Congestion control for low-priority filler traffic

Brian Ott, Timothy Warnky and Vincenzo Liberatore
Division of Computer Science, Case Western Reserve University

ABSTRACT

As the cost of Internet access rises and the amount of deployed bandwidth increases, a way to make efficient use of the oft-unused bandwidth is desired. Simply providing a lower priority for traffic than best effort allows this bandwidth to be used without noticeable interference with regular traffic. Because bursts of normal traffic are given priority over this background, or filler, traffic, a more aggressive congestion control protocol is called for in the filler traffic. In our paper, we compare numerous versions of TCP-like congestion control of our own design, over which to implement low-priority traffic, by using the unused bandwidth at any given time. These protocols are divided into six classes, which differ by the core congestion control algorithm and use different constants. Using the NS-2 network simulator, we collected network traces using each of our protocols in different network configurations, with multiple parameters for each configuration. These configurations simulated high- and low-bandwidth and latency networks. We compared the resulting throughput and sharing — the cumulative variation of throughput over each stream, normalized by the total throughput over the link — to our chosen baseline, TCP SACK. Most of the basic algorithms performed as well as or better than SACK in a background traffic environment, especially in terms of throughput. Using features from multiple classes, we also designed a more complex protocol that performed better than SACK in almost every environment, and performed better than the other algorithms in general.

1. INTRODUCTION

In most networks, a large portion of the bandwidth is unused. If this extra bandwidth could be scavenged without affecting the traffic that is already crossing the network, then a new source of second-class bandwidth would open up. This extra bandwidth can be utilized by two broad classes of background applications: those such as disk backup currently using best-effort traffic, and potential applications for which it is not currently cost-effective to use best-effort traffic (e.g., not-for-profit distributed computing). A method to deliver this non-intrusive service is to introduce a new class of lower priority traffic, which is termed "filler." Filler packets should be recognized by the network nodes and receive lower quality handling than best-effort traffic. Since lower priority packets should not affect other traffic classes, a filler flow could safely have a more aggressive congestion control algorithm than a standard TCP flow. Furthermore, filler flows could be constrained by bursts of other traffic, but cannot affect higher priority flows. Therefore, it is unclear whether current TCP congestion control is a good fit for filler.

In this paper, we will investigate TCP congestion control algorithms especially for filler traffic. Our efforts were directed at maximizing the goodput of filler traffic; previous work has verified that filler traffic implemented with packet differentiation is unobtrusive to other traffic.[1]

The best-performing protocol is named TCP-Background. It is an improvement over TCP SACK in throughput and fairness while fulfilling the requirement of unobtrusiveness.

The rest of our paper is organized as follows: Section 2 will describe TCP SACK, which is used as the baseline. Section 3 will go over the low priority schemes we have devised, and in Section 4, the testing procedures and relevant results of these protocols will be discussed. Section 5 compares our research to similar projects.

Further author information:
Address: 10900 Euclid Ave., Cleveland, Ohio 44106
Email: {bao5,txw46,vxl11}@po.cwru.edu

2. BACKGROUND: TCP SACK

TCP SACK was used as both the baseline protocol and the model for our modified protocols. SACK was chosen because of its ability to encounter multiple drops per RTT without having a timeout; it is also widely used.[2] SACK is based on TCP Reno, and makes use of TCP Reno's congestion-control algorithm. The high-level view of the algorithm is listed in Algorithm 1. Fast recovery improves the overall performance by recovering from drops quicker than previous TCP implementations (i.e., Tahoe), and makes SACK the best starting point with which to compare any other protocols. SACK attempts to give each competing flow a fair amount of bandwidth. However, SACK will turn out to be too passive for implementation of filler traffic because it acts cautiously to avoid congestion. The next section describes our congestion control algorithms for filler traffic.

