TCP Nicer: Support for Hierarchical Background Transfers

Neil Alldrin and Alvin AuYoung
Department of Computer Science
University of California at San Diego
La Jolla, CA 92037
Email: {nalldrin, alvina}@cs.ucsd.edu

***Abstract*— Background computation is a well-established method of improving user-perceived performance in an interactive computer system. Garbage collection, disk prefetching and file system defragmentation are examples of commonly employed background tasks. With many of today's computing environments shifting from a desktop-centric to a distributed computing paradigm, it would be advantageous to have background computation in a distributed computing environment.**

We introduce TCP Nicer, a mechanism enabling multiple priority levels among TCP flows. TCP Nicer provides an abstraction for performing background computation in a distributed environment. In this design, lower priority flows minimally interfere with higher priority flows while also utilizing available bandwidth. Our results show that we can achieve 4 levels of priority with only sender-side modification.

I. Introduction

TCP Nice[5], [4], [6] is a modified version of TCP-Vegas enabling background flows that minimally interfere with foreground flows. This effectively provides two levels of flow priority: one for standard traffic and another for background traffic. This is useful for many applications such as data prefetching and peer-to-peer (P2P) networking where traffic often has low priority. As an example, suppose a system is running a P2P application. Without a notion of flow priority, the P2P transfers might adversely affect user-interactive transfers such as web browsing. Using TCP Nice, however, the P2P application could utilize spare bandwidth and minimize interference with the user's other transfers.

While two levels of flow priority is sufficient for many applications, it would be "nice" to have more. We present *TCP Nicer,* which extends TCP Nice to provide four levels of flow priority. This is useful for applications like hierarchical prefetching where more important prefetches can be assigned different priority levels. In the P2P framework, this could be used to prioritize file transfers, lookups, and other services. Additionally, end users could use such a mechanism to prioritize their applications. For example, a P2P application could be given lowest priority, followed by a FTP transfer, with other applications running in the foreground. This is similar to task priority in an operating system scheduler except it applies to network resources instead of cpu cycles.

II. Background

There exists a large body of work in networks research that allow hosts and routers to quickly and accurately detect network conditions. Most of this work has been motivated by the need for congestion control to maintain stability in the Internet. RED, ECN and XCP are examples of network protocols that attempt to leverage accurate network detection to perform congestion avoidance. The same mechanisms that allow for accurate congestion detection can also be utilized by clients wishing to provide more control over their own TCP connections.

A. TCP Vegas

TCP Vegas[2] is an example of a protocol that leverages improved congestion detection in an attempt to improve utilization during network congestion, but unlike most other approaches, it is implemented entirely on the end-host and does not require network infrastructure support. We describe the key features of TCP Vegas in order to provide background for the discussion of TCP Nice and TCP Nicer.

The goal of TCP Vegas is to allow TCP to increase bandwidth utilization and decrease packet losses. It differs from traditional implementations of TCP-Reno in its congestion avoidance mechanism, its treatment of slow-start, and retransmit policy. We discuss each of the following techniques briefly.

The congestion avoidance mechanism in TCP Vegas attempts to detect congestion before packet loss occurs. It maintains an estimate of expected throughput (based on outstanding packets and estimated minimum round-trip time) and the actual throughput. It calculates the difference between these amounts as *Diff* $=$ *Expected* $-$ *Actual* and will adjust its congestion window according to the value of *Diff*. If *Diff* $< \alpha$ then the congestion window is linearly increased. If *Diff* $> \beta$ then the congestion window is linearly decreased. They define α and β as tunable parameters. The congestion window is unchanged in other circumstances.

The slow start mechanism in traditional TCP is designed to probe the network for bandwidth availability by effectively doubling the size of its congestion window every round trip time (RTT). While this allows a TCP client to quickly estimate available bandwidth, it can lead to a large packet loss – on the order of half the size of the current congestion window. TCP Vegas seeks to avoid this loss by waiting two RTTs before doubling its congestion window. It uses this RTT to determine if it is approaching congestion. If so, it enters the standard increase and decrease algorithm.

Retransmissions in early implementations of TCP Reno suffered from lack of a fine-grained timing mechanism (the clock used had granularity on the order of 500 ms). To enable the retransmit mechanism to perform faster retransmits, TCP Vegas uses the receipt of other ACKs as an additional trigger to check if a timeout is likely to have occurred.

