acmqueue

Passively Measuring TCP Round-trip Times

A close look at RTT measurements with TCP

Stephen D. Strowes, Boundary Inc.

Measuring and monitoring network RTT (round-trip time) is important for multiple reasons: it allows network operators and end users to understand their network performance and help optimize their environment, and it helps businesses understand the responsiveness of their services to sections of their user base.

Measuring network RTT is also important for TCP (Transmission Control Protocol) stacks to help optimize bandwidth usage. TCP stacks on end hosts optimize for high performance by passively measuring network RTTs using widely deployed TCP timestamp options carried in TCP headers. This information, if utilized, carries some distinct operational advantages for services and applications: hosts do not need to launch out-of-band ICMP (Internet Control Message Protocol) echo requests (pings), nor do they need to embed timing information in application traffic. Instead, hosts can passively measure RTT representative of full-path network latency experienced by TCP traffic.

Understanding network delay is key to understanding some important aspects of network performance. The time taken to traverse the network between two hosts affects how responsive services are, and it affects the effective bandwidth available to end hosts. Measuring this information passively on servers can help provide a fine-grained indication of service responsiveness from the customer's perspective, and it simultaneously offers a network-distance metric for customers that is more useful than coarse-grained and often-inaccurate IP-based geolocation.

Measuring RTT to many hosts or customers is nontrivial. One solution is active probing, in the form of ICMP echo requests and responses (that is, ping), but this incurs additional network load. In some cases ICMP traffic is deprioritized, dropped completely, or routed over a different path than TCP traffic. When none of this is true, there is still the possibility that RTT is being measured to a NAT (Network Address Translator) and not to the end host exchanging data.

Another possible solution for measuring network RTT is to measure the application-layer responsiveness. This, however, implies an application-specific, ad hoc measurement performed by embedding IDs or timestamps into the TCP bytestream, which may give a misleading, inflated measurement for network RTT, depending on network conditions (more on this later).

Neither of these solutions is totally satisfactory because neither is guaranteed to measure the network RTT that affects application traffic. The timestamp information carried in TCP headers, however, offers another solution: it is effectively a network-layer RTT measurement that passes through most middleboxes such as NATs and firewalls and measures the full-path RTT between both hosts on a connection. This information provides the only noncustom RTT estimation solution available to end hosts. Tools such as tcptrace can calculate RTT using this state, but any software that can inspect packet headers (usually accomplished via libpcap) or interrogate the local system for such kernel state can passively gather network RTTs for all active connections.

The key differences between these measurements and how differing network conditions affect them are not obvious. The purpose of this article is to discuss and demonstrate the passive RTT measurements possible using TCP traffic.

BACKGROUND

TCP offers a reliable bytestream to the applications that use it; applications send arbitrary amounts of data, and the TCP layer sends this as a series of segments with sequence numbers and payload lengths indicating the chunk of the bytestream each segment represents. To achieve an ordered byte stream, TCP retransmits segments if they go missing between the source and destination (or, if an acknowledgment for a received segment goes missing between the destination and the source). To improve performance in all network conditions, the TCP stack measures RTT between it and the other host on every connection to allow it to optimize its RTO (retransmission timeout) appropriately and optimize the time taken to recover from a loss event.

The original TCP specification contained no mandatory, dedicated RTT calculation mechanism. Instead, TCP stacks attempted to calculate RTTs by observing the time at which a sequence number was sent and correlating that with a corresponding acknowledgment. Calculation of RTTs using this mechanism in the presence of packet loss, however, makes accurate measurement impossible.[13] TCP timestamps were defined to permit this calculation independently at both ends of a connection while data is being exchanged between the two hosts. TCP timestamps offer a mechanism for calculating RTT that is independent of sequence numbers and acknowledgments.

The algorithm for calculating RTT from a TCP flow between two hosts, documented in RFC 1323,[3] is commonly used by both end hosts on a connection to refine the RTO to improve the performance of TCP in the presence of loss. The mechanism is enabled by default in modern operating systems and is rarely blocked by firewalls, and thus appears in most TCP flows; the TCP Timestamp option is known to be widely deployed in the wild.[5]

RTT is calculated continuously for each connection for as long as data is exchanged on those connections. TCP calculates RTT for packets exchanged on a per-connection basis and computes the exponential moving average of these measurements, referred to as SRTT (smoothed RTT). The TCP stack also maintains the variance in the measured RTT, the RTTVAR. The SRTT that the TCP stack calculates for a connection determines the RTO value. Given variables G, which is the system clock granularity, and *K*, which is set to 4,[8] the RTO is calculated as follows:

```
RTO = SRTT + max(G,K * RTTVAR)
```

The RTO is used to optimize the time the TCP stack waits, having transmitted a TCP segment, for a corresponding acknowledgment prior to retrying the send operation. Accurate measurements of RTT between two hosts allow TCP to tune its RTO accurately for each active connection.

