Inferring TCP Connection Characteristics Through Passive Measurements

S. Jaiswal, G. Iannaccone, C. Diot, J. Kurose, & D. Towsley

CMPSCI TR 04-10

Inferring TCP Connection Characteristics Through Passive Measurements

Sharad Jaiswal†, Gianluca Iannaccone§, Christophe Diot§, Jim Kurose†, Don Towsley†

†Computer Science Department
Univ. of Massachusetts, Amherst
{sharad,kurose,towsley}@cs.umass.edu

§Intel Research
Cambridge, UK
{gianluca.iannaccone,christophe.diot}@intel.com

Abstract— **We propose a passive measurement methodology to infer and keep track of the values of two important variables associated with a TCP connection: the sender's congestion window (cwnd) and the connection round trip time (RTT). Together, these variables provide a valuable diagnostic of end-user-perceived network performance. Our methodology is validated via both simulation and concurrent active measurements, and is shown to be able to handle various flavors of TCP. Given our passive approach and measurement points within a Tier-1 network provider, we are able to analyze more than 10 million connections, with senders located in more than 45% of the autonomous systems in today's Internet. Our results indicate that sender throughput is frequently limited by a lack of data to send, that the TCP congestion control flavor often has minimal impact on throughput, and that the vast majority of connections do not experience significant variations in RTT during their lifetime.**

Index Terms— **Network Measurements, Traffic analysis, TCP**

I. Introduction

TCP (Transmission Control Protocol) is the dominant end-to-end transport protocol currently deployed in the Internet, with a wide range of applications such as Web traffic, grid applications, and newly emerging peer-to-peer applications relying on TCP's transport services. Given this reliance on TCP, there is currently great interest in understanding TCP's performance and characterizing the factors (such as network congestion, sender/receiver buffer limits, and sender data-starvation) that can limit its behavior in practice.

In this paper we present a passive measurement methodology that observes the sender-to-receiver and receiver-to-sender segments in a TCP connection, and infers/tracks the time evolution of two critical sender variables: the sender's congestion window (*cwnd*) and the connection round trip time (*RTT*). As we will see, with knowledge of these two values, many important characteristics of the sender, receiver, and the network path that connects them can be determined. For example, by comparing *cwnd* with the amount of data actually sent, one can determine when a TCP connection is starved for application-level data (i.e., that the connection could support a higher transfer rate, if more data were available); by carefully observing the manner in which *cwnd* changes in response to loss, one can identify non-conformant TCP senders, or the particular conformant flavor of TCP (e.g., Tahoe, Reno, New Reno); by monitoring RTTs, one can characterize RTT variability within and among flows, and determine the extent to which application-level adaptivity is needed to cope with variable network delays.

Our work makes several important contributions. Our first contribution is methodological. We develop a passive methodology to infer a sender's congestion window by observing TCP segments passing through a measurement point. The measurement point itself can be anywhere between the sender and the receiver. We only require that packets can be observed from both directions of the TCP connection, a requirement our previous work [11] has shown to not be overly restrictive. In case the connection experiences losses, our methodology's estimate of *cwnd* is sensitive to the TCP congestion control flavor (Tahoe, Reno, or New Reno) that best matches the sender's observed behavior. We also propose a simple RTT estimation technique based on the estimated value of *cwnd*.

Our second contribution is in terms of the measurements made, and the application of our methodology to the traces gathered within the Sprint IP backbone. We present results on the distributions of congestion window sizes and RTTs in the observed TCP connections. Our

This work is supported by the National Science Foundation under grants EIA-0080119, ITR-0522631, and ANI- 0240487, and a gift from Sprint Corporation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. Part of this work was carried out when S. Jaiswal, G. Iannaccone and C. Diot were at Sprint ATL, Burlingame, CA.

study is unique in that it examines a remarkably large and diverse number of TCP connections. Given our passive methodology and measurement points within a Tier-1 network provider, we are able to analyze more than 10 million connections, with senders located in more than 45% of the autonomous systems in today's Internet. We find that sender throughput is frequently limited by lack of data to send, i.e., that lack of data, rather than network congestion, is often a limiting factor. We find that the majority of TCP connections reach a maximum congestion window on the order of 10 segments but that 50 to 60% of the packets belong to connections with windows larger than 10 segments. We find that connections do not generally experience large RTT variations in their lifetime. For example, for approximately 80-85% of the connections, the ratio between the 95th percentile RTT value and the 5th percentile RTT value is less than 3; in absolute terms, the RTT variation during a connection's lifetime is less than 1 second for 75-80% of the connections. Finally, we find that TCP congestion control flavors generally have a minimal impact on the sender's throughput; the vast majority of the connections would achieve the same throughput independently of the congestion control flavor implemented.

The remainder of this paper is organized as follows. Section II discusses related work. In Sections III and IV we present our methodology to keep track of the sender's congestion window, to infer the TCP sender's flavor, and to compute the RTT. In Section V we identify events that can introduce uncertainty into our estimates and we define bounds on this uncertainty. Section VI describes the results of the evaluation of our methodology via simulations and real-world experiments. Section VII presents our observations derived from the analysis of packet traces collected in various points in the core of the Sprint IP backbone. Finally, Section VIII summarizes our contributions and provides directions for the extension of this work.

II. Related work

Numerous measurement studies have investigated the characteristics of TCP connections in the Internet. Many of the early studies [17] either actively measured end-to-end properties (e.g., loss, delay, throughput) of TCP connections, or passively characterized a connection's aggregate properties (e.g., size, duration, throughput) [20].

More recently, researchers have focused their attention on specific details of TCP implementations. [15] develops a tool to characterize the TCP behavior of remote web servers. Their approach involves actively sending requests to web servers and dropping strategically chosen response packets in order to observe the server's response to loss. They observe the prevalence of various TCP implementations, the presence of TCP options such as SACK and ECN, and identify conformant/non-conformant congestion control behaviors. Our approach, by contrast, is passive and is thus able to easily characterize large numbers of TCP connections.

In [16], the author had the ability to run *tcpdump* over a set of end hosts. The paper describes a tool, *tcpanaly,* to analyze these traces, and reports on the differences in behavior of 8 different TCP implementations. Methodologically, our work is alike, in the sense that both involve passive observation of the behavior of TCP connection and the use of heuristics to decide which flavor or implementation of TCP best matches the connection being observed. However, the scope of [16] is focused on highlighting the differences between various TCP implementation stacks. Since our measurement point is located in the middle of the end-end path, it is not possible for us to distinguish if a particular sender behavior is due to events in the network or end-system TCP implementation issues. Moreover, since we track several millions of highly diverse TCP connections, we do not concern ourselves with implementation-level details of the senders. Our main goal is to track the sender's congestion window. We only seek to detect the cases in which our estimate of the congestion window may be different from that of the sender; we do not perform a detailed case-by-case analysis of the reasons for this difference. Also, in [16], the analyzed traces involve bulk file transfers, hence the author did not have to take into account effects of sender and application behavior. This aspect is discussed in some detail in our work. The location of the observation point also introduces methodological challenges in estimating a connection's RTT (in contrast to measurements taken at the end hosts) and we propose a technique to address this issue in this work. Finally, our study is much larger in scale and more diverse.

Another work of interest is [11], which presents a methodology to classify out-of-sequence packets,with the same measurement environment as in our current work. As discussed in section IV, we use the methodology in [11] to identify packet retransmissions in our traces.

[21] is the work that is perhaps most closely related to this present work. In [21], the authors passively monitor TCP connections and develop heuristics that are used to classify connections according to the factor(s) that limit their throughput. A technique is also proposed to estimate RTT by selecting a value (from among a set

of predetermined values) that most closely matches the observed packet flight dynamics. Our work differs from [21] in several important respects. Most importantly, our goal is not to study the rate-limiting factors of TCP, but more fundamentally to develop a methodology for estimating *cwnd* and *RTT*. These are arguably *the* two most important pieces of TCP sender state. As we will see, knowledge of these values will allow us to study many characteristics of TCP connections. These values can be used to determine the factors that limit a TCP's throughput. These values can also be used to detect non-conforming TCP senders, and to determine the extent to which various "flavors" of TCP are used in practice. These values can also be used to determine how often a newer version of TCP is able to exercise its enhanced capabilities (e.g., how often NewReno's fast recovery mechanism is actually used in practice). In cases where our work overlaps with [21] (e.g., in determining the factors that limit a TCP connection's throughput), a direct comparison is not currently possible, as the tools in [21] have not yet been released. We conjecture, however, that since the techniques in [21] and in this present work are quite complementary, their combined use will allow for even better classification of TCP behaviors than either tool alone.