B. TCP Nice

TCP Nice borrows the congestion avoidance mechanism from TCP Vegas to achieve its goals. Modifications include a more sensitive congestion detector, faster response to congestion, and a stronger mechanism for flow back-off.

The TCP Nice congestion detection mechanism depends on an estimation of minimum RTT and maximum RTT. Two parameters, *fraction* and *threshold* , are defined as follows:

```
if $n > fraction \cdot cwnd$
    $cwnd \leftarrow cwnd/2$
```

Where n is the number of observed RTTs greater than $minRTT + maxRTT \cdot threshold$ within some time period. This means that if a fraction of the segments sent out in the last congestion window took longer than the minimum RTT by some amount, TCP Nice assumes congestion. When congestion isn't detected by this mechanism, TCP Nice falls back on the α and β congestion avoidance rules of TCP Vegas. Upon segment loss, it will use TCP Reno's congestion control rules.

Multiplicative decrease of the congestion window during congestion avoidance allows faster response to detected congestion. In addition to this, the congestion window is permitted to go below one (up to a limit of $\frac{1}{48}$). This policy provides a stronger mechanism for back-off and is what allows even multiple Nice flows to retain non-interference properties with respect to a foreground TCP flow.

While TCP Nice is based on the same mechanisms as TCP Vegas, its primary goal is to avoid interference with a foreground flow while also achieving a reasonable share of spare bandwidth. Despite the fact that these goals are in direct conflict, TCP Nice is able to achieve around 70% of spare bandwidth while interfering with the foreground flow by no more than 5%.

III. TCP Nicer

With TCP Nicer, we wish to implement a hierarchy of TCP Nice flows, with each pair maintaing the same non-interference properties as TCP Nice. The existing TCP Nice design was clean enough to allow extensions for this. In particular, the separation of the congestion detection mechanism, and the flow backoff schemes naturally divided our (initial) design into two parts: congestion detection and flow backoff.

A. Congestion Detection

Congestion detection is based on using the RTT to infer congestion in the network. The parameters used in the TCP Nice congestion detection scheme are *fraction* and *threshold*. Analysis by [5] indicates that the performance of TCP Nice is stable for *threshold* values less than 1 and between 0.1 and 0.9 for *fraction*. We repeated this analysis and arrived at similar findings. Holding all other parameters equal, we found that $frac > 0$ was the lower bound such that flows achieve reasonable throughput on an empty link. $frac = 0.9$ was the maximum value before a TCP Nicer flow would interfere with a TCP flow. Our use of the parameter $threshold$ differs from the definition above. We define it as the percentage difference between the current RTT and the min RTT for congestion to be signaled. We found that $threshold$ values between 1.1 and 1.2 lead to the most stable results.

B. Flow Backoff

The two mechanisms we use to manipulate the level of flow backoff were the amount of multiplicative decrease

TABLE I
PARAMETER VALUES

	$thresh$	mul_decr	add_incr	$frac$
Level 0	1.10	1.75	1.0	0.90
Level 1	1.20	0.50	0.2	0.30
Level 2	1.20	0.10	0.1	0.05

and additive increase performed by a TCP Nicer flow. We define multiplicative decrease to be the factor the existing congestion window is multiplied by during multiplicative decrease. For example, a $mul\_decrease$ of $\frac{1}{2}$ would cause the congestion window to be divided by 4 instead of 2, and a $mul\_decrease$ of 2 causes the congestion window to remain the same. $add\_increase$ is defined to be the number of segments added to the congestion window (the definition can easily be extended to reflect bytes intead). Intuitively, these two parameters are related. For example a flow with a large multiplicative decrease would probably not employ a large additive increase since it would severely oscillate. Moreover, a multiplicative decrease of 2 or greater does not make sense since it would no longer be a decrease. We found that setting $mul\_decrease$ less than 1.75 leads to proper backoff as does an additive increase of under 1.

C. Tuning the parameters

None of the parameters are completely independent of each other. In fact, we found the initial bounds we placed on the parameters could be broken with sufficient adjustment of other parameters. However, we used the bounds as a guide to our experimentation. We first tried to create an extremely aggressive flow (one that achieves maximum bandwidth very fast) that would minimally interfere with a foreground TCP flow, and repeated this process for successively less agressive flows. We also worked in the opposite direction: beginning with a pair of very passive flows, we increased the aggressiveness until we started to interfere with foreground flows. The parameter values we arrived at are summarized in table I. With these parameters we achieve three levels of background flow priority.