Understanding the state contained within the TCP headers carried in most TCP traffic can help application designers or network operators understand the network RTTs experienced by application traffic. Many applications with realtime constraints use TCP to transport their traffic, which is acceptable within certain bounds.[1] It is useful to understand how the bytestream semantic can affect realtime performance. TCP timestamps are optional fields in the TCP header, so although they are extremely useful and carried in most traffic, they are not strictly required for TCP to function. The values are held in two 4-byte header fields: TSval (Timestamp Value) and TSecr (Timestamp Echo Reply). Each host involved in a connection emits TSval timestamps to the other host whenever a TCP segment is transmitted, and awaits the corresponding TSecr in return. The time difference measured between first emitting a TSval and receiving it in a TSecr is the TCP stack's best guess as to RTT. Timestamp here is an arbitrary value that increments at the granularity of the local system clock; it is not a timestamp that can be interpreted independently, such as number of seconds since the epoch.

By way of example, in figure 1 time progresses from top to bottom, and the horizontal lines indicate realtime incrementing (for example, in milliseconds). Two hosts, A and B, have an open connection and are exchanging packets. In reality the two hosts have differing clocks, but for simplicity assume they are perfectly synchronized.

The example operates as follows:

Host A emits a TCP segment that contains the timestamp options

```
TSval = 1, TSecr = 0
```

TSecr is set to 0 because no TSval from B has been observed at A; this usually indicates A is opening a connection to B. Host B receives this timestamp at time 1; at time 2, host B emits a TCP segment back to A, which contains the values

```
TSval = 2, TSecr = TSvalA = 1
```

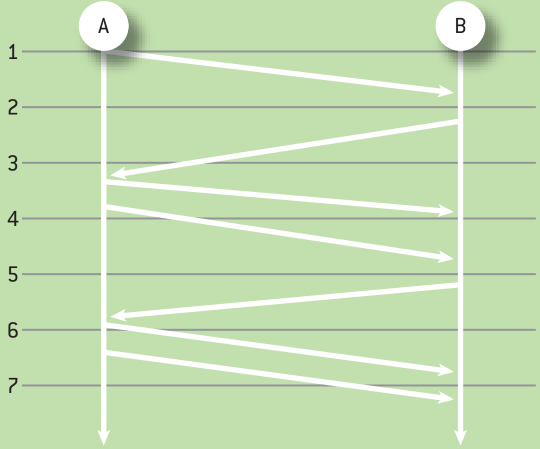
These are received at host A at time 3. Given this echoed value and the current time, host A knows that RTT in this instance is approximately 2 ms.

Subsequently, the next two segments that A emits both carry the values

```
TSval = 3, TSecr = TSvalB = 2
```

FIGURE 1

Simplified Demonstration of TCP Timestamp Exchange



The first of these is received at host B at time 4, so host B can also calculate an RTT of 2 ms. Given the two echoes of the same timestamp received, the minimum duration is assumed to be closest to the actual network delay; if network delay changes, future exchanges will measure this. Continuously sending values to the other host and noting the minimum time until the echo reply containing that value is received allows each end host to determine the RTT between it and the other host on a connection.

The caveat is that for a TSval to be considered useful, the TCP segment must be carrying data from the application. TCP segments can legitimately carry a zero-byte payload, most commonly when acknowledging a data segment, or when TCP keepalives are enabled. By requiring that valid TSvals come only from TCP segments carrying data, the algorithm is less likely to measure breaks in the communication between the hosts, where data exchange pauses for a time, then restarts using the last received TSval as the TSecr. This also implies that on a TCP flow in which data is exchanged exclusively in one direction, only one of the hosts will be able to calculate RTT. Usually, however, some application layer chatter occurs in both directions.

Finally, RTT calculation can be performed on any host that is forwarding traffic, not just end hosts, so full-path RTTs on all connections within a network can be calculated from its gateway host, for example. All that is necessary to compute accurate RTTs is that both directions of a connection pass through the monitoring host. Whether this is the case often relies on the network architecture, but it is known that paths on the Internet are normally not symmetric.[2] Running this algorithm on a gateway node for a network through which all traffic passes, however, allows the RTT calculation to take place passively on all connections from just one location.

