System Support for Background Replication

Arun Venkataramani   Ravi Kokku   Mike Dahlin

Department of Computer Sciences
University of Texas at Austin

Abstract

Application performance and availability can be improved by aggressive *background replication* – distributing data across a network to where it is needed before it is requested. However, given the rapid fluctuations of available network bandwidth and changing resource costs due to technology trends, hand tuning applications risks (1) complicating applications, (2) being too aggressive and interfering with other applications, and (3) being too timid and not gaining the benefits of background replication. Our goal is for the operating system to manage network resources in order to provide a simple abstraction of zero-cost background replication. Our system, TCP Nice, provably bounds the interference inflicted by background flows on foreground flows. And our microbenchmarks and case study applications suggest that in practice it interferes little with foreground flows while reaping a large fraction of spare network bandwidth and simplifying application construction and deployment. For example in one microbenchmark, when demand flows consume half of the available bandwidth, Nice flows consume 50%-80% of the remaining bandwidth without increasing demand packets' average latencies by more than 5%; if the same background flows are transmitted with TCP Reno, they can hurt foreground latencies by up to two orders of magnitude. In our prefetching case study application, aggressive prefetching improves demand performance by a factor of three when Nice manages resources; but the same prefetching hurts demand performance by a factor of six under standard network congestion control.

1 Introduction

Application performance and availability can be improved by aggressive *background replication* – distributing data across a network to where it is needed before it is requested. A broad range of applications and services can trade increased network bandwidth consumption and disk space for improved service latency [19, 21, 28, 34, 39, 48], improved availablity [15, 53], increased scalability [7], or support for mobility [30, 42, 46]. Many of these services have potentially unlimited bandwidth demands where incrementally more bandwidth consumption provides incrementally better service. For example, a web prefetching system can improve its hit rate by fetching objects from a virtually unlimited collection of objects that have non-zero probability of access [12, 14] or by updating cached copies more frequently as data change [17, 48, 47]; similarly, in peer to peer replication systems, Yu and Vahdat suggest a direct trade-off between the aggressiveness of update propagation and service availability [53]. Technology trends suggest that "wasting" bandwidth and storage to improve latency and availability will become increasingly attractive in the future: per-byte network transport costs and disk storage costs are low and have been improving at 80-100% per year [13, 20, 38]; conversely network availability [41, 54, 15] and network latencies improve slowly, and long latencies and failures waste human time.

Current operating systems and networks do not provide good support for aggressive background replication. In particular, because background transfers compete with foreground requests, aggressive replication can hurt overall performance and availability by increasing network congestion. Applications must therefore carefully balance the benefits of replication against the risk of both *self-interference*, where applications hurt their own performance, and *cross-interference*, where applications hurt other applications' performance. Often, applications attempt to achieve this balance by setting "magic numbers" (e.g., the prefetch threshold in prefetching algorithms [21, 28]) that have little obvious relationship to system goals (e.g., availability or latency) or constraints (e.g., current spare network bandwidth).

Our goal is for the operating system to manage network resources in order to provide a simple abstraction of zero-cost background replication. A self-tuning background replication layer will enable new classes of applications by (1) simplifying applications, (2) reducing the risk of being too aggressive, and (3) making it easier to reap a large fraction of spare bandwidth to gain the advantages of background replication. Self-tuning resource management seems essential for coping with network conditions that change significantly over periods of seconds (e.g., changing congestion [54]), hours (e.g., diurnal patterns), and months (e.g., technology trends [13, 38]). We focus on managing network resources rather than processors, disks, and memory both because other work has provided suitable end-station schedulers for these local resources [14, 26, 35, 40, 45] and because networks are shared across applications, users, and organizations and therefore pose the most critical resource management challenge to aggressive background replication.