Several recent efforts have considered the problem of estimating the RTT of a connection using passive measurements [12], [13]. These works compute one RTT sample per TCP connection, either during the triple-handshake or during the slow-start phase. Our works extends these efforts in that it computes RTT estimates throughout a connection's lifetime.

III. Tracking the Congestion Window

In this section we describe our methodology to keep track of a sender's congestion window, *cwnd*. The congestion window represents the maximum amount of data a sender can potentially transmit at any given point in time.

The basic idea is to construct a "replica" of the TCP sender's state for each TCP connection observed at the measurement point. The replica takes the form of a finite state machine (FSM). The replica FSM updates its current estimate of the sender's *cwnd* based on observed receiver-to-sender ACKs, which (if received at the sender) would cause the sender to change state. Transitions are also caused by detecting a timeout event at the sender. These timeouts are manifested in the form of out-of-sequence sender-to-receiver retransmissions, which are detected using the passive measurement techniques from [11]. The FSM implementation is described in more detail in [10].

Estimating the state of a distant sender poses many challenges:

- In order to process large amounts of data (i.e., hundreds of GBytes), a replica can only perform limited processing and maintain minimal state. Our replica thus works in a "streaming" fashion; it can neither backtrack nor reverse previous state transitions.
- Given its position in the "middle" of an end-to-end path, a replica may not observe the same sequence of packets as the sender. ACKs observed at the measurement point may not reach the sender. Additionally, packets sent from the sender may be reordered or duplicated on the sender-to-measurement-point path. Here, we use use techniques from [11] to identify and classify such out-of-sequence packets.
- The manner in which *cwnd* is modified after packet loss is dictated by the flavor of the sender's congestion control algorithm. We consider the 3 major flavors of TCP congestion control - Tahoe, Reno and NewReno[1] - and instantiate three different FSMs, one for each flavor.
- Implementation details of the TCP sender, as well as the use of TCP options, are invisible to the replica[2].

All of the considerations above introduce uncertainties into *cwnd* estimation, which we discuss in more detail in Section V.

Several variables must be initialized in the replica FSM. The sender's initial congestion window size, *icwnd*, is the maximum number of bytes that a sender can transmit after completing the triple-handshake and before any ACKs arrive. The typical value of *icwnd* can be up to twice the maximum segment size [3]. An experimental TCP specification allows *icwnd* to be as high as twice again this value [2]. We estimate *icwnd* by keeping a count of the number of data packets observed before seeing a receiver ACK. We also initialize the slow-start threshold (*ssthresh*) to an extremely large value, as it is commonly done in TCP stacks [3].[3].

During normal operations, a TCP sender can either be in slow-start or congestion avoidance. The arrival of a new ACK increases *cwnd* by 1 or by $1/cwnd$, respectively. If the sender detects a loss

[1]There exist other implementations such as TCP Vegas [4] and TCP Westwood [5], but we are not aware of any widely used OS stacks which implement these algorithms, hence we drop these from our study.

[2]Sprint's IPMON traces [7] only contain the first 44 bytes of all packets. Thus, for TCP packets we have only access to the first 4 bytes of the payload.

[3]In some TCP implementations the sender initializes the value of *ssthresh* from its *route cache*, an issue we discuss in Section V.

via timeout, it sets *cwnd* to 1 and *ssthresh* to $max(min(awnd, cwnd)/2, 2)$, where *awnd* is the receiver advertised window. The more interesting case is when packet loss is detected via the receipt of three duplicate ACKs, one event that brings out the differences between the three flavors under consideration:

Tahoe. A Tahoe sender reacts to the receipt of three duplicate ACKs with a so-called *fast retransmit,* behaving exactly as if the retransmission timeout had expired.

Reno. Reno TCP adds *fast recovery* to Tahoe's fast retransmit algorithm [3]. Fast recovery works on the assumption that each duplicate ACK is an indication that another packet has successfully reached the receiver. The sender adjusts its *cwnd* to account for this fact: *ssthresh* is set to $max(min(awnd, cwnd)/2, 2)$[4] and *cwnd* is set to $ssthresh + 3$. Thereafter, the sender increments the *cwnd* by 1 for every new duplicate ACK received. Once the sender receives a new ACK, it resets the value of *cwnd* to *ssthresh*, and exits fast recovery, returning to congestion avoidance.

NewReno. NewReno introduces a simple change in Reno's fast recovery mechanism [6] by removing the need to detect a loss through timeout when multiple losses occur within a single congestion window. The change occurs when the sender receives a new ACK while in the recovery phase. In NewReno the sender checks whether this is a *partial* new ACK, i.e., it does not acknowledge all packets sent before fast retransmit. If the ACK is partial, the sender immediately retransmits the packet requested by the receiver and remains in fast recovery. This behavior ensures that the sender is able to retransmit a lost packet after every RTT without the timeout mechanism. The NewReno sender remains in this phase until it receives an ACK that acknowledges all outstanding packets before the recovery phase, at which point it returns to congestion avoidance.

A. TCP flavor identification

As noted above, the three flavors of TCP can respond differently to loss events. In order to determine which flavor of TCP is implemented by a sender, we exploit the fact that a TCP sender can never have more outstanding unacknowledged ("in flight") packets than its usable window size. A sender's usable window size is the smaller of *cwnd* and the window advertised by the receiver. This forms the basis for our test to identify the sender's flavor. For every data packet sent by the sender, we check whether this packet is allowed by the current FSM estimate of *cwnd* for each particular flavor. Given a flavor, if the packet is not allowed, then the observed data packet represents a "violation" - an event that is not allowed. We maintain a count of the number of such violations incurred by each of the candidate flavors. The sender's flavor is inferred to be that flavor with the minimum number of violations, and, at any time during the life of a connection, the sender's congestion window is the value of *cwnd* as estimated for this flavor. If no violations are observed for any TCP flavors, we say that the flavors are indistinguishable. In section VII.B we quantify the extent to which various flavors of TCP are observed in our traces.

B. Use of SACK and ECN

TCP sender behavior also depends on two options whose deployment is reported to be growing fast [15]: Selective Acknowledgments (SACK) [14] and Explicit Congestion Notification (ECN) [18].

The TCP SACK option allows for the recovery from multiple losses in a single congestion window without timeout. SACK, by itself, does not change the congestion window dynamics of the sender, i.e. it only helps in deciding what to send, not when to send it. Our measurement points do not have access to SACK blocks. In some cases, it is possible to detect the presence of SACK blocks and/or infer the use of SACK information during fast recovery. Detecting and using SACK information is part of our ongoing work, and is not considered further in this paper.

An ECN-capable sender explicitly notifies the receiver of every reduction of the congestion window for any reason (fast retransmit, retransmission timeout or ECN-echo flagged acknowledgment). The measurement point could estimate the congestion window of the sender just by looking at the ECN bits in the TCP header. Unfortunately, ECN is still not widely deployed by end-hosts. In the packet traces we studied, only 0.14% of the connections were ECN-aware.

IV. Round-trip time estimation

In this section we describe a technique to compute RTT samples throughout the lifetime of a TCP connection, leveraging the *cwnd* estimation techniques from the previous section. The implementation of this method is simple and in some cases results in as many RTT samples as would be computed by the actual TCP sender. We only provide a brief overview of our technique here; the reader is referred to [11] for a more detailed description.

[4]RFC 2581 instructs that, after a loss, *ssthresh* should be set to $max(flightsize/2, 2)$, where *flightsize* is the number of packets currently unacknowledged. We choose $min(awnd, cwnd)$ instead of *flightsize* to follow the current implement of TCP in Linux and FreeBSD.