D. Damping Oscillations

In testing the effectiveness of particular sets of parameters, we encountered many surprising behaviors. The first occurred in slow start and in steady-state situations for low-priority background flows. For example, if a highly sensitive congestion scheme is coupled with a powerful backoff scheme, a flow can drive itself into severe oscillation preventing it from using available bandwidth, even on a network with no load. This is a side effect of queueing delay and the TCP Nice calculation of RTT values. On an empty network, a flow will observe a nearly constant RTT for the first several busrst of segment transmissions. However, due to queueing delay, the RTT will eventually drive up past the congestion threshold if the detection algorithm is too sensitive. This behavior repeats, causing oscillations and preventing sufficient network bandwidth utilization. Our solution to this is analogous to the TCP Vegas approach to slow start. We delay the back off mechanism for an RTT to avoid reacting to spurious latencies. We use this method to prevent particular cases of severe oscillations. We believe that by varying the number of RTTs to wait before reacting, we might be able to more finely tune the behavior of concurrent TCP Nicer flows of different priority levels. We leave this as future work.

IV. EXPERIMENTAL METHODOLOGY

The primary goal of the simulations is to show how well our priority levels work. All tests use *ns* 2.1b8a with a simple topology consisting of a single sender and a single bottleneck link. The bottleneck router is droptail FIFO, has a queue size of 50, bandwidth of 0.3Mbps, and latency of 40ms. Data packets are 512 bytes and traffic is generated from simulated FTP transfers.

¿From this point forward, foreground (FG) flows refer to TCP-Reno flows and background (BG) flows refer to TCP Nicer flows. Also, TCP Nicer flows are divided into three priority levels: level 0 is high priority, level 1 is medium priority, and level 2 is low priority.

V. RESULTS

Figure 1 shows the performance of a foreground flow against 19 high priority background flows. Before the background flows start, the foreground flow reaches full utilization; this is followed by a period of instability when the other flows begin. The foreground flow eventually recovers and again utilizes most of the bandwidth. The latency of the foreground flow is only affected during the period of instability where it jumps by about 13%. This shows that startup effects temporarily cause interference, but in practice this will probably have little impact since background flows will likely be long-lived.

Figure 2 compares a single foreground flow against 19 medium priority background flows. The startup effects are less significant than with high priority background

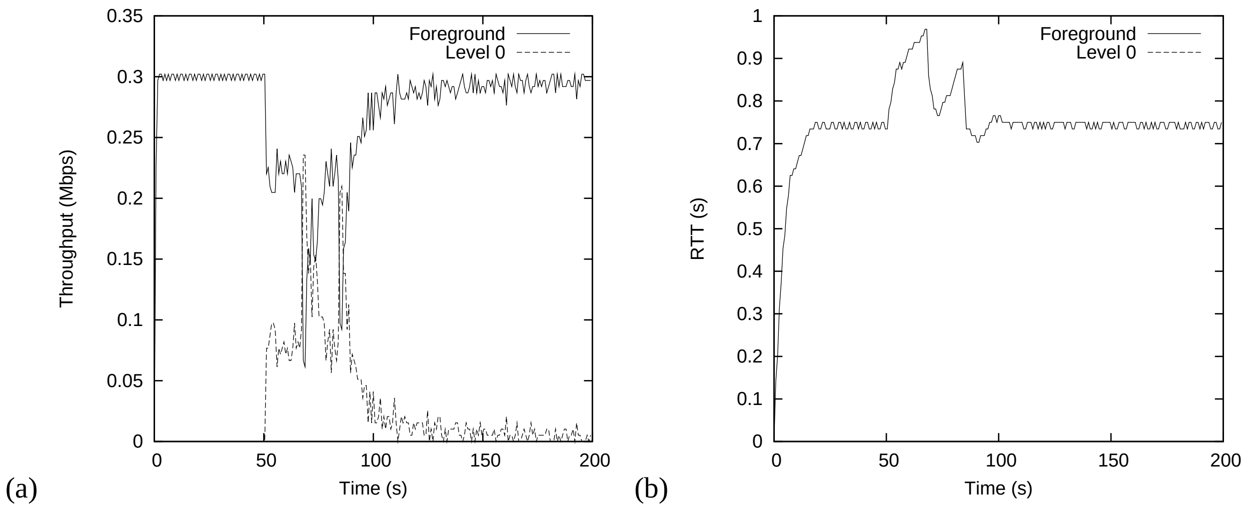

Fig. 1. Throughput (a) and latency (b) of a single TCP Reno FG flow compared to 19 priority level 0 BG flows (aggregated). All BG flows start at time = 50.

