

IMPROVING THE PERFORMANCE OF COOPERATING TCP CONNECTIONS

Dorgival Guedes*
Computer Science Department
Universidade Federal de Minas Gerais
dorgival@dcc.ufmg.br

Larry Peterson
Computer Science Department
Princeton University
llp@cs.princeton.edu

Abstract

Many applications use multiple concurrent TCP connections. In such cases, both the application and the network benefit from the collection of TCP connections cooperating rather than competing with each other for scarce resources (e.g., router buffers). This paper shows how the performance of an application that uses multiple concurrent TCP connections can be improved by forcing all the connections through rate controllers that shape the TCP traffic. The rate controller limits the bandwidth used by any connection, allows information about bandwidth and contention to be shared among connections, and breaks up packet trains. Simulation results show that rate control improves the performance of single-client/multiple-server connections in a LAN (parallel file system case) by 15-100%, and multiple-connections/single-server connections in a WAN by up to 50%.

1 Introduction

It is increasingly common to find multiple concurrent TCP connections running on behalf of a single application. This situation happens, for example, when a client of a scalable storage server establishes a TCP connection to each of several server nodes. In such a scenario, the application's request is not satisfied until the slowest transfer completes.

In typical implementations, each TCP connection is independent, deciding how much data to send and when to send it according to its own congestion control state. Unfortunately, such a set of TCP connections tend to compete with each other for link bandwidth—as well as buffers in the intermediate router(s). This

often results in under-utilization of the shared link, and hence, less aggregate throughput than the application is capable of achieving. More specifically, it often happens that some of the connections stall due to multiple losses, while the others proceed unaffected. This leads to reduced throughput for the application if the remaining sessions are not capable of utilizing the available bandwidth. In the worst case, which typically happens on high-bandwidth/low-latency networks, the link is actually idle due to the TCP timeout interval being longer than the required transfer time.

This paper describes how to improve the performance achieved by applications using concurrent TCP connections by adding rate control mechanisms to TCP. These mechanisms are not directly associated with individual connections, but are instead applied to a group of connections that share some network resource. Such aggregation improves the overall system performance by allowing connections to share their information about a critical resource, the bottleneck link, and by avoiding packet trains in individual connections.

2 TCP Limitations

The general TCP congestion avoidance algorithm was developed to guarantee that TCP traffic flow continuously, without congestion [6]. In some cases, however, the algorithm cannot handle problems like packet trains.

Packet trains are multiple packets belonging to one connection that are sent back to back, with no inter-packet gaps. Theoretically, if a connection operates below the speed of the sender output link, packets should be equally spaced, so that they would always keep the same sending rate over time. In practice, however, trains are a common factor, for several reasons:

*Partially sponsored by Conselho Nacional de Pesquisa (CNPq), Brazil, grant nos. 200861/93-0 and 300445/99-7

- During slow start, for each ACK received, TCP sends at least two new packets¹.
- In case an ACK is lost, a later ACK may be received acknowledging all the data before it and opening the window by many packets at once.
- Acks reaching a router behind longer packets may get queue together (ACK compression).

When packets trains arrive in a router, one or more packets in a train may be lost. This is not a loss caused by the additive increase in TCP congestion avoidance algorithm, but a spurious drop. If two such losses happen close enough together, the sender will not be able to recover based on the information gathered from duplicate acknowledgements, and the connection will stall until a timeout occurs.

It should be clear that on any case where a single connection exists, stalling due to losses always causes a degradation in performance. Degradation may still occur even when there are multiple connections and some of them continue to operate. Figure 1 exemplifies such a case. It shows the evolution of sequence numbers sent over time for two connections.

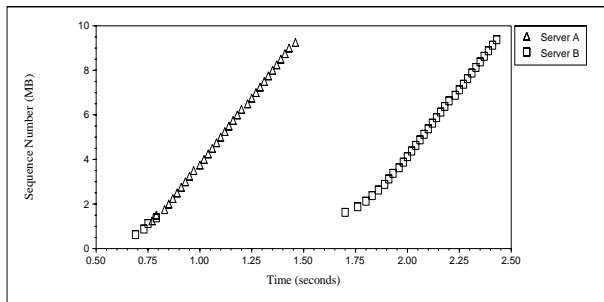


Figure 1: Poor link utilization due to stalling

In this scenario, a client on a high-bandwidth link (OC-3 in this case) is receiving data from two servers, each connected to the switch by 100baseT links. In theory, the application should be able to achieve throughput equal to its link capacity, 155 Mbps, but the final result in this case is just around 70 Mbps. There are two causes for this under-utilization:

- When one of the connections stalls, there is just one server capable of transmitting, and it is limited to a rate of 100 Mbps.
- The rates are high enough that the server that did not stall manages to complete its transfer in less

¹Three, if delayed ACKs are used.

time that it takes the other server to recover with a timeout, leaving the link idle.