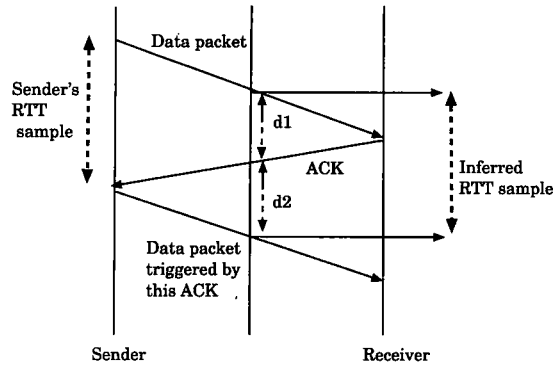

Fig. 1. TCP running sample based RTT estimation

The basic idea behind our RTT estimation technique is illustrated in Figure 1. Since we are not able to directly measure the sender's RTT sample shown in the left of Figure 1, we instead measure *(i)* the round trip delay from the measurement point to the receiver and then back to the measurement point (labeled $d1$ in the figure), and *(ii)*, the round trip delay between the measurement point, the sender and then back to the measurement point (labeled $d2$ in the figure). The sum of these two delays $d1 + d2$, as shown in Figure 1, is our estimate of the RTT. We refer to our method as a *running RTT* estimation technique, since it continuously makes RTT estimates, based on the measured values of $d1$ and $d2$ throughout the TCP connection's lifetime. In the case that the transmission time of the two data packets in Figure 1 is exactly the same, our RTT estimate will be exact. We will investigate the magnitude of the RTT estimation error shortly. We conclude here by mentioning two important aspects of the running RTT estimation technique.

An important requirement of the *running RTT* estimation technique is the ability to determine which data packet transmissions are triggered by the arrival of a particular ACK. This requires an accurate estimate of *cwnd* and it is here that our techniques from the previous section come into play.

Our technique must also be able to stop (and restart) the RTT estimation as a sender recovers from a loss in order to closely emulate the behavior of the actual TCP sender (which does not compute the RTT during the loss recovery). In order to do this, our technique relies on the knowledge of the state of the TCP connection, i.e., if the sender is in fast recovery.

V. Sources of Estimation Uncertainty

The idea behind replicating sender state, and tracking various TCP flavors is not complicated. However, the measurement point has only partial information about the TCP connections it observes. Given its location in the middle of the path, it may not observe the same events as the senders, and vice versa. Moreover, it assumes complete knowledge of the TCP stack implementations that may instead present subtle (or malicious) differences [15], [16]. These two characteristics of our measurement methodology can introduce uncertainties into our estimate of sender state. A significant aspect of our work is to understand how these issues impact our approach (or any passive measurement approach, in the middle of the end-end path). In order to address these issues, we start by identifying events in the network that can result in an ambiguous or erroneous state at the measurement point. Then, we discuss how our methodology can detect such events, and provide a quantitative upper bound on their occurrence.

A. Under-estimation of cwnd

The measurement point may underestimate *cwnd* if it observes three duplicate ACKs that never reach the sender. According to our methodology, the measurement point would infer that the sender will undergo a fast retransmit and reduce *cwnd* and *ssthresh* accordingly. The sender will eventually timeout and then retransmit the packet. At this point the measurement point will detect the timeout and reduce the the value of *ssthresh* twice (once for the fast retransmit and then as part of the timeout) while the sender would do so only once.

Note that even if the measurement point detects the timeout, it cannot later reverse its decision. Indeed, the measurement point would observe the same sequence of packets if the third duplicate ACK is lost (*ssthresh* is modified only once) or if the packet sent due to fast retransmit is lost (*ssthresh* is modified twice).

On the other hand, detection of these cases is relatively simple. A sender, if greedy, will transmit more packets than the estimated *cwnd* would permit. That sender would then trigger violations in all the TCP flavor FSMs. This way, the measurement point can identify the connections for which the congestion window is uncertain.

In order to quantify the frequency of these events, we counted the number of senders that incur in violations in all the flavors in the packet traces we study. Only 0.01% of all senders show this behavior. If we focus on senders with more than 13 packets to send[5] this percentage goes up to 5%. This is expected given that the more packets a

[5]Senders with at least 13 packets may experience a fast retransmit. In fact, a sender with an initial window of 2 packets and in presence of delayed ACKs would send 13 packets in 4 flights of 2, 3, 3, and 5 packets, respectively.

sender transmits, higher the likelihood that it will incur the loss scenario described above.

B. Over-estimation of cwnd

Two events will lead to an over estimate of the congestion window size.

Acknowledgments lost after the measurement point. Every ACK for new data observed at the measurement point causes an increment of *cwnd*. If the ACK is lost before reaching the sender, there will not be a corresponding increment at the sender, resulting in an over-estimation of the sender's congestion window. Moreover, during fast recovery, any extra duplicate ACKs observed at the measurement point cause *cwnd* to be increased by one. Again, if such ACKs are lost before they reach the sender, the measurement point will over-estimate the sender's *cwnd*.

Entire window of data packets lost before the measurement point. An entire window of packets transmitted by the sender and dropped before the measurement point will remain undetected. This is because, in order to detect a loss, the measurement point needs either to observe packets from the receiver related to the packet drops (in the form of duplicate ACKs) or earlier transmissions of the data packets. In this case the sender will timeout and update the values of *ssthresh* and *cwnd* while the measurement point will maintain the old values. The loss of the initial SYN packet is a special case of this event that will make the measurement point over-estimate *ssthresh* (set by the sender to 2) and thus incorrectly consider the sender to be in slow-start for longer than needed.

The detection of overestimation events is particularly difficult. In fact, a sender for which the measurement point has a larger estimate of *cwnd* would just appear as a "not-greedy" sender, i.e., with a rate limited by the application or by the kernel sending buffer size. Not-greedy senders are not uncommon: Zhang et al. [21] estimate that application-limited senders account for up to 34% of all senders. Our traces tend to confirm this estimate, with around 30% of the senders not-greedy (see Section VII). It is important to remember that these numbers represent very loose upper bounds on the magnitude of this type of estimation uncertainty.

C. Window scaling

The sender window is the minimum of *cwnd* and the receiver advertised window, *awnd*. Since, we collect only the first 44 bytes of the packets and thus can not track the advertised window for all connections. Indeed, for those connections that use the window scale option [9], capturing only the first 44 bytes hides the scale factor making it impossible to know the exact value of the receiver window.

In order to estimate how often this problem occurs we first count the the number of connections that could be using the window scale option; then, we count the connections for which *cwnd* could exceed *awnd*.

The standard [9] requires end-hosts to negotiate the use of the window scaling option during the initial connection setup. Therefore, the size of the SYN and SYN+ACK packet can be used to infer if those packets could accommodate the window scale option (that consumes 3 bytes).

For these connections, given that the window scale option can only "scale up" the window size, we have at least a lower bound on the size of the receiver window. Therefore, we can identify all the connections for which we are uncertain of the sender window as those where *cwnd* reaches the lower bound of *awnd*.

In the packet traces under study, our measurement point is uncertain of the sender window for around 2% of the connections. Note that these numbers refer to the case where the window size is uncertain at least once during the lifetime of the connection. In general, our methodology allows us to track correctly *cwnd* as long as it is below the lower bound of *awnd*. When *cwnd* exceeds the lower bound of *awnd* we may underestimate *cwnd*, as discussed earlier.

D. Issues with TCP implementations

Several previous works [15], [16] have uncovered bugs in the TCP implementations of various OS stacks. For example, using the TBIT tool, the authors in [15] found that as many as 8% of web servers probed used a version of Reno which would behave more aggressively then Reno during the recovery phase, and about 3% of web servers simply did not cut down their window after a loss.

Another issue that we mentioned in Section III is regarding the initialization of the senders *ssthresh* value. Some TCP implementations cache the value of the sender's *cwnd* just before a connection to a particular destination IP-address terminates, and reuse this value to initialize *ssthresh* for subsequent connections to this destination. Hence, in such cases a TCP sender can start out with a smaller value of *ssthresh*, which might result in it exiting slow-start earlier than predicted by the FSMs.