Algorithm 1 TCP congestion control algorithm

```
if new transmission or drop detected by timeout then
    {Slow-start}
    cwnd = 1
    for each RTT do
        cwnd = cwnd * 2
    end for
end if
if drop detected by dupACK then
    {Fast recovery}
    {Congestion avoidance}
    cwnd = cwnd/2
    for each RTT do
        cwnd++
    end for
end if
```

3. LOW-PRIORITY CONGESTION CONTROL

In contrast to filler traffic, we refer to the higher-priority (including best-effort) traffic as "pre-existing." Our protocols were designed to take advantage of the goals and environment which are different for filler and pre-existing traffic. Filler congestion control can be more aggressive than traditional protocols. The aggressive behavior will cause more filler packets to be queued, causing longer average delays, but the increased latency is not an important concern with filler traffic. If the link is congested by other flows of filler traffic, then any meaningful filler congestion control algorithm should adapt and allow any new filler flow to gain a foothold. Moreover, the aggressiveness of the protocols will not have an adverse affect on higher-priority traffic because of the strict priority differentiation. If the link is congested by pre-existing traffic, then very few, if any, filler packets will be transmitted, but when the pre-existing flow cuts back on its bandwidth usage the filler should adapt quickly. As a result, the protocol should use as much bandwidth as it can, even after a transient burst of pre-existing traffic, have higher goodput and faster transfer time, and be fair to other filler flows. Because current protocols such as SACK and Reno react more cautiously after they detect a drop, they might not get optimum goodput, and might be less suited for use with filler traffic. Filler protocols can cut back less after a drop is detected, and they can also probe for bandwidth faster. For example, Fast Start has already been shown to improve recovery from timeouts.[3] However, the filler protocols cannot ignore drops too much if packet losses are a sign of congestion or if new filler flows are attempting to gain a share of the bandwidth because ignoring losses could eventually lead to lower goodput and fairness.

Fast recovery is of primary interest. Ideally, timeouts will be less frequent than dupACKs, so our efforts were focused on congestion avoidance rather than timeout behavior. Experimental evaluation showed the predominance of dupACKs over timeouts.

Initially, a linear decrease was used as a response to drops, which made very good use of the excess bandwidth. However, the linear back-off was not enough to allow other filler flows to gain a foothold, causing unfairness. We

designed numerous protocols to improve the fairness, described in the following subsections. The constants in the algorithms of these protocols were varied to optimize the performance.

In all cases, the algorithms use a linear increase unless otherwise noted. The primary factor varied was the value of the congestion window after a drop was detected, but the rate of increase was also changed in some circumstances. Specific testing procedures will be described in Section 4. LowPri3 is the most complicated class of algorithms, so will be the last discussed. The "Throughput" and "Sharing" columns in the tables below will be explained in Section 4.

3.1. LowPri1

LowPri1 is an initial linear decrease algorithm. The algorithm called for the congestion window to be set to two less than the previous value: $cwnd = cwnd - 2$.

3.2. LowPri2

LowPri2 introduced an exponential decrease. The initial algorithm chosen was $cwnd = cwnd - cwnd\hat{}0.5$. Table 1 contains a listing of all the variants of this class.

Table 1. Listing of LowPri2 class

Protocol	**Algorithm**	**Throughput**	**Sharing**
LowPri2	$cwnd = cwnd - cwnd\hat{}0.5$	1071.066	0.030037
LowPri2.1	$cwnd = cwnd - cwnd\hat{}0.33$	1098.529	0.183856
LowPri2.2	$cwnd = cwnd - cwnd\hat{}0.66$	1057.272	0.128881
LowPri2.3	$cwnd = cwnd - cwnd\hat{}0.25$	1044.809	0.031748
LowPri2.4	$cwnd = cwnd - cwnd\hat{}0.4$	1057.438	0.017494
LowPri2.5	$cwnd = cwnd - cwnd\hat{}0.6$	1056.435	0.015777
LowPri2.6	$cwnd = cwnd - cwnd\hat{}0.75$	1052.044	0.04042

3.3. LowPri4

LowPri4 was a multiplicative decrease, initially $cwnd = cwnd * 0.66$. Table 2 contains a listing of the variants of this class.