Our system, TCP Nice, dramatically reduces the interference inflicted by background flows on foreground flows. It does so by modifying TCP congestion control to be more sensitive to congestion than traditional protocols such as TCP-Reno [31] or TCP Vegas [11] by detecting congestion earlier, reacting to it more aggressively, and by allowing much smaller effective minimum congestion windows. Although each of these changes is simple, the combination is carefully constructed to provably bound the interference of background flows on foreground flows while still achieving reasonable throughput in practice. Our Linux implementation of Nice allows senders to select Nice or standard Reno congestion control on a connection-by-connection basis, and it requires no modifications at the receiver.

Our goals are to minimize damage to foreground flows while reaping a significant fraction of available spare network capacity. We evaluate Nice against these goals using theory, microbenchmarks, and application case studies.

Because our first goal is to avoid interference regardless of network conditions or application aggressiveness, our protocol must rest on a sound theoretical basis. In Section 3, we argue that our protocol is always less aggressive than Reno, and we prove under a simplified network model that Nice flows interfere with Reno flows' bandwidth by a factor that falls exponentially with the size of the buffer at the bottleneck router independent of the number of Nice flows in the network. In our analysis, all three features described above are essential for bounding interference.

Our microbenchmarks comprise both *ns* [1] simulations to stress test the protocol and Internet measurements to examine the system's behavior under realistic conditions. Our simulation results in Section 4 indicate that Nice avoids interfering with Reno or Vegas flows across a wide range of background transfer loads and spare network capacity situations. For example, when there are 16 continuously backlogged background flows competing with demand HTTP cross traffic that average 12 open connections and that consume half of the bottleneck bandwidth, the background flows slow down the average demand packet by less than 5% and they reap over 70% of the spare network bandwidth. Conversely, 16 backlogged Reno (or Vegas) flows slow demand requests by more than an order of magnitude.

Our Internet microbenchmarks in Section 5 measure the performance of simultaneous foreground and background transfers across a variety of Internet links. We find that background flows cause little interference to foreground traffic: the foreground flows' average latency and bandwidth are little changed between when foreground flows compete with background flows and when they do not. Furthermore, we find that there is sufficient spare capacity that background flows reap significant amounts of bandwidth throughout the day. For example, during most hours Nice flows between London England and Austin Texas averaged more than 80% of the bandwidth achieved by Reno flows; during the worst hour observed they still saw more than 30% of the Reno flows' bandwidth.

Finally, our case study applications seek to examine the end-to-end effectiveness, the simplicity, and the usefulness of Nice. We examine two services. First, we implement a HTTP prefetching client and server and use Nice to regulate the aggressiveness of prefetching. Second, we study the Tivoli Data Exchange [4] system for replicating data across large numbers of hosts. In both cases, Nice allows us to (1) simplify the application by eliminating magic numbers, (2) reduce the risk of interfering with demand transfers, and (3) improve the effectiveness of background transfers by using significant amounts of bandwidth when spare capacity exists. For example, in our prefetching case study, when applications prefetch aggressively, they can improve their performancy by a factor of 3 when they use Nice, but if they prefetch using TCP-Reno instead, they overwhelm the network and increase total demand response times by more than a factor of six.

The primary limitation of our analysis is that we evaluate our system when competing against Reno and Vegas TCP flows, but we do not systematically evaluate it against other congestion control protocols such as equation-based [24] or rate-based [43]. Our protocol is strictly less aggressive than Reno, and we expect that it will cause little interference with other demand flows, but future work is needed to provide evidence to support this assertion. A second concern is incentive compatibility: will users use low priority flows for background traffic when they could use high priority flows instead? We observe that most of the "aggressive replication" applications cited above do, in fact, voluntarily limit their aggressiveness by, for example, prefetching only objects whose priority of use exceeds a threshold [21, 48]. Two factors may account for this phenomenon. First, good engineers may consider the social costs of background transfers and therefore be conservative in their demands. Second, most users have an incentive to at least avoid self-interference where a user's background traffic interferes with that user's foreground traffic from the same or different application. We thus believe that Nice is a useful tool for both responsible and selfish engineers and users.

The rest of this paper proceeds as follows. Section 2 describes the Nice congestion control algorithm. Sections 3, 4, and 5 describe our analytic results, NS microbenchmark results, and Internet measurement results. Section 6 describes our experience with case study applications. Finally, Section 7 puts this work in context with related work, and Section 8 presents our conclusions.