These implementation issues affect our ability to track the congestion window and RTT of a TCP sender. It is impractical to detect and track all possible TCP

implementations. On the other hand, we can use the same techniques described above to identify inaccuracies in the *cwnd* estimate to discover TCP senders with non-compliant congestion control or differences in implementation, and interrupt our RTT and *cwnd* estimation. For example, a sender that does not react to the receipt of three duplicate ACKs will later violate all TCP flavors and thus be discovered by our measurement point. Also, a sender with a smaller initial value of *ssthresh* which exits slow-start earlier than predicted by the FSM could appear as a non-greedy source, prompting us to stop computing RTT estimates.

E. Impact on RTT estimation

RTT estimation is directly affected by estimation inaccuracies in *cwnd*. Our methodology needs to know *cwnd* in order to identify the data packet whose transmission by the sender is triggered by the receipt of the ACK used in the estimation (see Section IV). Therefore, over-(under-) estimation of *cwnd* will result in an over-(under-) estimation of the RTT. For this reason when the measurement point detects estimation uncertainty in *cwnd*, it interrupts RTT estimation for that connection.

VI. Evaluation

We validated our methodology using both simulations and experiments over the Internet.

A. Simulations

In our simulations we used the *ns* simulator, with a typical "dumbbell" topology with a single bottleneck link. We generated long lived flows for analysis and cross traffic consisting of 5,700 short lived flows (40 packets) with arrival times uniformly distributed through the length of the simulation. We did sets of experiments with the bottleneck link located between the sender and the measurement node, and with the bottleneck after the measurement point. We also ran simulations with different parameters for the bottleneck link, varying the bandwidth, buffer size and propagation delay. The average loss rate in the various scenarios varied from 2% to 4%.

For each sender we compare the RTT samples measured with our methodology with the values computed by the *ns* sender. Estimated *cwnd* values are compared to those maintained at the sender every time the sender's window changes. Given a series of estimated and observed values, we compute the average of the relative errors for each sender. Figure 2 plots the cumulative distribution of the mean relative errors for RTT and *cwnd*. The estimated error for *cwnd* is shown to be less than 5% for more than 95% of the senders. Only

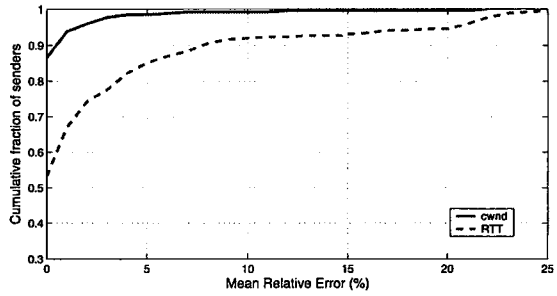

Fig. 2. Mean relative error of cwnd and RTT estimates in the simulations

1 sender out of the 280 under study incurred an error larger than 20%. In absolute terms, 265 senders have an average *cwnd* of less than 0.5 packets, while only 1 had an error of more than 1 packet. The RTT estimate error is less than 10% for 90% of the senders, and never exceeds 25% of the actual value.

Simulations can also give us insight into how accurately our methodology identifies the TCP flavor. Out of the 280 senders, the TCP flavor of 271 senders was identified correctly. Of the remaining senders, 4 either had zero violations for all flavors (i.e., they did not suffer a specific loss scenario that allows us to distinguish among the flavors) or had an equal number of violations in more than one flavor (including the correct one). Also, we misclassified 5 connections. This can happen if the FSM corresponding to the TCP sender's flavor understimates the sender's congestion window, for example, as in the scenario described in Section V-A. Thus, this FSM may experience more violations in its estimate of the sender's cwnd and this would lead to an incorrect identification of the sender's flavor.

We also observed that by increasing the duration of the simulations, we were able to increase the number of connections for which we correctly identified the flavor. This is as expected, since the more packets a sender transmits, the higher the probability that the sender would exhibit the behavior peculiar to its flavor.

B. Experiments over the network

We also validated our inference techniques in an experimental testbed. Our setup consists of PCs running the FreeBSD 4.3 and 4.7 operating systems, with a modified kernel that exports the connection variables to user space using the `sysctl` facility. The PCs were located at the University of Massachusetts, in Amherst, MA and Sprint ATL, in Burlingame, CA. Traffic between these two sites passed through an OC-3 access link in Stockton, CA, which was also passively monitored by an IPMON system. We carried out our experiments by

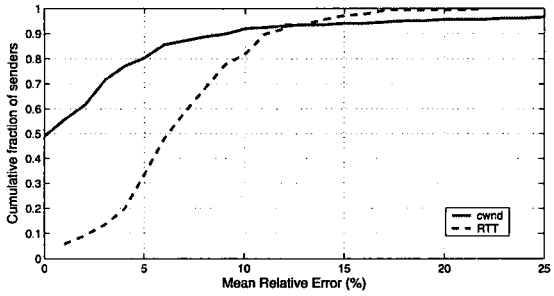

Fig. 3. Mean relative error of *cwnd* and RTT estimates with losses induced by dummynet

setting up 200 TCP connections (divided between Reno and NewReno flavors).

We ran a set of experiments at different times of the day, over several days, with bulk transfers lasting between 10 seconds and 5 minutes. Given the very low loss rates (rarely exceeding 0.2%) experienced by the connections our estimate were very accurate. The error in *cwnd* was on the order of the rounding error when converting *cwnd* from bytes to packets.

Given the low error rates observed in the native Amherst-to-Burlingame connection, we ran a second set of experiments using a `dummynet` bridge in front of the TCP senders [19] to emulate a bottleneck link (with loss rates in 3-5% range). Figure 3 plots the mean relative error for *cwnd* and RTT. The error in the *cwnd* estimate is less than 15% for almost 95% of the senders. In absolute terms, 182 out of the 200 senders exhibit an error of less than 0.5 packets. This level of error is likely due to rounding errors between the measurement point and the TCP stacks in the kernels under study. The RTT errors are also very small: 95% of the senders have an error of less than 15%. We note that RTT errors below 10% are to be expected. The average RTT between UMass and Sprint ATL is around 90ms, thus a 10% error is on the order of the kernel clock tick in the testbed machines (set to 10ms).

In summary, our simulations and live Internet experiments have demonstrated the accuracy of our methodology for tracking *cwnd* and *RTT*. The relative errors were found to be small, often on the order of (and possibly due to) rounding errors or clock tick precision.

VII. Backbone Measurements

The IPMON infrastructure provides packet-level traces from OC-3/12/48/192 links in several Points-of-Presence (POPs) in the Sprint backbone[6]. In the following, we show results from three data sets: i) a bidirectional OC-48 (2.5 Gbps) link on the East Coast of the U.S. (named *East Coast*); ii) two bidirectional OC-48 links connecting San Jose, CA to Relay, VA (*Transcontinental*); iii) one bidirectional OC-48 link inside the New York PoP (*Intra-POP*).

Table I presents a summary of the characteristics of the data sets. All the data sets are 1 hour long and contain a total of approximately 11M TCP connections. We believe these traces provide a representative sample of the Internet traffic. For example, we retrieved the BGP table from Sprint backbone routers during trace collection, and used the AS path information to derive the source and destination ASes of the observed TCP connections. Overall, the data sets contain TCP connections originating in 5,819 unique ASes. This represents around 45.3% of the total number of allocated ASes at the time of the trace collection (we counted 12,848 ASes as of November 21st, 2002).

A. Congestion window

A first metric of interest is the maximum window that a TCP sender reaches during its connection lifetime. As mentioned earlier, the maximum window size is limited by several factors, including the receiver's advertised window[7]. Figure 5 shows the empirical cumulative distribution function of the maximum window size during the lifetime of the connections. The five curves correspond to different thresholds for the size of the flows.[8] The curve with the minimum threshold, 5 data packets, has the steepest slope since it includes many small connections that do not have the opportunity to expand their window sizes. The median value of the maximum window for these flows is 8, while 80% of the flows have a maximum window size of less than 11.

