Measuring End-to-End Bulk Transfer Capacity

Mark Allman
BBN Technologies/NASA Glenn Research Center
mallman@bbn.com

Abstract— **This paper provides a preliminary assessment of the effectiveness of an application layer tool that measures the *Bulk Transfer Capacity* (BTC) of a network path. BTC is roughly defined as the throughput that a flow using standard congestion control techniques would obtain across a given network path at a given time. We utilize the NIMI mesh of measurement hosts to compare stock BSD TCP with a new BTC measurement tool, *cap*. While BTC tools have been around for some time, no systematic evaluation of their accuracy with respect to standard TCP congestion control across a wide variety of network paths has been conducted. The goal of this paper is to provide such an empirical evaluation of a BTC tool and therefore assess the reliability of the measurements obtained using BTC tools.**

Keywords— **TCP, Bulk Transfer Capacity, Congestion Control, Bandwidth Measurement**

I. Introduction

This paper examines a tool for measuring a network path's *Bulk Transfer Capacity* (BTC). BTC is loosely defined as the rate that a transmitting entity implementing standard congestion control can attain over a given network path [MA01]. We gave conducted experiments across the NIMI mesh of measurement hosts [PMAM98], [PAM00] using BSD TCP and a newly developed tool called *cap*. The BTC tool, *cap*, can operate in one of two modes. In the first mode, *cap*'s sending process communicates with a receiving process (*capd*) on the destination host. In the second mode of operation *cap* communicates with *capd*, but ignores any cumulative information offered by the receiving process (such as cumulative acknowledgments). In this mode, *cap* infers this information from unique packet numbers returned by the receiver. This mode is meant to emulate a BTC tool that induces ACKs in the form of ICMP messages from a non-cooperating receiver (ala *TReno* [Mat96]). (Note: Due to space constraints the one-way emulation mode is not further discussed in this paper, but rather will be discussed in an extended version in the future.)

In this paper, "standard congestion control" means the algorithms standardized in RFC 2581 [APS99] for TCP (the only form of congestion control currently standardized by the IETF). While BTC tools have been around for some time, no wide-scale investigation of their accuracy has been conducted over a wide range of network paths. Our focus in this paper is on determining the extent to which a tool can determine the BTC of a given network path. Our motivations behind developing and evaluating BTC tools are as follows.

- A tool that can accurately measure the BTC of a network path can be used to find and diagnose problems in the network. For instance, periodic BTC measurements could be used to detect changes in the network path and determine the cause of the change (e.g., sudden increase in the packet loss rate).
- A tool that measures BTC in a uniform manner without regard to the underlying operating system's TCP implementation is useful for comparing measurements from various networks taken with various host platforms. Of course, it is nearly impossible to completely factor out the impact of the operating system due to items beyond the control of the application (e.g., CPU scheduling policy). However, an application-level tool such as *cap* is likely to produce more uniform results across operating systems than just using the OS's stock TCP implementation to measure the BTC, since TCP implementations have been shown to vary greatly [Pax97].
- A BTC measurement tool implements standard congestion control algorithms and is therefore an at-

tractive method for researching new congestion control mechanisms. Traditionally, such research has involved detailed kernel development followed by attempting to secure a number of hosts around the network whose administrators are willing to run an experimental kernel. Using an application-layer tool makes development easier, as well as simplifying deployment for testing purposes (and, hence, encouraging wide-scale measurement studies). As an example, the *TReno* BTC tool has been used to develop advanced loss recovery algorithms [MM96] based on TCP's selective acknowledgment option [MMFR96]. Finally, we believe that finding and fixing bugs in an application-level program is likely to be easier than doing so inside the kernel.

• Several tools exist to determine the "available bandwidth" of a network path (see § II for a brief outline of related work). However, using standard congestion control techniques, a flow may not be able to fully utilize the available bandwidth of the network path. The hope is that the BTC is closer to the throughput an application can expect than an available bandwidth estimate will predict.

• A BTC tool can provide a safe way to measure details about a network path in addition to the BTC. For instance, a BTC tool can measure the loss rate along a given path. Using a congestion aware tool provides a safe way to measure path properties. For example, some researchers have used floods of packets to measure some path property such as packet reordering [BPS99] or the burst capacity [AHO98]. However, a BTC tool that uses standard congestion control makes such assessment more network friendly. In addition, using a BTC tool to measure path properties allows us to assess the prevalence of certain events (e.g., loss or reordering) on timescales that matter to real applications (most of which currently use TCP [MC00]).

It should be noted that while we are using BSD TCP as a benchmark in this study, *cap does not* attempt to faithfully emulate this variant of TCP. Rather, *cap* attempts to implement the congestion control algorithms specified in [APS99], which allows implementers discretion in some of the details of the algorithms (as outlined in [MA01]). Therefore, we do not expect *cap* and TCP will agree exactly – just as we would not expect two independent implementations of TCP to agree exactly.