From these results it is clear that there is a lot of room for improvement in such situations. Our goal is to improve the utilization of the network under these circumstances, avoiding stalls as much as possible and guaranteeing that connections are able to operate concurrently and fairly. The way to achieve this requires breaking up packet trains so as to restore packet spacing, which can be done by adding rate control to the protocol stack under TCP.

3 Related Work

The last few years have seen a large number of papers dealing with TCP performance for multiple connections in different scenarios. Clark *et al.* [9] discuss the effects of window synchronization among sessions sharing a common bottleneck. Morris [7] discusses the effects when the number of sessions grows.

Some solutions have focused on the behavior of a TCP connection after losses occur, like *New Reno* and SACK TCP [4]. Our approach, on the other hand, tries to avoid the cases where multiple losses occur.

A few papers address the combined characteristics of multiple connections, like Crowcroft and Oechlin [3] and Balackrishnan *et al.* [1]. They do this by changing TCP internal behavior to combine multiple connections internally. We, on the other hand, try to handle combined sessions without changing TCP drastically.

4 Implementation

We implemented the rate controller, `vrate`, as a *virtual protocol* in the *x*-kernel [8]. In this framework, a virtual protocol is an isolated module that controls the flow of messages up and down the stack without adding new headers or creating new messages.

The general organization is shown in Figure 2. An application can notify `vrate` to group multiple TCP connections that share a network resource. Each connection gets associated with an individual queue in the controller. Connections in a group are then serviced according to an approximated fair queuing policy with a limiting rate.

TCP's view of the available rate is kept by the size of the congestion window and the round-trip time (RTT) so, to a first approximation, the rate information is already known. The problem is that for most implementations of TCP, the RTT is actually measured with a

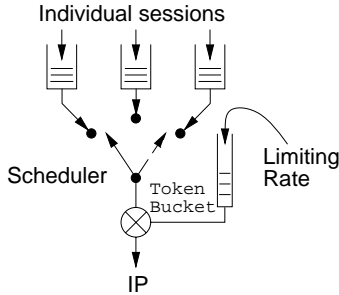


Figure 2: Rate controller internal structure

very coarse-grain timer (500 ms), so it has to be replaced. The performance penalty for this is essentially that of checking the system clock once or twice each round-trip time. When an acknowledgement arrives that allows a new precise RTT to be computed, TCP notifies `vrate`, which in turn combines the information from that connection with that from other connections sharing the same bucket to set a new limiting rate.

The fine-grained timers required to guarantee that the token bucket works close to the limiting rate are implemented using a timing wheel with a short period. All events are grouped by the resolution of the wheel. If rates are too high to be approximated well by that resolution, we just leave the connection uncontrolled for as long as that situation continues. The reasoning is that if the rate is actually that high, the system as a whole will have such hard time keeping up with it that no other limitations are necessary.

5 Simulation Results

This work uses the `x-sim` simulator based on the `x-kernel` framework. It works by executing actual protocol code instead of using abstract specifications, which allows us to use exactly the same code that would be used in a real system [2]. For all configurations tested, we used the standard “Big Windows” TCP as defined in the BSD4.4 distribution.

The main case we chose to study corresponds to the distributed storage server discussed in Section 1. Figure 3 shows the configuration used to represent a local area network with a single shared switch. The links were configured with speeds corresponding to both Fast Ethernet (100 Mbps) and OC-12/ATM (622 Mbps). Delays were fixed at 1ms per link, although slight variations were added in a few runs to make sure there were no phase problems. The switch was modeled as a packet switch with 30 buffers per link.

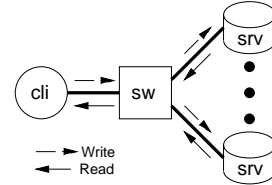


Figure 3: LAN with a single switch

We also considered in the model situations where clients and servers might be connected to different switches linked to each other. The same options of link speeds were used, and all combinations were checked, for completion. Results for this case are presented in another document [5].

For all the cases discussed, we ran simulations varying the number of servers (and hence, the number of concurrent connections) from one to eight. We used both the standard protocol stack and our modified stack, which includes `vrate`. Each case was executed at least ten times.

In virtually all cases, `vrate` performed at least as well as standard “Big Windows” TCP (`btcp`). In those cases that experienced congestion, the application achieved higher performance with the new technique. In the following discussion, we chose a few illustrative cases to show the major gains.