DEMONSTRATING RTT MEASUREMENTS

The network is a shared resource, and multiple factors can affect TCP RTT calculation. This section broadly covers some of these factors and demonstrates where the TCP RTT calculation differs from the RTT perceived by applications. The aim is to demonstrate parity with ICMP's RTT estimates, assuming all else is equal, and how packet loss and excessive buffering affect these measures relative to perceived latency at the application layer.

To demonstrate the responsiveness of RTT measurements, traffic flows were simulated in a virtualized environment. The environment is simple: two Linux hosts were configured on different subnets, and a third Linux host, with packet forwarding enabled, was configured with two network interfaces, one for each subnet.

This forwarding host is used to vary the network characteristics observed between the two end hosts, using the tc (traffic control) tool. Network characteristics are not modified on the end hosts, so their TCP stacks are not directly aware of the configuration for each experiment. For each experiment, an egress delay of 50 ms is set on each interface on the forwarding host, resulting in an RTT of 100 ms between the two end hosts. Each experiment runs for 180 seconds. The maximum data rate is set to 10 Mbps.

On these end hosts, the following components are running:

- Ping is running on both hosts, so each host is sending ICMP echo requests to the other once per second. This measures the ICMP RTT value to establish a "ground-truth" RTT between the hosts.
- A simple client/server pair of programs is running, where the client sends a local timestamp over a TCP connection once every second to the server, and the server echoes the timestamp back to the

client; the client calculates the difference whenever it reads the response out of the bytestream. The client application runs two threads: one for sending and one for receiving. This measures the RTT perceived by the application layer.

- Also running on both end hosts is a pcap (packet capture) reader that observes the TCP timestamp values carried in the TCP headers for the traffic generated by the client/server program from which it calculates RTT, outputting the latest RTT value once every second. The value exported for these experiments is the RTT rather than the SRTT, since the goal here is to examine actual RTT and not an approximation. This calculates RTT passively from TCP timestamps. No other traffic is exchanged between hosts, except during the demonstration of bufferbloat.

The following examples demonstrate:

1. The ability to monitor changing RTT accurately by modifying network latency.
2. The impact of packet loss.
3. The impact of oversized buffers (commonly referred to as bufferbloat).

NAGLE'S ALGORITHM

Before describing these experiments in detail, let's take a look at Nagle's algorithm,[6] which is enabled by default in many TCP stacks. Its purpose is to reduce the number of small, header-heavy datagrams transferred by the network. It operates by delaying the transmission of new data if the amount of data available to send is less than the MSS (maximum segment size), which is the longest segment permissible given the maximum transmission unit on the path, and if there is previously sent data still awaiting acknowledgment.

Nagle's algorithm can cause unnecessary delays for time-critical applications running over TCP. Thus, because the assumption is that such applications will run over TCP in the experiments presented here, Nagle's algorithm is disabled. This is achieved in the client and server by setting the TCP_NODELAY socket option on all sockets in use.

EXPERIMENT 1: CHANGING NETWORK CONDITIONS

When computing RTTs, it is critical that the measurements accurately reflect current conditions. The purpose of this experiment is simply to demonstrate the responsiveness of our metrics to conditions that change in a predictable manner. In this experiment the base RTT (100 ms) is initially set, and then an additional latency (50 ms) is alternately added to and deducted from that base RTT by incrementing the delay on both interfaces at the forwarding host by 25ms. No loss ratio is specified on the path, and no additional traffic is sent between the two hosts.