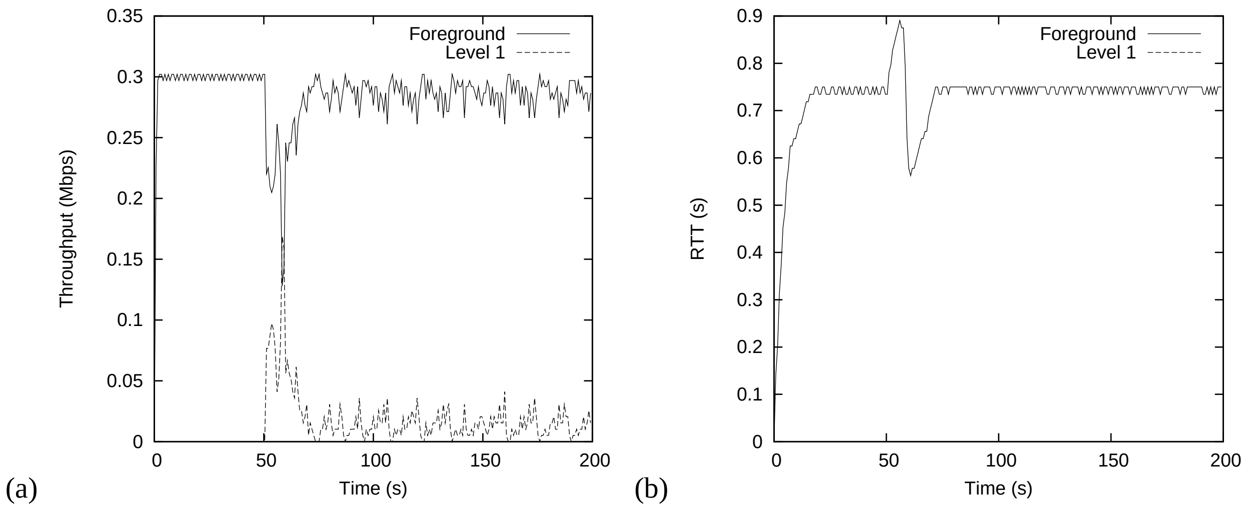

Fig. 2. Throughput (a) and latency (b) of a single TCP Reno FG flow compared to 19 priority level 1 BG flows (aggregated). All BG flows start at time = 50.

flows and near-optimal utilization is achieved by the foreground flow in steady-state.

Figure 3 shows a foreground flow with 19 low priority background flows. As expected, startup effects are further reduced and in steady-state the background flows have almost no effect on the foreground flow.

Figure 4 compares a single high priority flow to 19 medium priority flows. The medium priority flows cause roughly a 10% decrease in throughput and a negligable increase in latency. While this is fairly good, the variance of the bandwidth is high and could cause problems for real-time applications. This probably is not a major issue since real-time applications will likely not use background flows.

Figure 5 compares a single high priority flow to 19 low priority flows. As expected the interference is minimal.

Figure 6 shows that the low priority flows significantly interfere with medium priority flows. Even a single low priority flow causes the medium priority flow's throughput to periodically spike downward. With ten flows, the medium priority flow averages slightly under 50% of maximum utilization and with more flows the situation gets even worse. Latency (not shown) increases about 40% with ten flows. While these results are somewhat discouraging, a single medium priority flow still easily outperforms nine low priority flows.

Figure 7 shows the result of staggering four flows of different priority levels. The lowest priority flow