We observe that as the flow size threshold increases, the distribution of the curves become more similar, with a median maximum window size of approximately 10 to 12 packets. This could be because as the connections become larger, they have a higher probability of either incurring a loss, or being restricted by the receiver-advertised window. Additionally, the distributions show a set of spikes at values corresponding to maximum window sizes of 6, 12, or 16 packets. These values correspond to connections that are restricted by commonly-used advertised window values of 8 and 16 Kbytes, with an MSS of either 536 or 1460 bytes.

[6]For more details see http://ipmon.sprint.com

[7]As discussed in Section V, we remove senders for which we are unsure of the receiver window from this analysis

[8]We have combined data from all the three traces for this analysis, since the distributions of window size were similar across the traces. Please refer to [10] for a per-trace analysis.

	East Coast	Intra-POP	Transcontinental
Link Speed	OC-48 (2.5Gbps)	OC-48 (2.5Gbps)	OC-48 (2.5Gbps)
No. of links	1	1	2
Unique source ASes	1,565	4,945	2,326
Unique network prefi xes	7,359	25,133	11,218
TCP connections	844K	6M	4.9M
Percentage of all TCP connx	18.76%	51.98%	38.76%
TCP Data packets	18M	74M	110M

TABLE I

SUMMARY OF THE TRACES

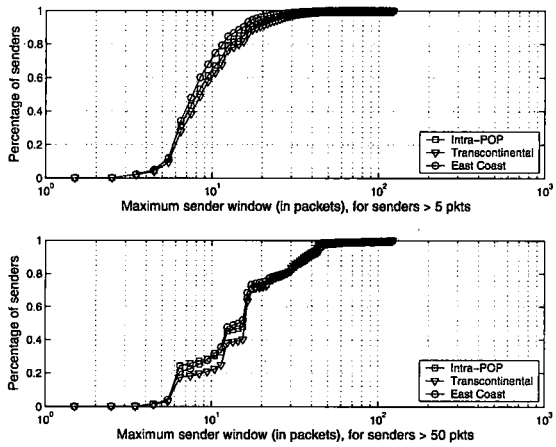

Fig. 4. Cumulative fraction of senders as a function of the maximum window

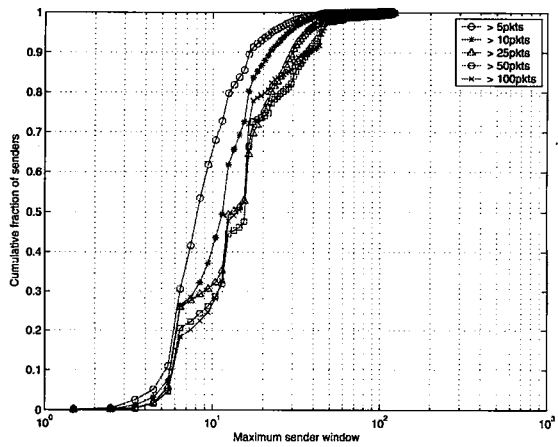

Fig. 5. Cumulative fraction of senders as a function of the maximum window

Although most of the senders have a relatively small maximum window size, most of the packets belong to connections with large window sizes. Figure 6 plots the cumulative distribution of packets that belong to a connection with a given maximum window size. Looking at the curve for flows with at least 5 data packets, we observe that although 80% of such flows have a maximum window size of less than 11 data packets, they carry only 45% of the packets. We also observe similar spikes in the distribution of Figure 6 as in the previous figure, again corresponding to commonly used values for receiver-advertised windows.

Overall, the throughput of 4% of the connections is limited by the receiver-advertised window at some point in their lifetime. These connections account for about 40% of the packets. If we look at connections with at least 10, 25 and 50 data packets, we observe that 44%, 62% and 72% of such connections, respectively, are limited by the receiver window.

B. TCP flavors

As described in Section III, congestion window evolution can depend on the flavor of TCP congestion control. Table II gives the number of connections and packets in the three data sets that conform to a specific flavor. The flavor of a particular TCP sender is defined to be that flavor whose estimated *cwnd* values resulted in the minimum number of observed violations, i.e., packets sent that exceeded the estimated available window. Senders with no window violations are classified as "indistinguishable", indicating that the specific flavor does not affect the connection throughput. In case of parity between TCP Reno and NewReno violations, we count the sender as a Reno sender.

In Table II we consider only those senders that transmit more than 13 data packets. This way we only consider connections that have the opportunity to experience a fast retransmit. These senders account for about 8% of all senders but contribute to almost 78% of data packets. Interestingly, the behavior of 97.05% of these senders does not allow us to distinguish a particular flavor of congestion control. This is because the congestion control flavors manifest differences only in the way in which they respond to the receipt of three duplicate acknowledgments. In our traces, this event is

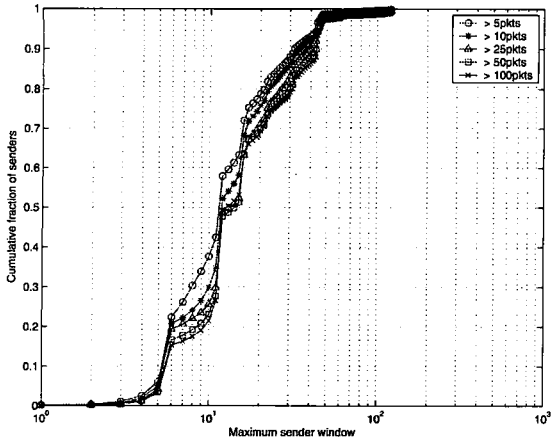

Fig. 6. Cumulative fraction of packets as a function of the sender's maximum window

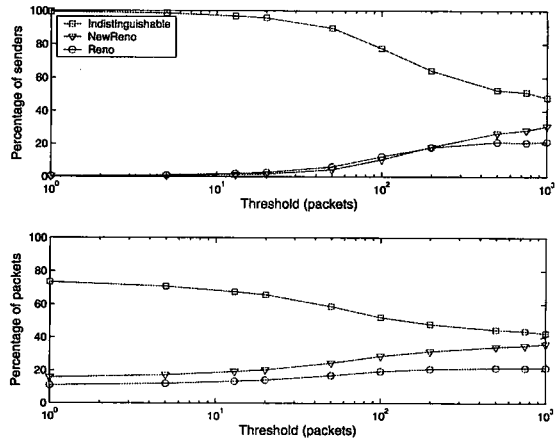

Fig. 7. Percentage of Reno/NewReno senders (above) and packets (below) as a function of the data packets to transmit

	TCP Senders	Data Packets
Senders $\geq$13 pkts	1,56M	139M
Indistinguishable	1,51M (97.05%)	94M (67.51%)
Tahoe	640 (0.04%)	340K (0.17%)
Reno	28K (1.84%)	18.5M (13.33%)
NewReno	17K (1.10%)	26M (19.17%)

TABLE II
TCP FLAVORS

relatively rare: only 5% of the senders experience a triple duplicate ACK. Moreover, even after a fast retransmit the three flavors will exhibit different behavior only in presence of specific loss patterns. Overall, only 2.95% of the senders have the opportunity to make use of this flavor-manifesting behavior over their lifetimes.

Table II also shows the number of packets that belong to connections of different flavors. Using this metric, the number of packets that belong to the "indistinguishable" category drops to 67.5%. This is expected, given that large connections have a higher probability of experiencing losses that allow us to classify the flow as a specific flavor of TCP.

Also, the NewReno connections seem to carry a disproportionately large number of packets when compared to Reno senders. But this is expected as well, given that the difference between Reno and NewReno manifests itself only in presence of multiple packet losses in the same window of data. The likelihood of such event happening is higher for long connections with large windows.

In order to confirm our conjectures we plot in Figure 7 the percentage of senders and packets classified as Reno or NewReno as a function of the number of data packets sent. As expected, the number of senders that fall in the "indistinguishable" category decreases as the number of packets sent increases. Also, as the number of data packets sent increases, NewReno appears to be the more prominant TCP flavor both in terms of senders and packets.

C. Greedy senders

A sender is defined as "greedy" if at all times the number of unacknowledged packets in the network equals the the available window size. Under normal conditions, the measurement point would observe a full window of data being sent, a set of acknowledgments being returned, a new (larger) full window of data packets being sent, and so on. In this model of sender behavior, a "flight" is a set of packets sent "together" separated from subsequent flights by the ACKs from the receiver.