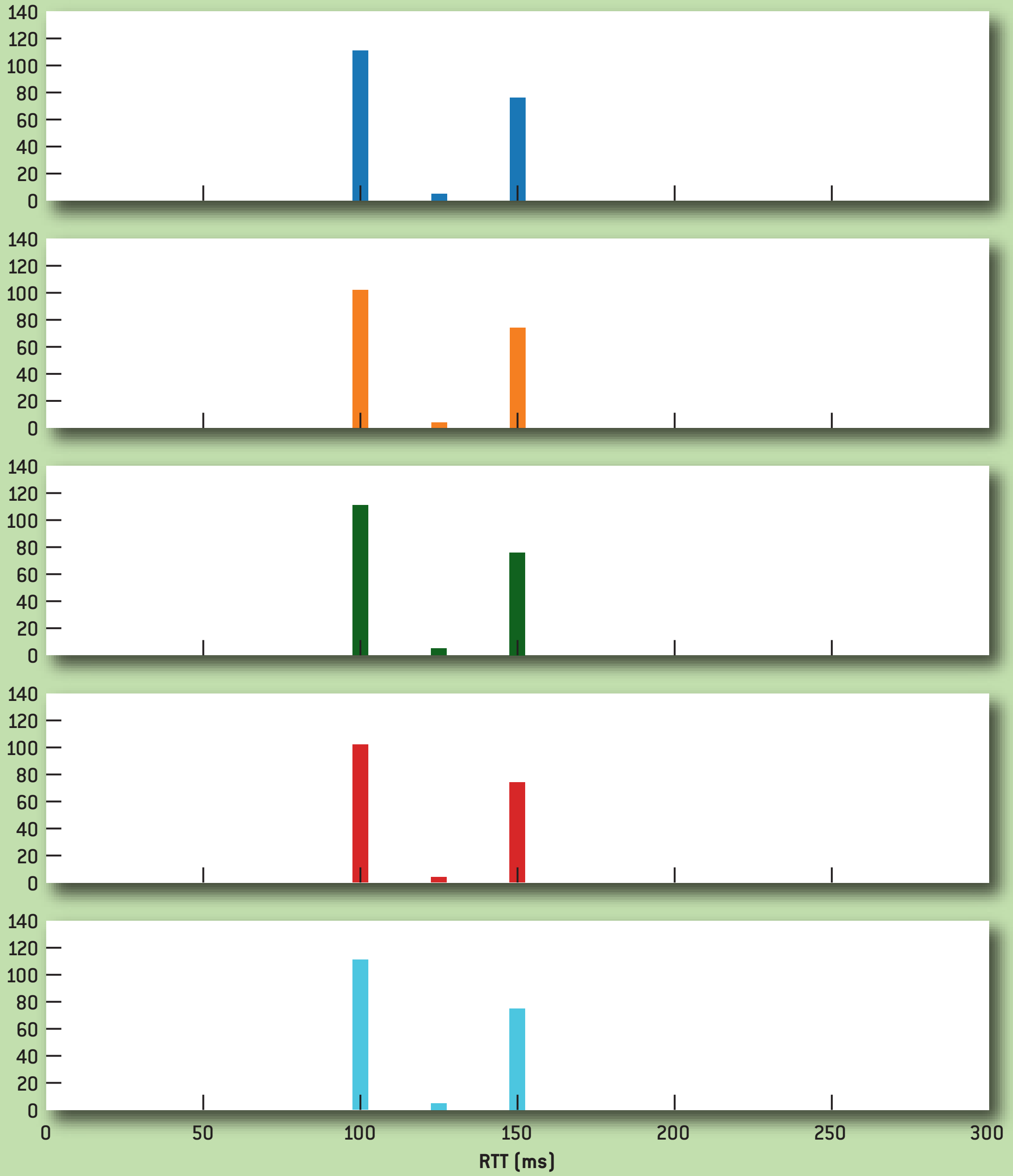
Note that since TCP's RTT calculation is wholly passive, it does not observe variation in RTT if no data is being exchanged. In the presence of traffic, however, it's beneficial that the RTT measurement update quickly. The results of this experiment are shown in figure 2. The measurements taken at all layers indicate a bimodal distribution, which is precisely what should be expected without other network conditions affecting traffic. The three forms of measurements taken are all effectively equivalent, with the mean RTT measured during the experiments varying by no more than 1 percent.

EXPERIMENT 2: PACKET LOSS

Packet loss on a network affects reliability, responsiveness, and throughput. It can be caused by many factors, including noisy links corrupting data, faulty forwarding hardware, or transient glitch-

FIGURE 2

Histograms Indicating Measured RTTs for All Tests



es during routing reconfiguration. Assuming the network infrastructure is not faulty and routing is stable, loss is often caused by network congestion when converging data flows cause a bottleneck, forcing buffers to overflow in forwarding hardware and, therefore, packets to be dropped. Loss can happen on either the forward or the reverse path of a TCP connection, the only indication to the TCP stack being the absence of a received ACK.

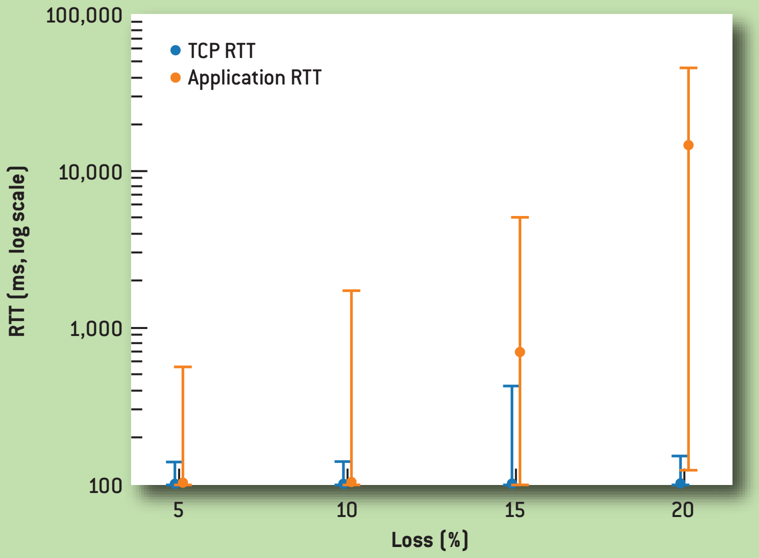
TCP offers applications an ordered bytestream. Thus, when loss occurs and a segment has to be retransmitted, segments that have already arrived but that appear later in the bytestream must await delivery of the missing segment so the bytestream can be reassembled in order. Known as head-of-line blocking, this can be detrimental to the performance of applications running over TCP, especially if latency is high. Selective acknowledgments, if enabled, allow a host to indicate precisely which subset of segments went missing on the forward path and thus which subset to retransmit. This helps improve the number of segments "in flight" when loss has occurred.