Table 2. Listing of LowPri4 class

Protocol	**Algorithm**	**Throughput**	**Sharing**
LowPri4	$cwnd = cwnd * 0.66$	1098.536	0.047101
LowPri4.1	$cwnd = cwnd * 0.75$	1071.335	0.014867
LowPri4.2	$cwnd = cwnd * 0.25$	1005.489	0.108694

3.4. LowPri5

LowPri5 used a different form of an exponential decrease. Rather than subtract an exponential value as with LowPri2, it reduced the value by a large fractional exponent. Initially, $cwnd = cwnd\hat{}0.9$ was used. Table 3 contains a listing of the variants of this class.

Table 3. Listing of LowPri5 class

Protocol	Algorithm	Throughput	Sharing
LowPri5	$cwnd = cwnd\hat{}0.9$	1068.412	0.033965
LowPri5.1	$cwnd = cwnd\hat{}0.85$	1051.062	0.00853
LowPri5.2	$cwnd = cwnd\hat{}0.95$	1056.368	0.009693

3.5. LowPri6

LowPri6 was designed as a logarithmic decrease, beginning with $cwnd = cwnd - 2 * ln(cwnd)$. Table 4 contains a listing of the variants of this class.

Table 4. Listing of LowPri6 class

Protocol	Algorithm	Throughput	Sharing
LowPri6	$cwnd = cwnd - 2 * ln(cwnd)$	1055.778	0.019107
LowPri6.1	$cwnd = cwnd - ln(cwnd)$	1068.851	0.106186
LowPri6.2	$cwnd = cwnd - 3 * ln(cwnd)$	1055.943	0.014772

3.6. LowPri3

LowPri3 combined two methods of acting more aggressively, cutting back less after a drop, and probing for bandwidth faster. The congestion window is decreased by a multiplicative factor (0.5 or 0.75) and then the traffic slow-starts up to a power of the previous congestion window. For example, TCP-Background is represented by $cwnd = cwnd * 0.75$, slow-start (*0.5) to $cwnd\hat{}0.95$, where (*0.5) is the slow-start factor. The slow-start factor causes the congestion window to increase by `factor` for every ACK it receives, multiplying the congestion window to by `(1+factor)` every round trip time. By default, `factor = 1`, which is equivalent to doubling the congestion window every RTT. This behavior is illustrated in Figure 1. Table 5 contains the listing of all modifications of LowPri3. The protocol previously named "LowPri3.6" is now listed as "TCP-Background," because it was found to be the best overall performer. The analysis will be discussed in Section 4.

Table 5. Listing of LowPri3 class

Protocol	Mult. decrease	Slow-start factor	Exp. limit	Throughput	Sharing
LowPri3	0.5	1	linear - 2*	1027.663	0.5196377
LowPri3.1	0.5	1	0.95	1030.893	0.1310478
LowPri3.2	0.5	1	0.9	978.450	0.5440624
LowPri3.3	0.5	0.5	0.95	1085.909	0.013973
LowPri3.4	0.75	0.5	0.85	1073.387	0.024724
LowPri3.5	0.75	0.5	0.9	1068.524	0.026504
TCP-Background	0.75	0.5	0.95	1102.343	0.021505

* The linear increase as the slow-start threshold was quickly changed to an exponential value after observing the results of LowPri3.

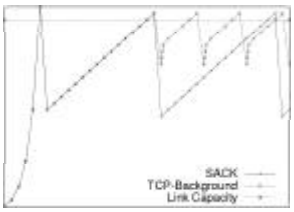

Figure 1. LowPri3.x Congestion Control

4. TESTING AND RESULTS

4.1. Methodology

All of the tests were run using Network Simulator, version 2[4] (NS). We used a dumbbell topology, diagrammed in Figure 2. The 10Mbps central link is the bottleneck is this network. The links from the bottleneck nodes to the sources and destinations were all 100Mbps links. The latency on the bottleneck link was adjusted for various tests, but the links to the source and destination nodes all had 1 ms latency, which made their delay insignificant in relation to that of the central node. All of the tests were run for 3000 seconds to ensure any random variation would be smoothed out. To write our low priority protocols, the congestion control of the existing NS code for TCP SACK was modified.