Using our estimate of the window available to the sender, and this notion of flights of packets, we propose a simple heuristic to identify not-greedy senders: if the number of unacknowledged packets (also defined as *flight-size*), at the end of a flight, is less than the available window at the beginning of a flight, then the sender is not greedy. The measurement point identifies the beginning of a new flight (and the end of the current flight) as the first data packet observed after an ACK that acknowledged a data packet in the current flight[9].

The basic assumption of this heuristic is that a set of packets sent in a "flight" by the sender is never interleaved with acknowledgments for packets in that flight, i.e. that all acknowledgments that arrive in the "middle" of a flight cover data packets belonging to previous flights. The validity of this assumption depends

[9]In the presence of packet losses, the measurement point needs to wait for the end of the recovery phase.

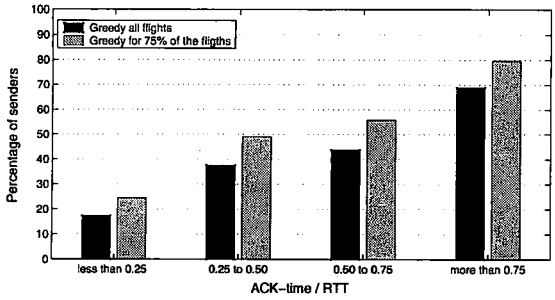

Fig. 8. Fraction of greedy senders based on the distance between measurement point and receiver

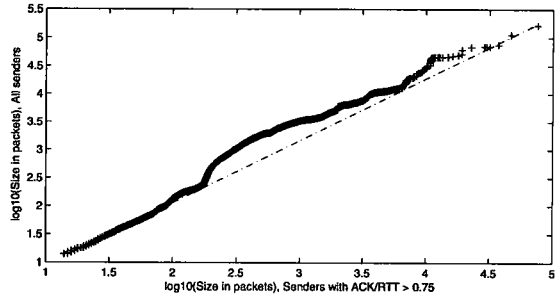

Fig. 9. qq-plot of flow size between flows with large ACK times, and all flows

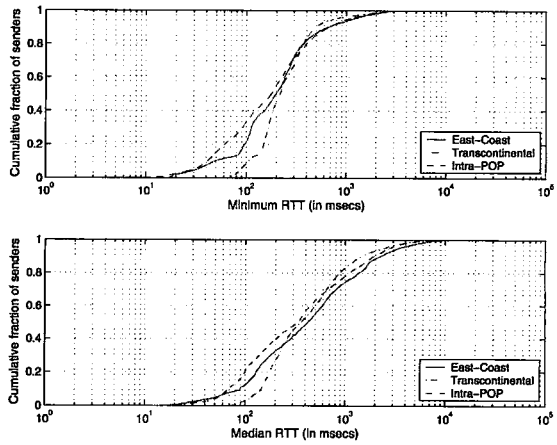

Fig. 10. Top: CDF of minimum RTT; Bottom: CDF of median RTT

on the position of the measurement point with respect to the sender and receiver. Indeed, if the measurement point is very close to the receiver, each data packet will be immediately acknowledged (or every other data packet in presence of delayed acknowledgments). Hence, the measurement point will estimate a flight size of one or, at most, two packets and deem the sender as not-greedy. In order to identify those connections for which the measurement point is "too close" to the receiver, we define the "ACK-time" as the time between the observation of a data packet and the observation of the acknowledgment that covers that data packet. In our discussion, we use the ratio between the ACK-time and the RTT (called *the ACK-time ratio*) as an indication of the proximity of the measurement point to the receiver.

To summarize, we split the lifetime of a connection into "flights", and test, at the end of each flight, if the number of packets sent is equal to the sender's window at the beginning of the flight. We only examine senders that transmit more than 4 flights (i.e., send more than 13 data packets). Nearly 54% of these senders are greedy.

In order to test the impact of the proximity to the receivers on our classification, we group the senders into four categories depending on their *ACK-time ratio* (Figure 8). The figure shows that the fraction of greedy senders is largest in the category with an *ACK-time ratio* of at least 0.75 (the measurement point is far from the receivers). In this category, as much as 68% of all senders are greedy for all the flights, and nearly 79% of senders are greedy for 75% of the flights transmitted.

We can consider the greediness of these senders to be a close approximation of the percentage of actual greedy senders. However, it is important to know whether this is a representative set of senders. In order to address this question, we compare the distribution of size of the connections with an *ACK-time ratio* greater than 0.75 with that of all other connections. Figure 9 plots the quantile-quantile plot of the log of the connection sizes. We can observe an almost perfect linear fit for flow sizes until 100 data packets (which corresponds to a little more than 90% of all flows in these two populations), and a linear-seeming trend for the higher percentiles. This lends confidence to our assumption that the connections with large *ACK-time ratio* are representative of the total population of connections in our traces.

D. Round trip times

Figure 10 plots the minimum and median values of the RTTs of the connections in the three data sets. We only plot RTTs for those connections with at least 10 RTT samples (in both directions). We choose this threshold to discount the effects of single erroneous values, and also to have enough samples to examine the variability of the round trip time within the lifetime of a flow.

We observe that for 50% of the examined senders the minimum value of the RTT is less than 150-200 msecs. Note, however, that the RTT can range from less than 10 msecs, to as high as 10 secs. The distribution of the median value of the RTT in the lifetime of the flow is similar in shape to that of the minimum RTT, and in 50% of the senders, this value is less than 300-400 msecs.

Next, in Figure 11 we look into the RTT variability within a connection. To this end, we consider the 95th-

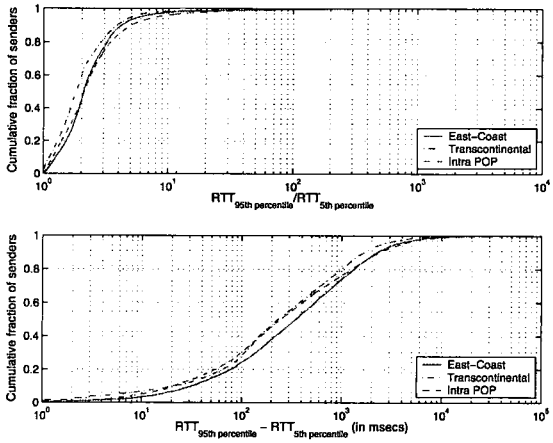

Fig. 11. Variability of RTT. Top: ratio 95th/5th percentile; Bottom: difference between 95th and 5th percentiles.

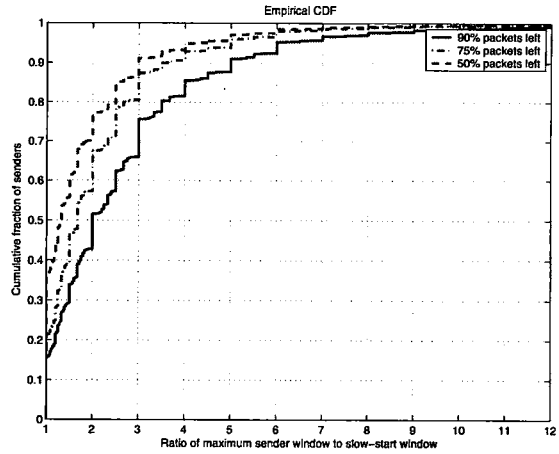

Fig. 12. Ratio of maximum sender window to the window size before exiting slow-start

percentile and the 5th-percentile of the RTT samples. In the top plot of Figure 11 we look at the ratio between these two values. Quite consistently across the three data sets, for as much as 50-60% of all connections, the 95th-percentile RTT is less than two times the 5th-percentile, while for nearly 40% of all senders, this can range between 2-6 times. In absolute terms, 50% of the connections exhibit an RTT variability of less that 200-300ms (bottom plot in Figure 11). Similar observations about the variability in TCP round-trip times have also been recently reported in [1], using measurements taken at a university access link.[10] In their work, the authors report that for 40% of the senders, the ratio between the 90th-percentile and the minimum RTT sample of a connection is less that 2, and for about 50% of the senders, this ratio ranges between $2 - 10$.