In this experiment, packet loss was enabled on the forwarding host at loss rates of 5, 10, 15, and 20 percent, the purpose being to demonstrate that TCP segments are still exchanged and RTTs estimated by TCP are more tolerant to the loss than RTTs measured by the application. The results of this experiment are shown in figure 3. The points represent median values, with 5th and 95th percentiles shown.

In these tests, a 5 percent packet loss was capable of introducing a half-second delay for the application, even though the median value is close to the real RTT of 100 ms; the measured mean for application-layer RTT with 5 percent loss is 196.4 ms, 92.4 ms higher than the measured mean for TCP RTT. The measured means rise quickly: 400.3 ms for 10 percent loss, 1.2 s for 15 percent loss, and 17.7 s for 20 percent loss. The median values shown in figure 3 for application-layer RTT follow a similar pattern, and in this example manifest in median application-layer RTTs measured at around 12 seconds with 20 percent packet loss. The TCP RTT, however, is always close to the true 100-ms distance; although delayed packet exchanges can inflate this measure, the largest mean deviation observed in these tests between TCP RTT and ICMP RTT was a 57.7-ms increase in measured RTT at the TCP layer. The effect of packet loss can be devastating to the responsiveness of TCP applications, but it is clear that passively measuring network-level RTTs is still feasible, and distinct from the

FIGURE 3

RTTs Measured in the Presence of Varying Packet Loss Rates



perceived latency experienced by applications that can be introduced by TCP's in-order delivery semantics.

EXPERIMENT 3: BUFFERBLOAT

Misunderstandings about the relationship between loss prevention and network performance have led to excessive buffering being introduced to forwarding and routing hardware as a loss-avoidance strategy. Often (but not exclusively), this affects commodity CPE (customer premises equipment), and thus directly affects end users. Excessive buffering, however, works against TCP's loss-detection algorithm by increasing delay and thus delaying the time taken for a TCP stack to identify loss and back off—that is, the additional delay introduced by large buffers can disrupt TCP's congestion-control mechanism.

Bufferbloat is a well-known phenomenon[7] where the deepest buffer on a network path between two hosts is eventually filled by TCP. Ostensibly, system designers increase buffer size to reduce loss, but deeper buffers increase the actual time taken for packets to traverse a path, increasing the RTT and delaying the time it takes for TCP to determine when a loss event has occurred. Loss is the driver for TCP's congestion-control algorithm, so increasing buffer size is actually counterproductive.

To demonstrate bufferbloat in this experiment, tc queue sizes were systematically increased from 10 KB, to 100 KB, then 200 KB, then finally 300 KB on the forwarding host, and netcat was used to create a high-bandwidth flow between each of the end hosts prior to starting the client/server application. The intention of the high-bandwidth flow was to fill the longer queues on the forwarding host, demonstrating that the draining time affects application responsiveness.

The results of the experiment are shown in figures 4 and 5. Figure 4 shows the dispersion of RTT measurements as the buffer sizes were increased. The 300-KB test in figure 5 shows very similar RTT measures from both hosts in the ICMP measurements, at the TCP layer, and in the application layer; mean and median values for all layers in these experiments were all within 2 ms of each other. All RTT measures are inflated by the same amount because the excessive buffer size effectively increases the network-layer path length. Given that the test application emits only a handful of packets once per second, the sawtooth pattern is indicative of the netcat data filling a queue, then TCP waiting for the queue to drain prior to sending more of netcat's data, forming a bursty pattern. These filled queues adversely affect the delivery of all other traffic and increase the test application's RTTs, which vary from 100 ms to about 250 ms as a result.