2 Design and Implementation

In designing our system, we seek to balance two conflicting goals. An ideal system would (1) cause no interference to demand transfers and (2) consume 100% of available spare bandwidth. In order to provide a simple and safe abstraction to applications, we emphasize the former goal and will be satisfied if our protocol makes use of a significant fraction of spare bandwidth. Although it is easy for an adversary to construct scenarios where Nice does not get any throughput in spite of there being sufficient spare capacity in the network, our experiments confirm that in practice, Nice obtains a significant fraction of the throughput of Reno or Vegas when there is spare capacity in the network.

2.1 Background: Existing Algorithms

Congestion control mechanisms in existing transmission protocols are composed of a *congestion signal* and a *reaction policy*. The congestion control algorithms in popular variants of TCP (Reno, NewReno, Tahoe, SACK) use packet loss as a congestion signal. In steady state, the reaction policy uses additive increase and multiplicative decrease (AIMD) in which the sending rate is controlled by a congestion window that is multiplicatively decreased by a factor of two upon a packet drop and is increased by one per window of data acknowledged. The AIMD framework is fundamental to the robustness of the Internet [31, 16].

However, with respect to our goal of minimizing interference, this congestion signal–a packet loss–arrives too late to avoid damaging other flows. In particular, overflowing a buffer (or filling a RED router enough to cause it to start dropping packets) may trigger losses in other flows, forcing them to back off multiplicatively and lose throughput.

In order to detect incipient congestion due to interference we monitor round-trip delays of packets and use increasing round-trip delays as a signal of congestion. In this respect, we draw inspiration from TCP Vegas [11], a protocol that differs from TCP-Reno in its congestion avoidance phase. By monitoring round-trip delyas, each Vegas flow tries to keep between α (typically 1) and β (typically 3) packets buffered at the bottleneck router. If fewer than α packets are queued, Vegas increases the window by one per window of data acknowledged. If more than β packets are queued, the algorithm decreases the window by one per window of data acknowledged. Vegas does this estimation as follows:

```
$E = \frac{W}{minRTT}$            // Expected throughput

$A = \frac{W}{observedRTT}$       // Actual throughput

$Diff = E - A$

if($Diff < \frac{\alpha}{minRTT}$)
    $W = W + 1$
else if($Diff > \frac{\beta}{minRTT}$)
    $W = W - 1$
```

Bounding the difference between the actual and expected throughput translates to maintaining between α and β packets in the bottleneck router. Although Vegas seems a promising candidate protocol for background flows, it has some drawbacks:

1. Vegas has been designed to compete for throughput approximately fairly with Reno.
2. Vegas attempts to back off when the number of queued packets from its flows increase, but it does not necessarily back off when the number of packets enqueued by other flows increases.
3. Each Vegas flow tries to keep 1 to 3 packets in the bottleneck queue, hence a collection of background flows could cause significant interference.

Note that even setting α and β to very small values does not prevent Vegas from interfering with cross traffic. The linear decrease on the "$Diff > \beta$" trigger is not responsive enough to keep from interfering with other flows. We confirm this intuition using simulations and real-world experiments, and it also follows as a conclusion from the theoretical analysis.

2.2 TCP Nice

The Nice extension adds three components to Vegas: first, a more sensitive congestion detector; second, multiplicative reduction in response to increasing round trip times; and third, the ability to reduce the congestion window below one. These additions are simple, but our analysis and experiments demonstrate that the omission of any of them would fundamentally increase the interference caused by background flows.