To help visualize the impact of `vrate`, the performance graphs show the bandwidth perceived by the application with a varying number of connections (servers). For each case we present the achieved bandwidth by both `vrate` and `btcp` as a percentage of the maximum bandwidth possible². We chose this presentation to factor out differences in the overall bandwidth available for a given number of servers.

For example, when a client on a OC-12 link reads from a group of servers on 100baseT links, the maximum bandwidth possible per session will be 200 Mbps when accessing two servers, and 622 Mbps when accessing more than six servers.

²Let n_{cli} , n_{srv} be the number of clients and servers. Let bw_{cli} , bw_{srv} and bw_{isw} be the bandwidths for each client, server and inter-switch link (if applicable), respectively. So the maximum available bandwidth for a client is given by:

$$bw_{max} = \min(bw_{cli}, n_{srv} \frac{bw_{srv}}{n_{cli}}, \frac{bw_{sw}}{n_{cli}})$$

Obviously, the last term is not used in configurations with a single switch.

5.1 Single Switch Case: Read

For the model we study, the chances of congestion are always greater when the client reads from the servers. The reasons for that should be obvious: Even if the client has a link of higher bandwidth than the servers, as the application tries to read from a larger number of machines, the aggregate throughput can easily exceed the client’s link capacity.

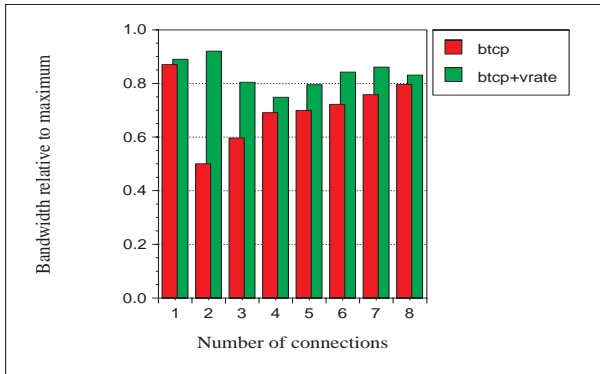


Figure 4: Application performance: OC-12 client reading from 100baseT servers

Figure 4 shows our results for such a case: a client with an OC-12 link reads from multiple servers, each using a 100baseT link. In this case, `vrate` always outperforms `btcp`, although in some cases by just a narrow margin. The results for a single connection, as might be expected, are close together. As the number of servers grows, `btcp` suffers a drastic drop in performance. For two servers, `vrate` achieves a rate 80% better. As the number of servers continues to increase `btcp` improves, but performance is still poor. On the other hand, `Vrate` keeps all connections at a higher level all the time. This is the sort of behavior found in a large subset of the cases we simulated.

The problem with `btcp` is because of connection stalls, as exemplified previously in Figure 1, which was obtained from a trace for this case. When one connection stalls the remaining connection is not able to use all the available bandwidth for the client; furthermore, it finishes before the stalled connection restarts, leaving the link temporarily idle.

The behavior of a transfer using `vrate` is quite different. The two connections proceed close together, never suffering any multiple loss events. In each congestion window period, each connection loses exactly one packet, and they finish close together, yielding a throughput close to the maximum achievable. The

analysis of other cases with more servers confirms these facts.

5.2 Single Switch Case: Write

When a client sends data to a group of servers, the only case where bandwidth is not limited by the client link is when it has a higher capacity link than the servers. Figure 5 shows the results for the write/OC-12 client/100baseT servers case.

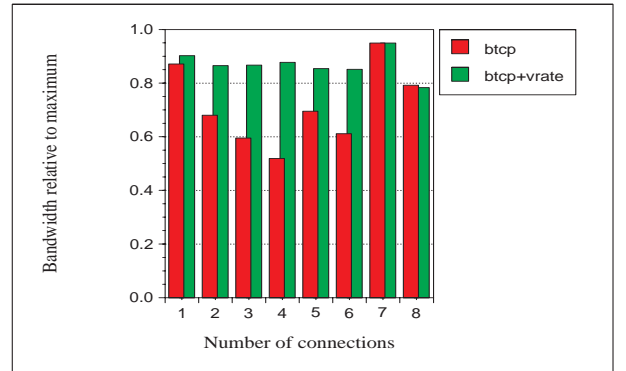


Figure 5: Application performance: OC-12 client writing to 100baseT servers

The main reason for `vrate` performing so much better than `btcp` in this case is that it not only reduces packet trains, but also allows connections to share their knowledge about the link. For seven and eight servers, the throughput is limited by the client capacity, and rate control is not relevant.