The bufferbloat problem is being actively worked on. Mechanisms such as SACK (Selective Acknowledgments), DSACK (Duplicate SACK), and ECN (Explicit Congestion Notification), when enabled, all help alleviate bufferbloat. Additionally, active queue management strategies such as Codel have been accepted into mainline Linux kernels.

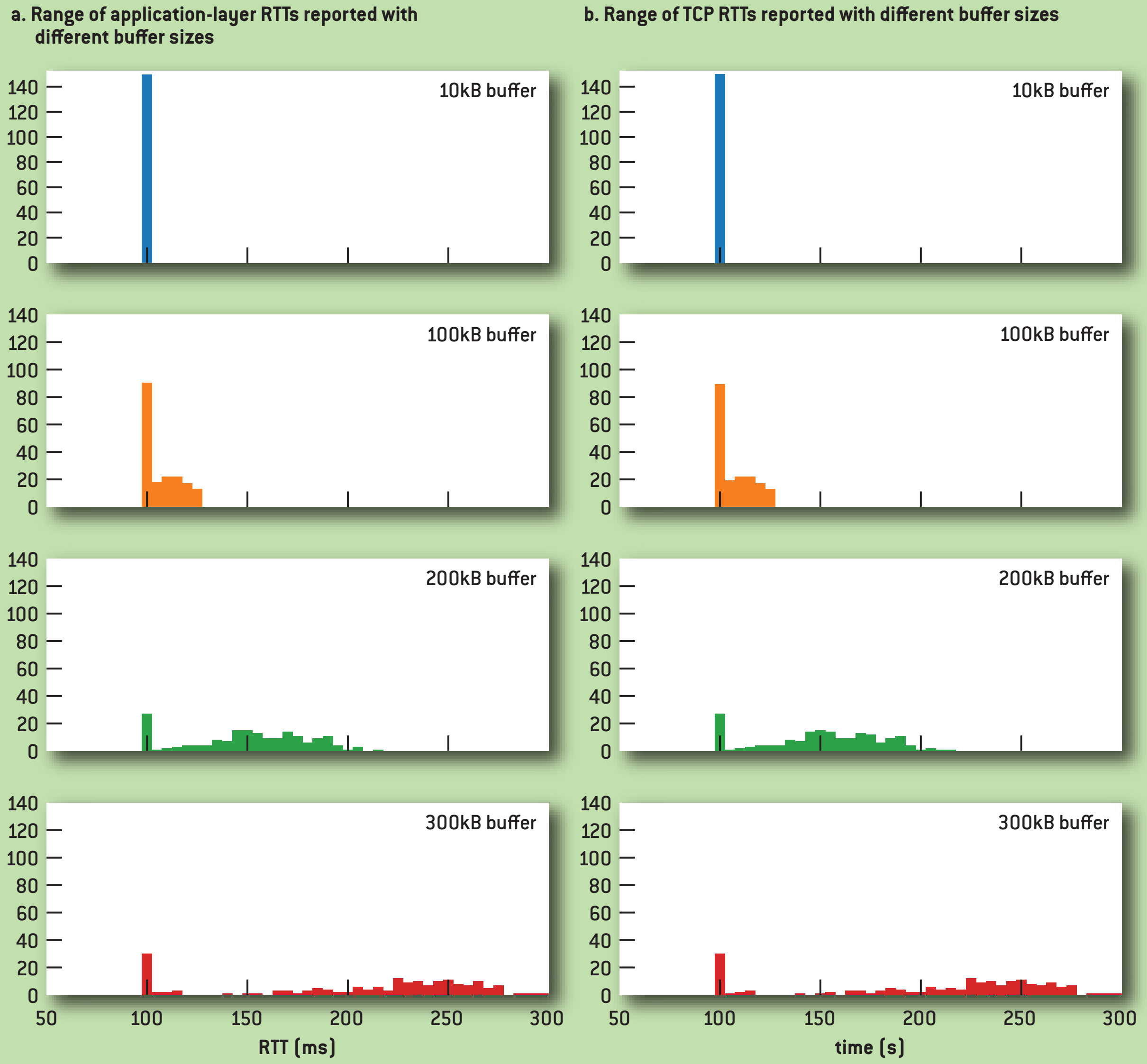
In summary, it is clear that to minimize delays caused by head-of-line blocking in TCP, packet loss must be kept to a minimum. Given that we must expect packet loss as a primary driver of TCP's congestion-control algorithm, we must also be careful to minimize network buffering, and avoid the delays incurred by bufferbloat. The latter requirement in particular is useful to keep in mind when provisioning networks for time-critical data that must be delivered reliably.

RELATED WORK

The key issue when using TCP for time-sensitive applications is that TCP offers a reliable bytestream.