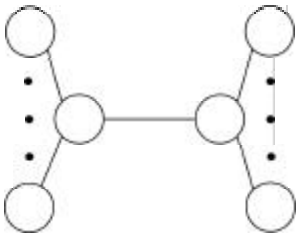

Figure 2. Dumbbell network topology

Harvard traces[5] that were collected on an Ethernet 10 Mbps link were used to simulate pre-existing traffic. There were three traces, collected at different times of the day (8:39am, 12:39pm, and 4:39pm). The average bandwidth utilization differed from trace to trace. We primarily used the first trace, but tests were run with the three different traces to verify the observed behavior. The packet sizes in the traces varied, but were generally either 1500 bytes or much smaller than the MTU. The packet size for the filler traffic was set to 1500 bytes.

Different values were experimented with for a number of factors before choosing a reasonable value that we kept constant for the remainder of the testing. However, both the latency and the buffer size were varied in a number of tests. The size of the advertised window was found to have a slight effect even at very large values. This is because a very large advertised window allows packets to be sent for a portion of a timeout period corresponding to the case when a retransmitted packet is dropped by SACK. Some additional testing led us to an advertised window 5.3 times the size of the BDP (Bandwidth Delay Product). This value is slightly larger than the point at which increasing the size no longer had a significant effect, giving us a margin in case of unexpected variation. The size of the buffer on the bottleneck queue was 1.3 times the BDP as suggested in the literature.[6] The base value of the delay was 20 ms.

To analyze the traces, data was collected on each flow and the throughput, bandwidth, average packet delay, and percentage of packets dropped for each flow were recorded. Most of the tests were conducted with groups of filler flows, so we were interested in the their total throughput and the corresponding sample coefficient of

variation as a measure of the fairness or *sharing*. The formula for sharing is:

$$\frac{\sqrt{\frac{\sum (t_i - t_{avg})^2}{n-1}}}{\sum t_i}, \tag{1}$$

where t is the throughput of a filler flow. The delay, drops, and throughput of the pre-existing traffic were compared against the same tests without filler traffic to make sure the filler flows were not interfering with it noticeably. The testing sequence for each protocol is outlined in Table 6. After running this battery of tests on all of our protocols, additional tests were devised for further analysis of TCP-Background and SACK. These other tests will be described in the next subsection.

Table 6. Testing sequence

Order	Test description
1	One filler flow in isolation
2	One filler flow with a flow of pre-existing traffic
3	One filler flow with a flow of pre-existing traffic, with no priority difference
4	Five concurrent filler flows
5	Five concurrent filler flows with a flow of pre-existing traffic

4.2. Results

The results of the tests indicated the overall suitability of TCP-Background for use with filler traffic instead of TCP SACK. The data in the "Throughput" and "Sharing" columns of Tables 1–5 show the performance improvements leading us to choose TCP-Background. In order ensure that the filler traffic was indeed transparent to the pre-existing traffic, the throughput, delay, and drops of the pre-existing traffic were compared to its performance with filler added (Tests 2 and 5 from Table 6), shown in Table 7. Using the priority differentiation scheme, the loss of throughput and additional delay were minimal. When there was no differentiation between pre-existing and filler traffic (Test 3 in Table 6), the throughput had a small but noticeable decrease, but the delay increased significantly as shown in Table 8. It is important to note here that the pre-existing traffic is not adaptive because it is a static trace file. If it had been adaptive, an increase in drops in each of the pre-existing flows would result in lower throughput. Thus, the use of these filler protocols is not recommended in a non-differentiating environment.