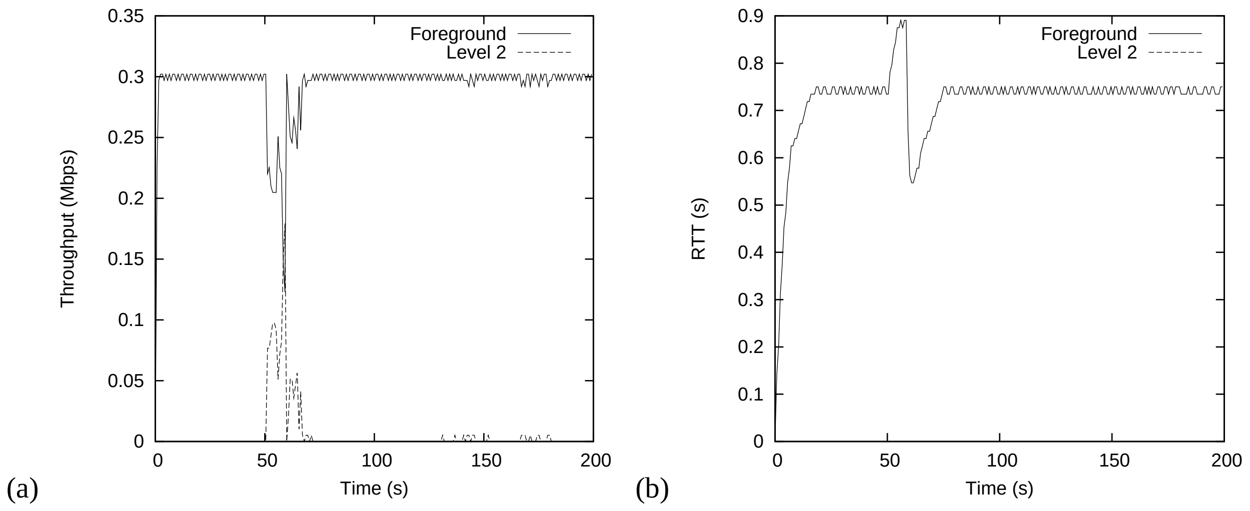

Fig. 3. Throughput (a) and latency (b) of a single TCP Reno FG flow compared to 19 priority level 2 BG flows (aggregated). All BG flows start at time = 50.

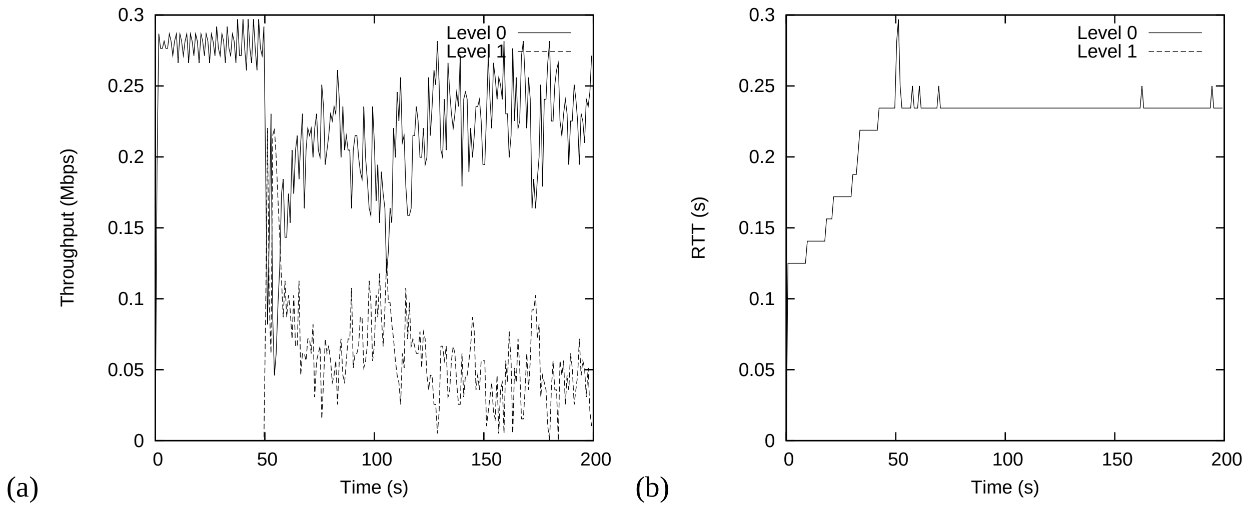

Fig. 4. Throughput (a) and latency (b) of a single priority level 0 BG flow compared to 19 priority level 1 BG flows (aggregated). All level 1 flows start at time = 50.

starts first and is taken over when the medium priority flow starts. Similarly, the high priority and foreground transfers take over when they start.

VI. Conclusions

We have shown that TCP Nice can be extended to support four priority levels. However, our results indicate that extending beyond this many levels will require more advanced techniques. We highlight suggestions for some of these techniques below.