This paper is organized as follows. § II discusses related work. § III provides a detailed discussion of the tools used in our investigation. § IV outlines our experimental methodology. § V compares the BTC tools performance with that of TCP. Finally, § VI gives our conclusions and outlines future work in this area.

II. Related Work

Several researchers have defined methods for measuring bandwidth for some definition of bandwidth. Due to space constraints a detailed discussion of these tools is not included. However, some of these tools are: *ttcp*, *netperf*, *cprobe* [CC96], *pathchar* [Jac97], [Dow99].

III. Tools

The following is a discussion of the tools employed in our investigation and their use in our experiments.

A. NIMI

We utilized the NIMI measurement infrastructure [PMAM98], [PAM00] to conduct our investigation. NIMI is a generic system capable of running specific measurements at future points in time, as scheduled by the user. In addition, the request indicates where the NIMI host should send the results after the measurement is completed. As outlined in [PMAM98] a mesh of hosts, such as NIMI, has good scaling properties. Utilizing a mesh of n hosts we can measure $O(n^2)$ network paths. Our investigation spans nearly 900 network paths yielding results without strong biases based on the network.

We installed two scripts on the hosts in the NIMI infrastructure to conduct our tests (a script to execute the sending process and another to initiate the receiving process). Each script performs two data transfers (with the BTC tools discussed in the subsequent subsections), as well as recording various other information about the test. For instance, we record the time each measurement is initiated and the time it is completed. Therefore, even if the data transfer fails for some reason the NIMI host should still return something for each measurement. We consider cases when this does not happen to be NIMI failures, not transfer failures.

B. *traffic/discardd*

To make TCP transfers between a pair of hosts we wrote two small programs to act as a data generator (*traffic*) and a data receiver (*discardd*). A TCP connection is established between these two user-level programs and a memory-to-memory data transfer is performed. The *traffic* program allows for sending a given amount of data or transmitting an arbitrary amount of data for a given length of time. In addition, the tools allow the user to configure the size of the send and receiver socket buffers (and, hence the advertised window). In our tests we set both buffers to 195,480 bytes (135 segments, assuming each segment carries 1448 bytes of data). The use of window sizes greater than 64 KB requires the use of the window scaling and timestamp TCP options [JBB92]. In the TCP implementations used in our experiments the timestamp option is also used for the *Round-Trip Time Measurement* algorithm [JBB92]. Using this algorithm, each ACK contains the timestamp found in the data packet that triggered the ACK. Upon each ACK arrival the TCP sender takes a round-trip time (RTT) sample and updates its estimate of the *retransmission timeout* (RTO) [PA00]. Finally, the BSD TCP implementations used in our measurements employ delayed ACKs [Bra89], [APS99].

We could not use one of the standard tools, like *ttcp* or *netperf*, in our TCP experiments because in addition to sourcing or sinking data both *traffic* and *discardd* can use the *libpcap* interface to the Berkeley Packet Filter [MJ93] to allow capturing of packet traces of the data transmission. NIMI does not provide access to a general facility for packet tracing and therefore we were required to build the feature into the tool itself. Tracing packets may slow down the data transfer on some slow machines and/or fast networks. We transferred 100 MB of data between two Pentium III NetBSD 1.3 machines connected via a lightly loaded 10 Mbps Ethernet and found a 4% decrease in performance when packet level tracing was enabled in both *traffic* and *discardd*. However, we note that the data transfer rate was over 800,000 bytes/second which is faster than the vast majority of the connections in our dataset. Furthermore, we enabled packet level tracing in both *traffic* and *discardd* for *all* experiments presented in this paper, therefore we expect any slight reduction in throughput to be experienced by all transfers.

C. *cap*

As outlined in § I we developed a BTC tool called *cap*. This tool actually consists of two programs – a sender (*cap*) and a receiver (*capd*). The tools send UDP "data" and "acknowledgment" (ACK) packets that contain a sequence/ACK number and a unique packet identifier to facilitate the tools emulating TCP behavior (as well as careful analysis of the transfers). In addition, each segment contains a flags field and possibly a number of options (e.g., selective acknowledgment information). The operation of each program is controlled using a large number of command line options that allow for the modification of various algorithms and constants used by the congestion control algorithms (e.g., the initial value of the congestion window).

One advantage of *cap* is that the user has control over both the sender and receiver. For instance, the user can opt to use a larger initial congestion window [AFP98] on the sender side and delayed ACKs [Bra89], [APS99] with a given ACK interval on the receiver side. This allows researchers to test the impact of new algorithms and their interactions with current mechanisms. In addition, having measurement tools on both the sender and receiver side of the "connection" allows for packet-level traces to be collected at two vantage points allowing for more robust analysis of the transfer's behavior. A disadvantage of *cap* is that it *must* be installed on the two end-points of the connection and therefore must be used between cooperating machines.