A Nice flow monitors round-trip delays, estimates the total queue size at the bottleneck router, and signals congestion when this total queue size exceeds a fraction of the estimated maximum queue capacity. Nice uses $minRTT$, the minimum observed round trip time, as the estimate of the round trip time when queues are empty, and it uses $maxRTT$ as an estimate of the round trip time when the bottleneck queue is full. If more than *fraction* of the packets Nice sends during a RTT window encounter delays exceeding $minRTT + (maxRTT - minRTT) \cdot threshold$, our detector signals congestion. After the first rount-trip delay estimate, maxRTT is initialized to $2 \cdot minRTT$. As in Vegas, such running measures have their limitations – for example if the network is in a state of persistent congestion a bad estimate of $minRTT$ is likely to be obtained. However, past studies [5, 44] have indicated that a good estimate of the minimum round-trip delay can typically be obtained in a short time; our experience supports this claim.

Route changes during a transfer can also contribute to inaccuracies in RTT estimates. However such changes are uncommon [41] and we speculate that they can be handled by maintaining exponentially decaying averages for $minRTT$ and $maxRTT$ estimates.

An early prototype signalled congestion when encountering delays exceeding $minRTT * (1 + threshold')$. Our new approach of expressing the threshold in terms of the difference between $minRTT$ and $maxRTT$ not only makes the problem more mathamatically tractable but also eliminates the need to hand-tune the threshold for different networks.

When a Nice flow signals congestion, it halves its current congestion window. In contrast Vegas reduces its window by one packet each round that encoungers long round trip times and only halves its window if packets are lost (falling back on Reno-like behavior.) The combination of more aggressive detection and more aggressive reaction may make it more difficult for Nice to maximize throughput, but our design goals lead us to minimize interference even at the potential cost of throughput. Our analysis bounding interference with demand flows relies on detecting when queues exceed a threshold and on backing off multiplicatively, and our experimental results show that even with these aggressively timid policies, we achieve reasonble throughput in practice.

Figure 1, discussed in detail in Section 3, shows a queue at a bottleneck router that routes Nice flows with a threshold t and fraction *f*. Round-trip delays of packets are indicative of the current queue size. The Nice congestion avoidance mechanism incorporating the *interference trigger* can be written as:

```
per ack operation:
    if($observedRTT > minRTT + t \cdot maxRTT$)
        numCong++;
per round operation:
    if($numCong > f \cdot W$)
        $W \to W/2$
    else {
        //.... Vegas congestion avoidance follows
    }
```

If the congestion condition does not trigger, Nice falls back on Vegas' congestion avoidance rules. If a packet is lost, Nice falls back on Reno's rules.

The final change to congestion control is to allow the window sizes to multiplicatively decrease below one if so dictated by the congestion trigger and response. In order to affect window sizes less than one, we send a packet out after waiting for the appropriate number of smoothed round-trip delays. In these circumstances we lose *ack-clocking*, but the flow continues to send at most as many packets into the network as it gets out. In this phase the packets act as network probes waiting for congestion to dissapate. By allowing the window to go below one, Nice retains the non-interference property even for a large number of flows. Both our analysis and our experiments confirm the importance of this feature: this optimization significantly reduces interference, particularly when testing against several background flows. A similar optimization has been suggested even for regular flows to handle cases when the number of flows starts to approach the bottleneck router buffer size [37].

2.3 Prototype Implementation

We implement a prototype Nice system[1] by extending an existing version of the Linux kernel that supports Vegas congestion avoidance. Like Vegas, we use microsecond resolution timers to monitor round-trip delays of packets to implement a congestion detector as described in section 2.2. In our implementation, we set the Vegas parameters α and β to 1 and 3 respectively.

The Linux TCP implementation maintains a minimum window size of two in order to avoid delayed acknowledgements by receivers that attempt to send one acknowledgement every two packets. In order to allow the congestion window to go to one or below one, we add a new timer that runs on a per-socket basis when the congestion window for the particular socket(flow) is below two. When in this phase, the flow waits for the appropriate number of RTTs before sending two packets into the network. Thus, a window of 1/16 means that the flow sends out two packets after waiting for 32 smoothed round-trip times. We limit the minimum window size to $1/48$ in our prototype.