FIGURE 4

RTT Histograms Representing the Difference in RTTs



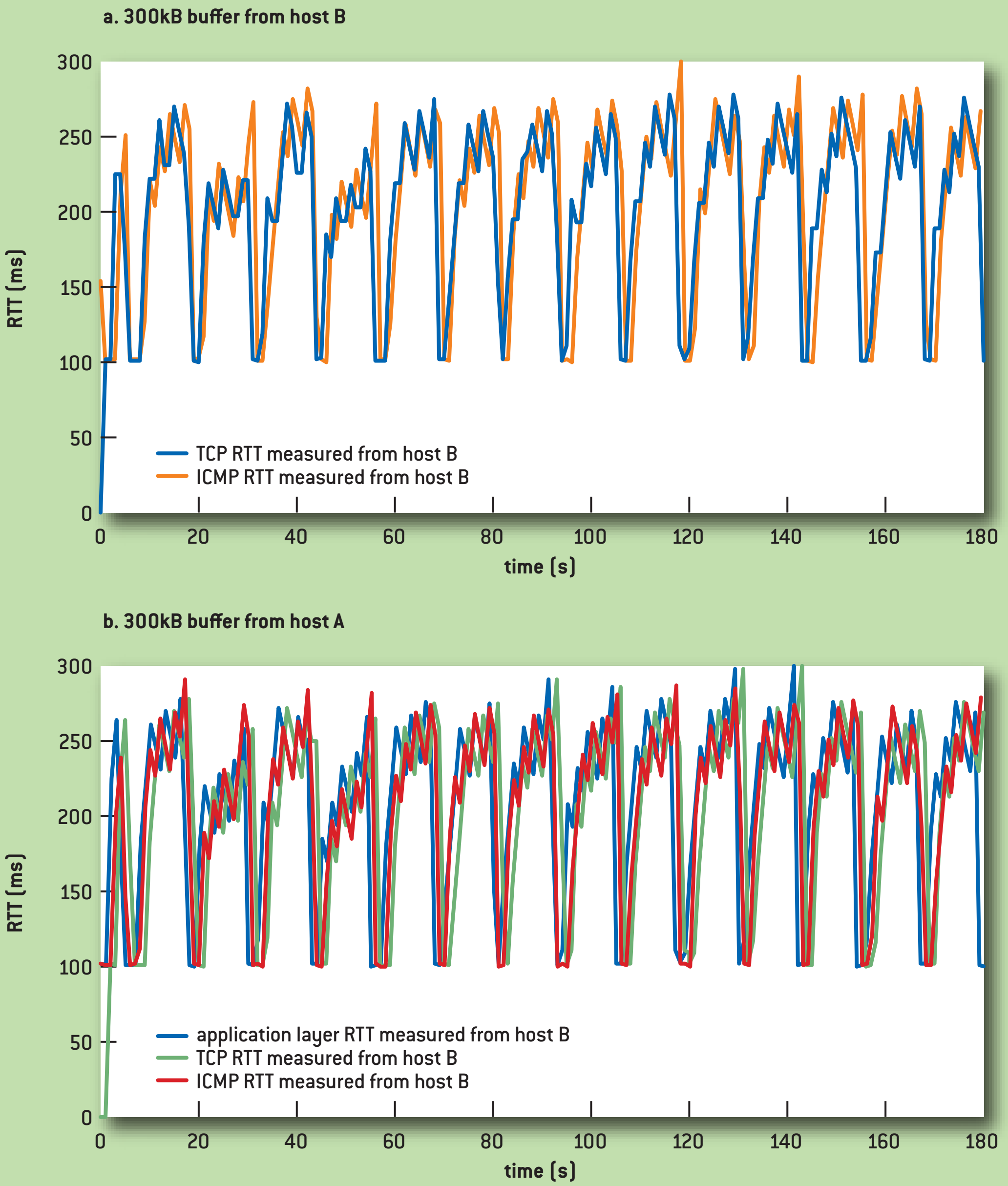
This requirement is distinct from other key aspects of TCP, such as congestion control and flow control. TCP is not suitable for all applications, however. Eli Brosh et al. discuss in more detail the behavior of TCP in the presence of delay and certain acceptability bounds for application performance.[1]

UDP[9] is the most commonly used transport protocol after TCP; it's a datagram-oriented protocol with no congestion control, flow control, or message-ordering mechanisms. It effectively augments the IP layer with UDP-layer port numbers. Without the message-ordering constraint, it is not affected by the head-of-line blocking problem that can affect TCP connections.