In our measurements, we set the maximum congestion window (essentially a sender-imposed advertised window) to 135 segments, just as for our TCP transfers. In addition, we enabled to use of timestamps (and the RTTM algorithm) in our *cap* measurements in order to mimic our TCP transfers. Note that window scaling is not needed in *cap*. Finally, we configured *capd* to use delayed ACKs.

IV. Methodology

We scheduled measurements on NIMI hosts using a Poisson process with a mean of 30 seconds between measurements. Each measurement consisted of executing a script that makes two back-to-back BTC measurements using either *cap* or TCP. In this preliminary presentation we do not discuss any results involving *cap*'s one-way emulation mode. These results will be presented in an expanded version of this paper. The transfers are conducted between two randomly chosen NIMI hosts. The tool used to make each of the two transfers is chosen at random (with replacement). Each transfer consists of 1,049,800 bytes (or 725 segments carrying 1448 bytes of data each). We combine the measurements using *cap* for the first transfer and TCP for the second transfer with the measurements that use TCP for the first transfer and *cap* for the second transfer. After scrubbing the bad data from the dataset we collected[1] we ended up with 104 measurements that use TCP for both transfers, 109 measurements that use *cap* for both transfers and 181 measurements that use *cap* for one of the transfers and TCP for the other and.

V. Results

First, we examine the experiments and the degree to which each tool delivers consistent results. Obviously, an exact match between two transfers is not possible due to differences in the network conditions. However, by observing the distribution of the difference in the transfer rate between two transfers made with the same tool the differences between *cap* and TCP should become apparent.

Figure 1 shows the stability of both TCP and *cap*. That is, the ability of each BTC tool to deliver consistent results across the two transfers. We expect some variation in the throughput observed due to changing network conditions, as indicated by the distribution of the percent difference in throughput TCP is able to achieve. The measurements involving only *cap* have a similar distribution of percent difference in throughput when compared to the measurements involving only TCP transfers. We therefore conclude that *cap* is roughly equivalent to TCP in its ability to make consistent measurements.

Figure 2 shows the distributions of throughput obtained by *cap* and BSD TCP during the measurements involving one transfer of each type. As shown, the throughput distribution across all *cap* transfers is similar to that of all TCP transfers. While we cannot expect these lines to match exactly due to changing network conditions the trend shown is clearly that *cap* obtains slightly more throughput that TCP. While we cannot conclusively say why this is happening we believe that it may be caused by the difference in the congestion avoidance algorithm employed by each tool. While BSD TCP increases *cwnd* by 1/*cwnd* for each incoming ACK, *cap* waits for an entire *cwnd* to be acknowledged and then increased *cwnd* by 1 segment. Therefore, in the face of delayed ACKs and ACK loss *cap* is slightly more aggressive than TCP (but, still within the specification [APS99]). We believe this may explain the difference in throughput, but are continuing the investigation into this phenomenon. Further, we have

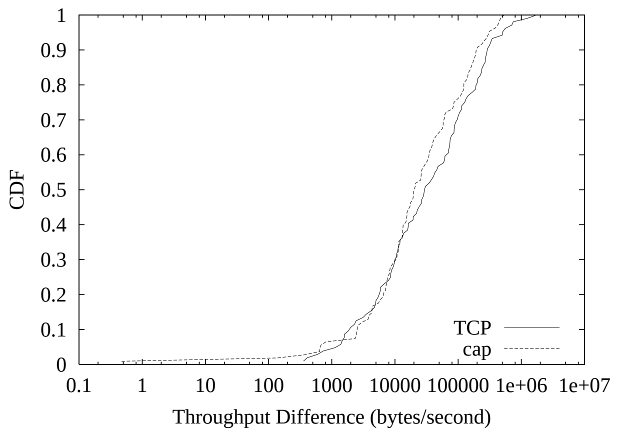

Fig. 1. Stability of the BTC tools used in our study.

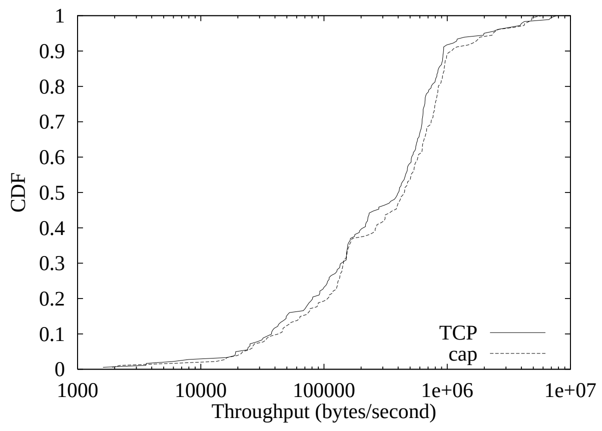

Fig. 2. Distribution of *cap* and TCP throughput.