Our scheme of performing a variable delay for flow backoff provided a rough approximation to percentile RTT estimates. It allows flows to ignore spurious RTTs that might cause delays. This is particularly effective for lower priority flows which have large backoff values. The authors of TCP Nice specifically mention improving the robustness of RTT estimates by calculating first and ninety-ninth percentile values. It isn't clear whether or not the potential gain warrants the added complexity of this scheme, in light of past studies [1] and the presence of simpler approximation schemes for mean and mode values.

It is clear from our results that particular combinations of foreground-background flows work better than others. We would like to investigate the possibility of enabling a TCP Nicer flow to dynamically adjust its priority or even its parameters within a priority level. This would be valuable to maintaining more stable results and also be

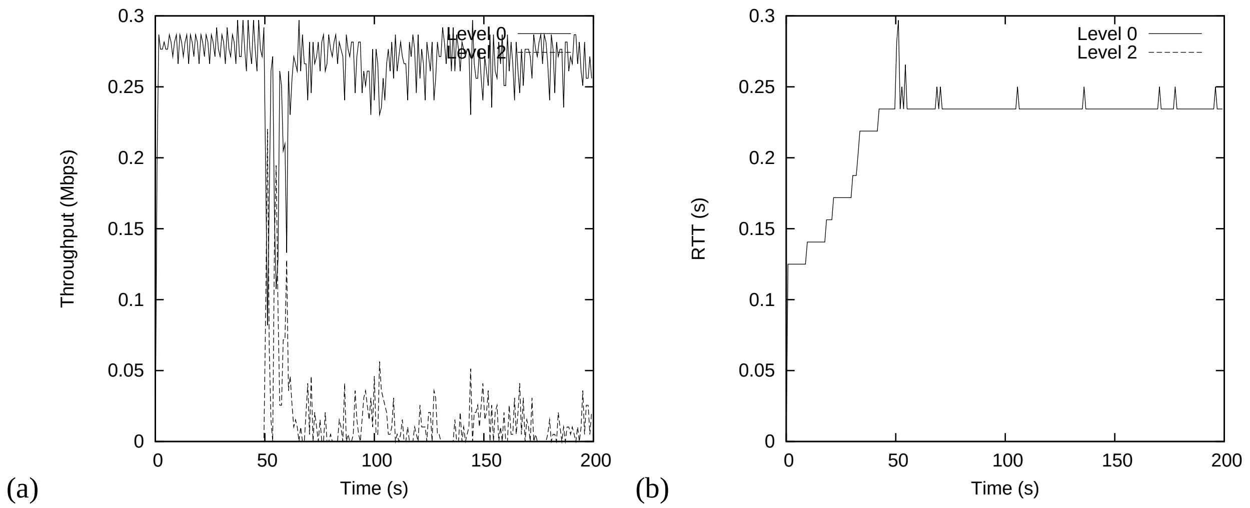

Fig. 5. Throughput (a) and latency (b) of a single priority level 0 BG flow compared to 19 priority level 2 BG flows (aggregated). All level 2 flows start at time = 50.