5.3 Multiple Clients

When we consider cases with two clients, both of them may be transferring data in the same direction (which accentuates congestion directly), or in opposite directions. In the latter case the main effect is the occurrence of ACK compression in both connections, contributing to losses due to the formation of packet trains.

Most of the time, two way traffic does not affect the performance of `vrate`. In some cases, though, `btcp` was seriously hurt by the increase in the number of packet trains. In those cases that already had high congestion on one connection, both systems were adversely affected. The results nevertheless do not differ significantly from the ones for a single client in terms of general conditions, so we will not discuss them in detail here.

One important aspect of the two-client case is that while many of the cases considered at first were trivial for a single client³, they become contention prone if we consider that multiple applications might be running concurrently in the network (a likely scenario). To illustrate this, consider the results for the single switch, all-100baseT network, with clients writing to servers, as shown in Figure 6.

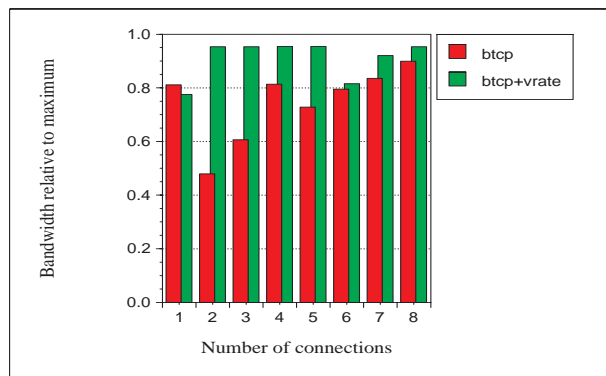


Figure 6: Effect of added clients

With a single client, this case is trivial: No matter what number of servers is considered, the performance is always close to maximum. When we add a second client they compete for bandwidth to access each server. Since their operation is not synchronized, there is no way they may control their transfers so that only one of them talks to any server at any time. *Vrate*'s ability to shape traffic to reduce impact on the switches guarantees that its connections will keep link utilization close to maximum at all times.

6 Conclusions and Future Work

This work discusses problems faced by applications using multiple concurrent TCP connections to transfer data to or from a set of servers. After showing how TCP fails to achieve the best performance possible in many cases due to competition for resources among these otherwise cooperating connections, we propose the use of rate-based traffic shaping to improve performance without changing TCP's basic congestion control algorithm.

The results show that performance improves markedly in cases where packet trains would be important factor acting against the transfer. In those

³Those where the client link was the bottleneck, with performance limited by the sender.

cases where congestion occurs, our solution outperforms plain TCP, sometimes by as much as 100 percent. In virtually all cases, it performs at least as well as plain TCP.

We believe that this technique may be useful specially for applications that require the setup of multiple parallel TCP connections, like scalable storage servers, distributed parallel file systems, and Web servers. Possible ways to continue this work would include analyzing in detail the effects of this technique on wide-area applications, in particular Web servers, and studying different ways to combine the information from cooperating connections, maybe even affecting the scheduling of connections in the kernel.

References

- [1] H. Balackrishnan, V. Padmamabhan, S. Seshan, M. Stemm, and R. Katz. TCP behaviour of a busy internet server. Technical Report CSD-97-966, University of California at Berkeley, 1997.
- [2] Lawrence S. Brakmo and Larry L. Peterson. Experiences with network simulation. In *SIGMETRICS'96*. ACM, June 1996.
- [3] J. Crowcroft and P. Oeschlin. Differentiated end-to-end internet services using a weighted proportional fair sharing TCP. *Computer Communication Review*, 28(3), July 1998.
- [4] Kevin Fall and Sally Floyd. Simulation-based comparisons of Tahoe, Reno and SACK TCP. *Computer Communications Review*, July 1996.
- [5] Dorgival Olavo Guedes. *Operating System and Network Support for High-Performance Computing*. Ph.d. dissertation, Dept. of Computer Science of the University of Arizona, Tucson, AZ, May 1999.
- [6] Van Jacobson. Congestion Avoidance and Control. In *Proc. SIGCOMM '88 Workshop*, August 1988.
- [7] Robert Morris. TCP behavior with many flows. In *Proceedings of the Fifth IEEE International Conference on Network Protocols*. IEEE, October 1997.
- [8] Larry L. Peterson and Bruce S. Davie. *Computer networks: a systems approach*. Morgan Kaufman, 1996.
- [9] Lixia Zhang, Scott Schenker, and David Clark. Observations on the dynamics of a congestion control algorithm: The effects of two way traffic. *ACM Computer Communications Review*, 1991.