UDP alone is not suitable for many applications, however, because reliability is often a

FIGURE 5

Indication of Measured RTT Values in the Presence of Excessive Buffering



requirement, and congestion control is important to permit fair sharing of network resources. Many applications choose to layer protocols on top of UDP, such as RTP (Real Time Protocol) in tandem with RTCP (Real Time Control Protocol),[10] primarily intended for carrying time-sensitive realtime traffic able to handle small amounts of loss. These protocols suit applications such as VoIP that do not require 100 percent reliability and find delay incurred by head-of-line blocking detrimental. RTCP permits coarse-grained congestion control and allows realtime applications to modify their usage by choosing different quality settings for the live stream, but congestion control is not built in per se.

DCCP[4] is a message-oriented, best-effort transport-layer protocol that does not enforce strict ordering on data delivery, does not handle datagram retransmission, but does perform congestion control to conserve network resources. DCCP is useful for a similar set of applications as RTP and RTCP, but the addition of congestion control without potential datagram duplication is important, permitting RTP to run over DCCP with fewer concerns for network resource consumption.

SCTP[11] is also a message-oriented transport protocol, where each message is delivered to the application in order. Strict message ordering, however, is optional, and so the transport can be more responsive for application traffic. SCTP also caters to partial reliability.[12]

Note that bufferbloat is endemic, and other transport protocols are affected in the same way as TCP, but relaxing strict ordering constraints at the transport layer is one approach to improving performance by removing the additional response time incurred when the stream is blocked waiting for missing data. AQM (active queue management) techniques[7] are being deployed in new Linux kernels to help further alleviate bufferbloat without modification to applications.

CONCLUSION

TCP is the most commonly used transport-layer protocol today, and it meets the requirements that many applications desire: it offers a reliable bytestream and handles concerns of retransmissions and congestion avoidance. TCP's semantics can mean that there is a large discrepancy between the RTT measured at the transport layer and the RTT measured by the application reading the bytestream. Thus, TCP is not always the best transport for time-critical applications, but the TCP RTT measurement mechanism that is enabled in most TCP stacks today achieves measurements very close to the ICMP "ground truth" and performs substantially better than a similar echo-based protocol embedded within the TCP bytestream.

RELATED ARTICLES

Gettys, J., Nichols, K. 2011. Bufferbloat: Dark Buffers in the Internet; http://queue.acm.org/detail.cfm?id=2071893.

Currid, A. 2004. TCP Offload to the Rescue; http://queue.acm.org/detail.cfm?id=1005069.

REFERENCES

1. Brosh, E., Baset, S., Misra, V., Rubenstein, D., Schulzrinne, H. 2010. The delay-friendliness of TCP for realtime traffic. *IEEE/ACM Transactions on Networking* 18 (5): 478-491.
2. He, Y., Faloutsos, M., Krishnamurthy, S. and Huffaker, B. 2005. On routing asymmetry in the Internet. In *Proceedings of IEEE Global Telecommunications Conference (GLOBECOM).*
3. Jacobson, V., Braden, R., Borman, D. 1992. TCP extensions for high performance. RFC 1323.
4. Kohler, E., Handley, M., Floyd, S. 2006. Datagram Congestion Control Protocol. RFC 4340.
5. Kühlewind, M., Neuner, S. and Trammell, B. 2013. On the state of ECN and TCP options on the Internet. In *Passive and Active Measurement*. M. Roughan and R. Chang, eds. Lecture Notes in Computer Science 7799: 135-144. Springer Berlin Heidelberg.
6. Nagle, J. 1984. Congestion control in IP/TCP internetworks. RFC 896.
7. Nichols, K., Jacobson, V. 2012. Controlling queue delay. *ACM Queue* 10 (5); http://queue.acm.org/detail. cfm?id=2209336.

8. Paxson, V., Allman, M., Chu, J., Sargent, M. 2011. Computing TCP's retransmission timer. RFC 6298.
9. Postel. J. 1980. User Datagram Protocol. RFC 768.
10. Schulzrinne, H., Casner, S. Frederick, R., Jacobson, V. 2003. RTP: a transport protocol for realtime applications. RFC 3550.
11. Stewart, R. 2007. Stream Control Transmission Protocol. RFC 4960.
12. Stewart, R., Ramalho, M., Xie, Q., Tuexen, M., Conrad, P. 2004. Stream Control Transmission Protocol (SCTP) partial reliability extension. RFC 3758.
13. Zhang, L. 1986. Why TCP timers don't work well. In *Proceedings of SIGCOMM* (August).

STEPHEN STROWES is a senior engineer at Boundary Inc., where he works on network measurement metrics and tools. He has previously studied scalable inter-domain routing protocols, NAT traversal protocols, and peer-to-peer protocols for realtime content.