E. Efficiency of slow-start

TCP's slow-start mechanism is intended to quickly identify the "equilibrium" point of a connection so that a sender can run stably with a full window of data in flight [8]. Once the connection reaches the equilibrium, congestion avoidance is used to keep the connection operating around that point. The assumption behind this overall approach is that the first loss event is a valid estimator of the available bandwidth.

In order to verify this assumption we compare the maximum window reached by a sender with the window right before exiting slow-start (called *sswnd*). Figure 12 plots the CDF of the ratio between the maximum window and *sswnd*. We only consider senders with more than 13 packets to transmit and divide them into three categories based on the packets sent after exiting slow-start (50%, 75% and 90% of all sent packets).

We observe that 25% to 55% of the senders reach a maximum window that is two or more times larger than *sswnd*. Note that a TCP connection assumes that the data rate reached during slow-start is at most twice the available bandwidth. Figure 12 shows that a large portion of the connections actually reach rates that are four times the estimated available bandwidth.

This may indicate that TCP connections are overly conservative and may operate at rates far from the available bandwidth. Once in congestion avoidance, a TCP connection could spend many RTTs to move away from the slow-start dictated operating point. Further investigation is needed to understand the impact of slow-start on application performance and to identify how to improve the bandwidth estimation (or quickly reverse an incorrect decision). This will be part of our future work.

VIII. Conclusions

We have presented a passive measurement methodology that observes the segments in a TCP connection and infers/tracks the time evolution of two critical sender variables: the sender's congestion window (*cwnd*) and the connection round trip time (*RTT*). The measurement point itself can be anywhere between the sender and the receiver. This allows us to apply our methodology to packet traces collected in the middle of a backbone, providing a significant advantage in comparison to traditional active end-to-end measurements. We are able to monitor millions of TCP connections originated and destined to a large portion of the entire Internet.

[10]Given the fact that the measurement point is close to the sender in [1], the authors compute a RTT to be the time difference between a data packet and its corresponding ACK.

We have also identified the difficulties involved in tracking the state of a distant sender and described the network events that may introduce uncertainty into our estimation, given the location of the measurement point. We have also defined bounds on these uncertainties and investigated how frequently they occur in the traces under study.

Our results lead to the following observations:

- The sender throughput is often limited by lack of data to send, rather than by network congestion.
- In the few cases where TCP flavor is distinguishable, it appears that NewReno is the dominant congestion control algorithm implemented.
- Connections do not generally experience large RTT variations in their lifetime. The ratio between the 95th and 5th percentile of the RTT is less than 3 for approximately 80-85% of the connections.

As part of our future work, we would like to investigate the efficiency of TCP slow-start, specifically, the issue that the first packet loss a connection experiences is not a particularly good estimator for the available bandwidth along the path.

Looking further, our work represents a fundamental building block for addressing a wide range of research questions about Internet traffic, such as identifying the root causes behind packet losses (i.e., congestion, routing, network failures). The methodology proposed in [11] to classify out-of-sequence packets represented a first building block. This work complements [11] with a technique to accurately estimate the sender's state and the connection flavor.

The next step is to correlate the behavior of different TCP connections to identify common patterns. This would allow us to understand if connections experience congestion in one or multiple points in the network. It would also allow us to identify the Autonomous Systems that contribute significantly to connection loss, and thus drive down connection throughput.

ACKNOWLEDGMENTS

The authors would like to thank Richard Gass at Sprint ATL for systems support and help in setting up the experiments for this work, and to the anonymous reviewers for their detailed comments.

REFERENCES

[1] J. Aikat, J. Kaur, F. Smith, and K. Jeffay. Variability in tco roundtrip times. In *Proceedings of ACM Sigcomm Internet Measurement Conference*, Oct. 2003.

[2] M. Allman, S. Floyd, and C. Partridge. Increasing TCP's initial window. RFC 2414, Sept. 1998.

[3] M. Allman, V. Paxson, and W. R. Stevens. TCP congestion control. RFC 2581, Apr. 1999.

[4] L. Brakmo and L. Peterson. TCP Vegas: End to end congestion avoidance on a global internet. *IEEE Journal on Selected Areas in Communications*, 13(8):1465–1480, Oct. 1995.

[5] C. Casetti, M. Gerla, S. Mascolo, M. Y. Sanadidi, and R. Wang. TCP Westwood: Bandwidth estimation for enhanced transport over wireless links. In *Proceedings of ACM Mobicom*, pages 287–297, July 2001.

[6] S. Floyd and T. Henderson. The NewReno modification to TCP's fast recovery algorithm. RFC 2582, Apr. 1999.

[7] C. Fraleigh, S. Moon, B. Lyles, C. Cotton, M. Khan, R. Rockell, D. Moll, T. Seely, and C. Diot. Packet-level traffic measurements from the Sprint IP backbone. *IEEE Network*, 2003.

[8] V. Jacobson. Congestion avoidance and control. In *Proceedings of ACM Sigcomm*, pages 314–329, 1988.

[9] V. Jacobson, R. Braden, and D. Borman. TCP extensions for high performance. RFC 1323, May 1992.

[10] 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.

[11] 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. In *Proceedings of IEEE Infocom*, Mar. 2003.

[12] H. Jiang and C. Dovrolis. Passive estimation of TCP round-trip times. Technical report, July 2001.

[13] H. Martin, A. McGregor, and J. Cleary. Analysis of internet delay times. In *Proceedings of Passive and Active Measurment Workshop (PAM)*, 2000.

[14] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP selective acknowledgement options. RFC 2018, Oct. 1996.

[15] J. Padhye and S. Floyd. On inferring TCP behavior. In *Proceedings of ACM Sigcomm*, Aug. 2001.

[16] V. Paxson. Automated packet trace analysis of TCP implementations. Sept. 1997.

[17] V. Paxson. End-to-end internet packet dynamics. *IEEE/ACM Transactions on Networking*, 7(3):277–292, June 1999.

[18] K. Ramakrishnan, S. Floyd, and D. Black. The addition of explicit congestion notification (ECN) to IP. RFC 3168, Sept. 2001.

[19] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1), Jan. 1997.

[20] K. Thompson, G. J. Miller, and R. Wilder. Wide-area internet traffic patterns and characteristics. *IEEE Network Magazine*, Nov. 1997.

[21] Y. Zhang, L. Breslau, V. Paxson, and S. Shenker. On the characteristics and origins of internet flow rates. In *Proceedings of ACM Sigcomm*, Aug. 2002.

APPENDIX

procedure *updateCwnd*

Inputs :

Argument	Input type	Remarks
packetType	RTO_Data | ACK | Dup_ACK	packet types which cause this procedure to be invoked
num_dupacks	*Integer*	number of duplicate ACKs
ack_seqno	*Sequence number*	sequence number of last ACKed data packet
next_seqno	*Sequence number*	sequence number of the next expected data packet
ocwnd	*Integer*	value of *cwnd* before it deflates, used by procedure *flavorTest*
numACKed	*Integer*	number of packets cumulatively acked by a new ACK, used only by NewReno

Variables :

Variable name	Type	Remarks
cwnd	*Integer*	sender's congestion window estimate
ssthresh	*Integer*	sender's slow start threshold estimate
recover	*Sequence number*	for New Reno, sequence number of last data packet observed before entering fast recovery
state	DEFAULT	Tahoe states
	DEFAULT | FAST_RECOVERY	Reno and NewReno states
MSS	*Bytes*	global variable, Maximum Segment Size for this connection
awnd	*Integer*	Another global variable, the receiver's advertised window

Outputs

Name	Remarks
cwnd, ssthresh, ocwnd	updated values of cwnd, ssthresh and ocwnd

Input Format - for Tahoe, Reno

<packetType, num_dupacks, ack_seqno, next_seqno, ocwnd>

Input Format - for NewReno

<packetType, num_dupacks, ack_seqno, next_seqno, ocwnd, numACKed>

Fig. 13. Variables required to keep track of the cwnd of the three TCP flavors

procedure *incr_cwnd*
Inputs :
cwnd, ssthresh

Outputs :
cwnd

if($cwnd < ssthresh$)
    $cwnd++;$
else
    $cwnd += 1/cwnd$

Fig. 14. Function used to update *cwnd* during slow-start and congestion avoidance