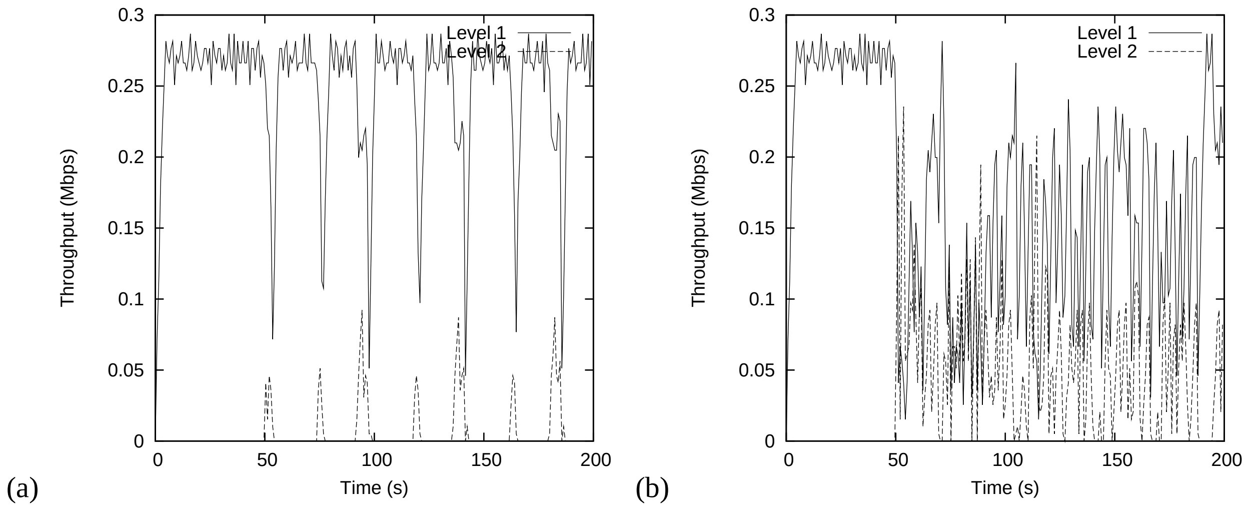

Fig. 6. Throughput of a single priority level 1 BG flow compared to (a) a single low priority BG flow and (b) nine low priority BG flows (aggregated). All level 2 flows start at time = 50.

of value to an end-user, who might want to dynamically reassign TCP Nicer priorities based on the status of the network or other active flows.

We would like to implement these changes into the TCP code of the Linux kernel and measure the real-world performance of TCP Nicer. It would be interesting to compare its performance to a proprietary scheme like Microsoft's Background Intelligent Transfer Service (BITS)[3], where they are seemingly able to offer background transfers without directly modifying the Operating System's internals.

We would also like to note that the notion of using background transfers has several limitations. First of all it is sensitive to other traffic on the network. It is unclear how one client's TCP Nicer background transfers might affect another client's transfers that share some of the same links. Also, this technique is limited to providing control on the sender-side as it cannot be assumed that other hosts will cooperate. While receivers could probably be made to support prioritized inbound flows, TCP Nicer does not address this.

REFERENCES

[1] A. Acharya and J. Saltz. A study of internet round-trip delay. Technical report, University of Maryland, 1996.

[2] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. Tcp vegas: New techniques for congestion detection and avoidance. *In Proceedings of the SIGCOMM Symposium*, pages 24–35, 1994.

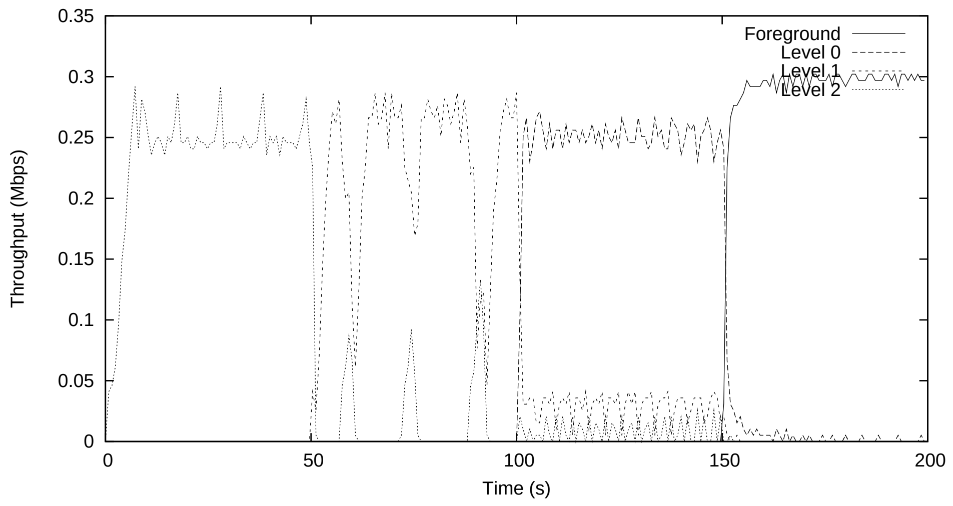

Fig. 7. Comparison of four flows of varying priority. The low priority flow starts at 0, the medium priority flow starts at 50, the high priority flow starts at 100, and the foreground flow starts at 150.

[3] Microsoft. Windows server 2003 : Background intelligent transfer service. Technical report, Microsoft Corporation, 2002.

[4] A. Venkataramani, R. Kokku, and M. Dahlin. Operating system support for massive replication systems. *In Proceedings of the Tenth ACM SIGOPS European Workshop*, 2002.

[5] A. Venkataramani, R. Kokku, and M. Dahlin. System support for background replication. *ODSI*, 2002.

[6] A. Venkataramani, R. Kokku, and M. Dahlin. Tcp nice: A mechanism for background transfers. *OSDI*, 2002.