[1]We omit the details of this scrubbing due to space constraints, but will include all such details in an extended version of this paper in the near future. Measurements were removed for various reasons including NIMI failures, clocks that were not synchronized, and general mysterious failures that resulted in not all results being returned to us.

changed *cap* to use the algorithm used by BSD TCP and are conducting experiments.

Finally, we compare the distribution of loss rates across the TCP and *cap* transfers, since research has shown the loss rate to be a key contribution to TCP performance [PFTK98]. We found that 85% of the measurements indicate a loss rate agreement within 1% between TCP and *cap*. In addition, nearly all measurements show a percent difference less than 10% between the tools. This indicates that *cap* is performing reasonably similar to TCP when using loss rate as the metric.

VI. Conclusions and Future Work

This paper has outlined the benefits of using an application layer BTC tool as well as some preliminary results of one such tool. The early results presented in the last section indicate that *cap* is likely to be able to measure the BTC of a network path with similar accuracy to that of TCP.

Acknowledgments

This work benefited from discussions with the following people: Andy Adams, Sally Floyd, Joseph Ishac, Matt Mathis and Vern Paxson. In addition, Andy Adams and Vern Paxson provided a large amount of help in using the NIMI infrastructure. My thanks to all!

References

[AFP98] Mark Allman, Sally Floyd, and Craig Partridge. Increasing TCP's Initial Window, September 1998. RFC 2414.

[AHO98] Mark Allman, Chris Hayes, and Shawn Ostermann. An Evaluation of TCP with Larger Initial Windows. *Computer Communication Review*, 28(3), July 1998.

[APS99] Mark Allman, Vern Paxson, and W. Richard Stevens. TCP Congestion Control, April 1999. RFC 2581.

[BPS99] Jon Bennett, Craig Partridge, and Nicholas Shectman. Packet Reordering is Not Pathological Network Behavior. *IEEE/ACM Transactions on Networking*, December 1999.

[Bra89] Robert Braden. Requirements for Internet Hosts – Communication Layers, October 1989. RFC 1122.

[CC96] Robert Carter and Mark Crovella. Measuring Bottleneck Link Speed in Packet-Switched Networks. Technical Report BU-CS-96-006, Boston University, March 1996.

[Dow99] Allen Downey. Using pathchar to Estimate Internet Link Characteristics. In *ACM SIGCOMM*, September 1999.

[Jac97] Van Jacobson. Pathchar – A Tool to Infer Characteristics of Internet Paths, April 1997. Talk at MRSI.

[JBB92] Van Jacobson, Robert Braden, and David Borman. TCP Extensions for High Performance, May 1992. RFC 1323.

[MA01] Matt Mathis and Mark Allman. A Framework for Defining Empirical Bulk Transfer Capacity Metrics, April 2001. Internet-Draft draft-ietf-ippm-btc-framework-06.txt.

[Mat96] Matt Mathis. Diagnosing Internet Congestion with a Transport Layer Performance Tool. In *Proceedings of INET '96*, June 1996.

[MC00] Sean McCreary and K. Claffy. Trends in Wide Area IP Traffic Patterns A View from Ames Internet Exchange. May 2000. http://www.caida.org/outreach/papers/AIX0005/.

[MJ93] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Winter USENX Conference*, January 1993.

[MM96] Matt Mathis and Jamshid Mahdavi. Forward Acknowledgment: Refining TCP Congestion Control. In *ACM SIGCOMM*, August 1996.

[MMFR96] Matt Mathis, Jamshid Mahdavi, Sally Floyd, and Allyn Romanow. TCP Selective Acknowledgement Options, October 1996. RFC 2018.

[PA00] Vern Paxson and Mark Allman. Computing TCP's Retransmission Timer, November 2000. RFC 2988.

[PAM00] Vern Paxson, Andrew Adams, and Matt Mathis. Experiences with NIMI. In *Proceedings of Passive and Active Measurement*, 2000.

[Pax97] Vern Paxson. Automated Packet Trace Analysis of TCP Implementations. In *ACM SIGCOMM*, September 1997.

[PFTK98] Jitendra Padhye, Victor Firoiu, Don Towsley, and Jim Kurose. Modeling TCP Throughput: A Simple Model and its Empirical Validation. In *ACM SIGCOMM*, September 1998.

[PMAM98] Vern Paxson, Jamshid Mahdavi, Andrew Adams, and Matt Mathis. An Architecture for Large-Scale Internet Measurement. *IEEE Communications*, 1998.