Table 7. Transparency of filler traffic to pre-existing traffic

Pre-existing with	Average total delay	Dropped packets	Throughput
no filler	0.020396	0	179.533
5 Reno filler	0.021005	0	179.533
5 NewReno filler	0.02101	0	179.533
5 SACK filler	0.021011	0	179.533
5 TCP-Background filler	0.021011	0	179.533

In a differentiating environment with pre-existing traffic, there is an advantage to be gained by using our more aggressive protocols, particularly TCP-Background. Sharing, often a weakness for the more aggressive protocols, is improved by the addition of pre-existing traffic, as found by comparing Test 1 against Test 2, and Test 4 against Test 5. This is because the pre-existing traffic will force drops in the filler traffic. These drops will be proportional to the amount of bandwidth each filler flow is using, so those taking a larger share will generally incur more drops (in some sense, this is similar to RED[7]). When the pre-existing traffic leaves, the filler traffic

Table 8. No priority differentiation

Protocol	Low priority throughput	Pre-existing throughput	Pre-existing delay
pre-existing	-	179.533	0.020396
Reno	913.704	176.338	0.035767
NewReno	1009.77	175.606	0.039063
SACK	1017.1	175.489	0.039369
TCP-Background	1017.27	175.481	0.03941

flows will all bounce back at the same rate, with the threshold factor of the 0.95 power on TCP-Background letting the flows that previously had a smaller share to slow-start to a relatively higher point.

The alternative protocols, and TCP-Background in particular, also suffer less of a loss of throughput when pre-existing traffic is included (Tests 1, 2, 4, and 5). The aggressive behavior of our protocols allows them to use more of the bandwidth right after a spike in pre-existing traffic than standard protocols. As shown in Tables 9 and 10, TCP-Background, along with other protocols, performs slightly better than SACK in throughput and sharing. Due to random variation, TCP will achieve better performance over a longer period of time. However, to ensure that TCP-Background consistently got better sharing than SACK, the two protocols were tested over different periods of time. Figure 3 shows TCP-Background's improvement over SACK. In more extreme circumstances, TCP-Background fared better than SACK by a far greater margin. When the bottleneck link delay was 5 seconds TCP-Background got 17% better throughput than SACK, as shown by Table 11. Throughout all the tests with pre-existing traffic, TCP-Background got equivalent or better throughput and sharing when compared with SACK.

Table 9. Throughput of one filler flow with pre-existing

Protocol	Throughput
Reno	819.94
NewReno	982.92
SACK	1006.96
TCP-Background	1012.95

Table 10. Sharing and throughput of 5 flows with pre-existing

Protocol	Sharing	Throughput
Reno	0.011173	998.384
NewReno	0.013684	1013.851
SACK	0.020655	1007.635
LowPri1	0.04226	1016.354
LowPri2.5	0.013887	1016.775
TCP-Background	0.00469	1016.86
LowPri4.1	0.009259	1016.834
LowPri5.2	0.015613	1017.026
LowPri6.2	0.014994	1015.53

Our testing did show however that without pre-existing traffic (Tests 1 and 4) TCP-Background did not outperform SACK in any meaningful way. Table 12 shows that it only gets about equivalent throughput to

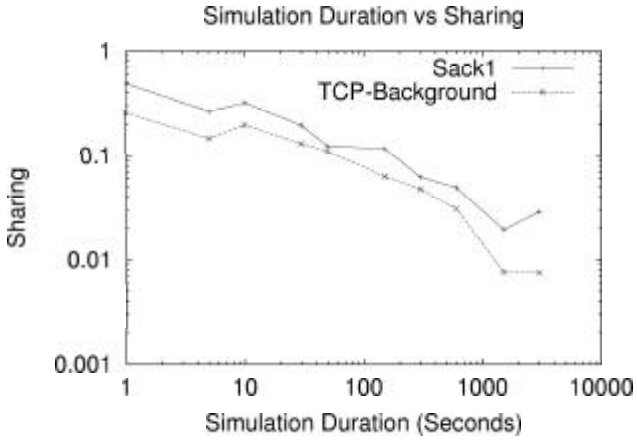

Figure 3. Sharing of SACK vs. TCP-Background over time of test

Table 11. Sharing and throughput of 5 filler flows with pre-existing at 5ms delay

Protocol	Sharing	Throughput
SACK	0.040149	725.147
LowPri1	0.044606	830.406
LowPri2.5	0.056237	754.212
TCP-Background	0.041731	853.394
LowPri4.1	0.027999	846.897
LowPri5.2	0.052473	852.992
LowPri6.2	0.023549	570.849

SACK when it is the only flow on a link. Table 13 shows that it does not gain any improvement in sharing or throughput when several filler flows are sharing a link.