State	Input	Action
<DEFAULT>	<ACK, *, *, *, *ocwnd*> <Dup_ACK, 3, *ack_seqno*, *next_seqno*, *ocwnd*>	$incr\_cwnd(cwnd, ssthresh)$; $ssthresh = max(min(awnd, cwnd)/2, 2)$; $if(flightsize < min(cwnd, awnd))$ $\quad ocwnd = min(cwnd, awnd)$ $elseocwnd = 0$ $cwnd = 1$;
	<RTO_packet, *, *, *, *ocwnd*, >	$ssthresh = max(min(awnd, cwnd)/2, 2)$; $cwnd = 1$; $ocwnd = 0$;

Fig. 15. Pseudocode to track congestion window for Tahoe

State	Input	Action
<DEFAULT>	<ACK, *, *, *, *ocwnd*>	$incr\_cwnd(cwnd, ssthresh)$; $cmp\_ocwnd(cwnd, ocwnd)$;
	<Dup_ACK, 3, *ack_seqno*, *next_seqno*, *ocwnd*>	$ssthresh = max(min(awnd, cwnd)/2, 2)$; $if(flightsize < min(cwnd, awnd))$ $\quad ocwnd = min(cwnd, awnd)$ $else\ ocwnd = 0$ $cwnd = cwnd/2 + 3$; $state$ = FAST_RECOVERY;
	<RTO_packet, *, *, *, *ocwnd*>	$ssthresh = max(min(awnd, cwnd)/2, 2)$; $cwnd = 1$; $ocwnd = 0$;
<FAST_RECOVERY>	<ACK, >= 3, *ack_seqno*, *next_seqno*, *ocwnd*>	$if(flightsize < min(cwnd, awnd)\ and\ ocwnd == 0)$ $\quad ocwnd = min(cwnd, awnd)$ $else\ ocwnd = 0$ $cwnd = ssthresh$; $num\_dupacks = 0$; $state$ = DEFAULT;
	<Dup_ACK, *, *, *, *ocwnd*>	$cwnd++$; $cmp\_cwnd(cwnd, ocwnd)$;
	<RTO_packet, *, *, *, *ocwnd*>	$ssthresh = max(min(awnd, cwnd)/2, 2)$; $cwnd = 1$; $ocwnd = 0$; $num\_dupacks = 0$; $state$ = DEFAULT;

Fig. 16. Pseudocode to track congestion window for Reno

State	Input	Action
<DEFAULT>	<ACK, *, *, *, *ocwnd*, *numACKed*>	*incr_cwnd*(*cwnd*, *ssthresh*); *cmp_ocwnd*(*cwnd*, *ocwnd*);
	<Dup_ACK, 3, *ack_seqno*, *next_seqno*, *ocwnd*, 0>	$ssthresh = max(min(awnd, cwnd)/2, 2)$; $if(flightsize < min(cwnd, awnd)$ $ocwnd = min(cwnd, awnd)$ $cwnd = cwnd/2 + 3$; $recover = next\_seqno - 1$ $state$ = FAST_RECOVERY;
	<RTO_packet, *, *, *, *ocwnd*, 0>	$ssthresh = max(min(awnd, cwnd)/2, 2)$; $cwnd = 1$; $ocwnd = 0$;
<FAST_RECOVERY>	<ACK, *, *ack_seqno*, *next_seqno*, *ocwnd*, *numACKed*>	$if(flightsize < min(cwnd, awnd))$ $ocwnd = min(cwnd, awnd)$ $else\ ocwnd = 0$

		$num\_dupacks = 0$;
		$if(ack\_seqno > recover)$
		  $state$ = DEFAULT;
		  $recover = 0$;
		  $cwnd = ssthresh$;
		$else$
		  $cwnd = cwnd - numACKed + 1$;

	<Dup_ACK, *, *, *, *ocwnd*, 0>	$cwnd++$; *cmp_cwnd*(*cwnd*, *ocwnd*);
	<RTO_packet, *, *, *, *ocwnd*, 0>	$ssthresh = max(min(awnd, cwnd)/2, 2)$; $cwnd = 1$; $ocwnd = 0$; $recover = 0$ $num\_dupacks = 0$; $state$ = DEFAULT;

Fig. 17. Pseudocode to track congestion window of NewReno

procedure *cmp_ocwnd*
Inputs :
cwnd, *ocwnd*
Outputs :
ocwnd

if($cwnd > ocwnd$)
  $ocwnd = 0$;

Fig. 18. Function used to reset *ocwnd*

procedure *flavorTest*

Inputs :

Argument	Input type	Remarks
$flavor$	TAHOE | RENO | NEWRENO	flavor being tested
cwnd	*Integer*	estimate of *cwnd* for this flavor
ocwnd	*Integer*	value of *cwnd* before it deflated
num_dupacks	*Integer*	Number of duplicate ACKs
ack_seqno	*Sequence number*	last sequence number ACKed
cur_seqno	*Sequence number*	sequence number of current data packet
$flightsize$	*Integer*	the number of packets currently unacknowledged
$inFastrecovery$	*Integer*	flag which denotes if the state m/c for this flavor is in the fast recovery state

Variables :

Name	Type	Remarks
$isFalse$	*Integer*	flag which indicates whether this packet was allowed by this flavor
MSS	*Bytes*	Maximum Segment Size for this connection

Outputs :

$isFalse$

Input Format :

$<cwnd, ocwnd, num\_dupacks, ack\_seqno, cur\_seqno, flightsize, >$

$isFalse = 0;$
if($flightsize > max(cwnd, ocwnd)$)
    if($cur\_seqno > ack\_seqno$)
        $isFalse = 1;$
    if($cur\_seqno == ack\_seqno$)
        if($flavor \neq NEWRENO$)
            if($num\_dupacks < 3$)
                $isFalse = 1;$
        elsif ($inFastrecovery \neq 1$)
            $isFalse = 1;$

Fig. 19. Testing TCP flavors

procedure *rttEstimate*

Inputs:

Argument	Input type	Remarks
packetType	New_Data | Retx_Data | Dup_ACK	packet types which cause this procedure to be invoked
num_dupacks	*Integer*	number of duplicate ACKs
uwnd	*Integer*	current estimate of the sender's usable window, i.e. $min(awnd, cwnd)$
time		the current time
cur_seqno	*Sequence number*	sequence number of a New_Data packet
ack_seqno	*Sequence number*	most recent ACK

Variables:

Name	Type	Remarks
state	DEFAULT | FROZEN	whether RTT estimation is currently frozen or on
startTime		time at which this estimate for the RTT started
startSeqno	*Sequence number*	sequence number of data packet that starts the RTT sample
sampleSeqno	*Sequence number*	sequence number of data packet that completes the RTT sample
MSS	*Bytes*	Maximum Segment Size for this connection

Outputs:

Name	Remarks
rtt_estimate	current RTT sample

Input Format

<packetType, num_dupacks, uwnd, time, cur_seqno | ack_seqno >

State	Input	Action
<FROZEN>	<New_Data, *, *, *time*, *cur_seqno* >	$startSeqno = cur\_seqno$ $sampleSeqno = NOTSET$ $startTime = time$ $state$ = DEFAULT
<DEFAULT>	<New_ACK, *, *uwnd*, *, *ack_seqno*>	if ($ack\_seqno > startSeqno$) $sampleSeqno = startSeqno + uwnd * MSS$
<DEFAULT>	<New_Data, *, *, *time*, *cur_seqno*>	if ($cur\_seqno == sampleSeqno$) $rtt = time - startTime$ $startSeqno = cur\_seqno$ $sampleSeqno = NOTSET$ $startTime = time$
<DEFAULT>	<Retx_Data, *, *, *, *>	$state$ = FROZEN
<DEFAULT>	<Dup_ACK, 3, *, *, *>	$state$ = FROZEN

TABLE III

PSEUDO-CODE DESCRIBING THE RTT ESTIMATION SCHEME