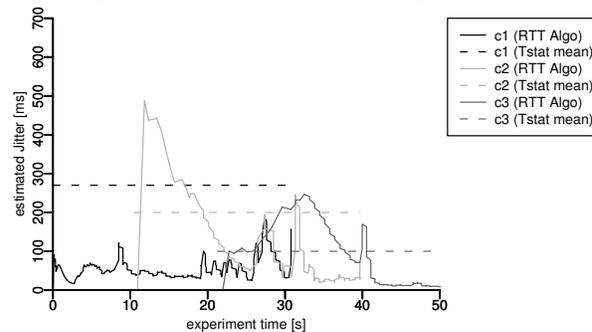


Figure 13. Estimated Jitter – Delayed ACKs

13. Again we see excellent tracking of the TCP RTT estimator by the passive estimation algorithm.

6. Conclusion

RTT estimation of a TCP stream monitored at a midpoint is an inaccurate process. A single estimate [1] does not take into account network variability during the TCP flow [2, 5]. Instead, a sequence of RTT estimates for the duration of the flow produces snapshots in time of current network conditions.

Continuous RTT estimation involves predicting events occurring at each endpoint of the TCP flow. The algorithm presented by Jaiswal et al. [2, 5] has certain conditions under which the RTT estimate is inaccurate, it also predicts an upper bound on the actual link RTT.

In this paper we present a different RTT estimation algorithm. Like that of Jaiswal et al. [2, 5], this also has certain conditions where estimation is inaccurate.

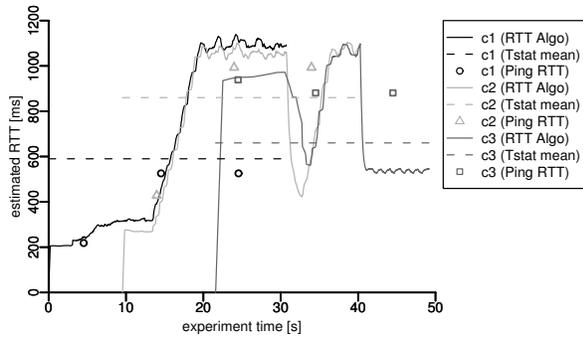


Figure 14. Estimated RTT – Different Delays

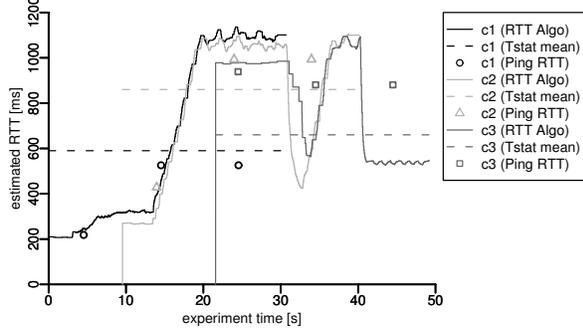


Figure 15. TCP Stack – Different Delays

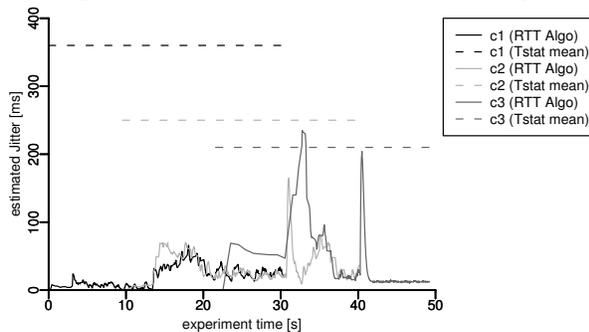


Figure 16. Estimated Jitter – Different Delays

However, the conditions under which errors occur are different, furthermore, this algorithm determines a lower bound on the actual RTT. A combination of these two techniques could be used to produce a bounded RTT estimate. We also apply Jacobsons algorithm to minimise the effect of transient network packet transfer times on the running RTT estimates.

We also present a means of estimating TCP stream jitter from the running RTT estimates, this technique can be applied to a stream of estimates generated from another algorithm, including Jaiswal et al. [2, 5].

In the case of artificially introduced jitter, the algorithms accurately estimated the configured network conditions. Where the network was congested, the algorithm tracks the RTT estimated in the TCP stack of the sender with great accuracy, often converging to a steady state value more quickly. The running Jitter estimates indicate the stability of network conditions.

Running estimates are clearly superior to the output generated by existing tools such as **tstat** which produce a single RTT mean and standard deviation – which do not accurately reflect changing network conditions.

7. References

- [1] H. Jiang and C. Dovrolis "Passive Estimation of TCP Round-Trip Times", ACM SIGCOMM Computer Communications Review, vol. 32 no. 3, 2002, pp. 75-88
- [2] S. Jaiswal, G. Iannaccone, C. Diot, J. Kurose, and D. Towsley "Inferring TCP Connection Characteristics Through Passive Measurements", Proceedings of IEEE InfoCom2004, March 2004
- [3] M. Allman, V. Paxson and W.R. Stevens "TCP Congestion Control", IETF RFC 2581, <http://www.ietf.org/rfc/rfc2581.txt>, April 1999.
- [4] V. Paxson and M. Allman "Computing TCP's Retransmission Timer", IETF RFC 2988, <http://www.ietf.org/rfc/rfc2988>, November 2000
- [5] S. Jaiswal, G. Iannaccone, C. Diot, J. Kurose and D. Towsley "Inferring TCP Connection Characteristics Through Passive Measurements (extended version)", Technical Report RR03-ATL-070121, Sprint ATL, July 2003.
- [6] V. Jacobson "Congestion Avoidance and Control", Proceedings of ACM SIGCOMM Communications Architectures and Protocols Symposium, 1988, p 314.
- [7] J. Padhye and S. Floyd "On Inferring TCP Behaviour", Proceedings of ACM SIGCOMM2001, August 2001
- [8] J. Aikat, J. Jaur, F. Smith and K. Jeffay "Variability in TCP Roundtrip Times", Proceedings of ACM SIGCOMM Internet Measurement Workshop, October 2003, pp. 279-284
- [9] S. Jaiswal, G. Iannaccone, C. Diot, J. Kurose and D. Towsley "Measurement and Classification of Out-of-Sequence Packets in a Tier-1 IP Backbone", Proceedings of IEEE Infocom2003, April 2003
- [10] V. Paxson, M. Allman, S. Dawson, W. Fenner, J. Griner, I. Heavens, K. Lahey, J. Semke and B. Volz "Known TCP Implementation Problems", IETF RFC 2525, <http://www.ietf.org/rfc/rfc2525.txt>, March 1999
- [11] L. Rizzo "Dummynet: a Simple Approach to the Evaluation of Network Protocols", ACM Computer Communication Review, vol. 27 no. 1, January 1997
- [12] G. Armitage and L. Stewart "Some Thoughts on Emulating Jitter for User Experience Trials", Proceedings of NetGames2004 WorkShop, SIGCOMM2004, Portland, August 2004.
- [13] NISTNet – Network Emulation Package, National Institute of Standards and Technology, <http://www.itl.nist.gov/div892/itg/carson/nistnet>, August 2004
- [14] M. Carson and D. Santay "NIST Net – A Linux-based Network Emulation Tool", ACM Computer Communications Review, vol. 33 no. 3, July 2003
- [15] M. Mellia, A. Carpani, and R. Lo Cigno "TStat: TCP STatistic and Analysis Tool", Proceedings of QoSIP2003, February 2003
- [16] TSTAT – TCP Statistic and Analysis Software Tool, Politecnico di Torino, <http://tstat.tlc.polito.it>, Accessed January 2005