TCP-Background is also only non-intrusive to flows which are given higher priority than it is. Using both TCP-Background and SACK for filler traffic on the same network would be grossly unfair to the SACK flows. Table 14 shows how bad the sharing was between two TCP-Background flows and three SACK flows. The poor sharing was caused by the SACK flows receiving much less than a fair share of the bandwidth. In the TCP-Background test, each TCP-Background flow got more than twice what each SACK flow got. So although TCP-Background flows can share well among themselves, they will not allow a more passive protocol to take a fair share of the bandwidth if they are operating in the same background environment.

Table 12. Throughput of flows in isolation

Protocol	Throughput
Reno	1187.97
NewReno	1187.89
SACK	1188.81
TCP-Background	1187.07

Table 13. Sharing and throughput of 5 flows

Protocol	Sharing	Throughput
Reno	0.02393	1189.913
NewReno	0.02225	1191.655
SACK	0.028639	1192.828
TCP-Background	0.02659	1190.096

Table 14. Sharing and throughput of 3 SACK flows and 2 low-priority flows

Protocol	Sharing	Throughput
SACK	0.020655	1007.635
LowPri1	0.860923	1010.941
LowPri2.5	0.36798	1014.96
TCP-Background	0.492539	1014.962
LowPri4.1	0.340802	1014.478
LowPri5.2	0.525471	1015.477
LowPri6.2	0.19204	1012.354

4.3. Heterogeneous Link Delay

The dumbbell topology was modified so that there were nodes with longer propagation delays than the others to determine if a node in this situation would be starved by our protocol. The results are listed in Table 15. All of the protocols shared poorly. The poor sharing was caused by the starvation of the longer delay nodes. TCP behavior in general tends to exhibit this starvation problem, but TCP-Background performed slightly better than SACK.

Table 15. Sharing and throughput of flows with varying delays

Protocol	Sharing	Throughput
SACK	0.415848	1011.326
TCP-Background	0.384118	1016.418
LowPri4.1	0.384105	1016.449
LowPri5.2	0.357822	1016.577

4.4. Bulk Best-Effort Traffic, Alternate Router Queuing, and LowPri3.7

Because the network traces we used do not adapt to the bandwidth properties, they do not provide a full simulation of all pre-existing traffic behavior. To see if additional pre-existing flows that were trying to use all of the excess bandwidth would have a different effect on the protocols, simulated flows sent at an average of half the link capacity in alternating on/off periods of various lengths. During "on" periods, the link was be fully utilized, and during "off" periods, the link was idle. These tests continued to use the pre-existing traffic in the traces in its normal capacity. As shown in Tables 16 and 17 TCP-Background got better throughput than SACK, and worse sharing.

All the previous simulations were run with the router configured to queue the lower priority filler traffic separately, so that incoming pre-existing traffic would not actually cause filler packets already received to be dropped. Because this might not always be possible however, tests were run with a router configuration where all the packets were stored in one queue. The separate queues allow filler packets already queued to continue on their

way after pre-existing traffic takes up the entire queue for some period of time. In a single queue environment, an entire congestion window of filler packets in the queue would be dropped if too many pre-existing packets arrived. Since separate queues avoid this problem, timeouts are much less common. Fewer timeouts are highly desirable in any router configuration that would be used with TCP-Background. Using the single queue version of the router made the filler flows perform significantly worse in the presence of a bulk pre-existing traffic flow. As shown in Tables 16-19, the overall throughput and sharing was much worse. TCP-Background gets much better throughput than SACK in this circumstance, making the best of a bad situation.

To try and improve performance in an environment where timeouts where more common we devised the protocol LowPri3.7. It is a more aggressive modification of TCP-Background that does not cut back as much after a timeout, and does not continue to cut back after repeated timeouts. This differs from another method of recovery from timeouts, TCP Fast Start.[3] Fast Start retains the values of the congestion window and round-trip time before a timeout, and uses those values when sending resumes. Repeated timeouts would occur if a pre-existing flow was taking the entire link for an extended period of time. Because timeouts were very rare, LowPri3.7 performed nearly the same as TCP-Background in the tests listed in Table 6. In the tests with the greedy pre-existing traffic flow, the results were inconclusive.

Table 16. 5 Flows of Filler with 250KB of Alpha every 0.5 seconds

Protocol	**Sharing**	**Throughput**
SACK	0.023447585	518.23
TCP-Background	0.096845284	530.6805
LowPri3.7	0.028829489	530.667

Table 17. 5 Flows of Filler with 1MB of Alpha every 2 seconds

Protocol	**Sharing**	**Throughput**
SACK	0.077526522	508.4624
TCP-Background	0.122666946	528.1125
LowPri3.7	0.142324186	526.5347

Table 18. 5 Flows of Filler with 250KB of Alpha every 0.5 seconds (single queue)

Protocol	**Sharing**	**Throughput**
SACK	0.043525941	367.4226
TCP-Background	0.195301044	426.5885
LowPri3.7	0.135467671	429.5329

5. RELATED WORK

Padmanabhan and Katz improved recovery from timeouts with TCP Fast Start,[3] but their work focused on more aggressive congestion control in order to speed up web transfers. Feldman, Alzidi, and Liberatore analyzed the effect of filler on pre-existing traffic.[1] Davison and Liberatore have shown that filler traffic can be used to speed up web transfers by pre-fetching.[8] Kuzmanovic and Knightly at Rice designed a client-based low-priority algorithm called TCP-LP[9] which used one-way delay measurements to determine congestion across a link.

Table 19. 5 Flows of Filler with 1MB of Alpha every 2 seconds (single queue)

Protocol	Sharing	Throughput
SACK	0.147244279	354.4448
TCP-Background	0.326246427	457.835
LowPri3.7	0.308405428	488.4238

6. CONCLUSIONS

In this paper we have demonstrated TCP-Background, which is more suited for use on lower priority filler traffic than existing protocols. It trades off average packet delay, which is unimportant to filler traffic, for increased goodput, and consequently, shorter times to complete an entire transfer. While implementation would be limited until a global standard was in place to indicate packets with a lower priority than best effort, we have shown that more aggressive protocols can be a more effective means of transport for filler traffic operating in the background.

ACKNOWLEDGMENTS

This research has been supported in part by an REU extension to NSF Grant ANI-0123929.

REFERENCES

1. A. Feldman, N. Alzidi, and V. Liberatore, "Effects of filler traffic in IP networks," in *Proceedings of the DIMACS Mini-Workshop on Quality of Service Issues in the Internet*, 2001.
2. M. Allman, "A web server's view of the transport layer," *ACM Computer Communication Review* **30**(5), October 2000.
3. V. N. Padmanabhan and R. H. Katz, "TCP Fast Start: A technique for speeding up web transfers," in *Proceedings of IEEE Globecom '98 Internet Mini-Conference*, 1998.
4. "Network Simulator, version 2." http://www.isi.edu/nsnam/ns.
5. "Harvard network traces and analyses." http://www.eecs.harvard.edu/net-traces/.
6. J. Semke, J. Mahdavi, and M. Mathis, "Automatic TCP buffer tuning," *ACM Computer Communications Review* **28**(4), October 1998.
7. S. Floyd and V. Jacobson, "Random Early Detection gateways for congestion avoidance," *IEEE/ACM Transactions on Networking* **1**(4), August 1993.
8. B. Davison and V. Liberatore, "Pushing politely: Improving web responsiveness one packet at a time," in *Proceedings of PAWS00*, 2000.
9. A. Kuzmanovic and E. W. Knightly, "TCP-LP: A distributed algorithm for low priority data transfer," in *Proceedings of IEEE INFOCOM 2003*, 2003.