Our congestion detector signals congestion when more than *fraction* $= 0.5$ packets during an RTT encounter delays exceeding $threshold = 0.2$. We discuss the sensitivity to $threshold$ in more detail in Section 3. The *fraction* does not enter directly into our analysis; our experimental studies in Section 4 indicate that the interference is relatively insensitive to the *fraction* parameter chosen. Since packets are sent in bursts, most packets in a round observe similar round-trip times. In the future we plan to study pacing packets [6] across a round in order to obtain better samples of prevailing round-trip delays.

Our prototype provides a simple API to designate a flow as a background flow through an option in the *setsockopt* system call. By default, flows are foreground flows.

3 Analysis

Experimental evidence alone is insufficient to allow us to make strong statements about Nice's non-interference properties for general network topologies, background flow workloads, and foreground flow workloads. We therefore

[1]An on-line demonstration and a source code distribution are available at http://www.cs.utexas.edu/users/arun/nice/

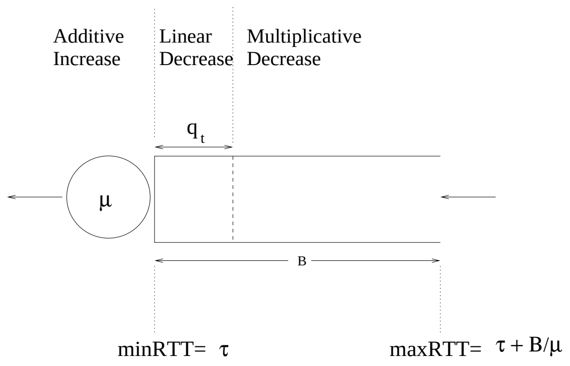


Figure 1: Nice Queue Dynamics

analyse it formally to bound the reduction in throughput that Nice can impose on foreground flows. Our primary result is that for long transfers, the reduction in the throughput of Reno flows is asymptotically bounded by a factor that falls exponentially with the maximum queue length of the bottleneck router irrespective of the number of Nice flows present. This analysis has also guided our design, allowing us to include features that are necessary for noninterference while excluding those that are not. Our experience with the prototype has supported the benefit of using theoretical analysis to guide our design: we encountered few surprises and required no topology- or workload-dependent tuning during our experimental effort.

Theoretical analysis of network protocols, of course, has limits. In general, as one abstracts away details to gain tractability or generality, one risks omitting important behaviors. For example, our formal analysis holds for long background flows, which are the target workload of our abstraction. But it also assumes long foreground Reno flows, which are clearly not the only cross-traffic of interest. Our formal analysis also assumes a simplified fluid approximation and synchronous network model, as described below. Finally, in our analysis, we abstract detection by assuming that at the end of each RTT epoch, a Nice sender accurately estimates the queue length during the previous epoch. Our implementation approximates this detection with packet round trip time measurements. Although these assumptions are restrictive, the insights gained in the analysis lead us to expect the protocol to work well in other circumstances, and our experimental evaluations complement the analysis by showing low interference in practice.

We use a simplified fluid approximation model of the network to help us model the interaction of multiple flows using separate congestion control algorithms. This model assumes infinitely small packets. We simplify the network itself to a source, destination, and a *single bottleneck*, namely a router that performs drop-tail queuing as shown in Figure 1. Let μ denote the service rate of the queue and B the available buffer size at the queue. Let τ be the round-trip delay of packets between the source and destination excluding all queuing delays. We consider a fixed number of connections, m following Reno and l following Nice, all of which attempt to transfer a single large file from the source to the destination. The connections are homogeneous, *i.e.* they experience the same propogation delay τ. Moreover, the connections are synchronized so that in the case of buffer overflow, all connections simultaneously detect a loss and multiply their window sizes by γ. Such synchronization phenomena have been reported in studies [6] and models assuming flow synchronization have been used in previous analyses [10]. We model only the congestion avoidance phase to analyze the steady-state behaviour.

Let $W_r(t)$ and $W_n(t)$ denote respectively the total number of outstanding Reno and Nice packets at time t. $W(t)$, the total window size, is $W_r(t) + W_n(t)$. We trace the dynamics of these window sizes by dividing the duration of the flows into *periods*. The end of a period and the beginning of the next is marked by a packet loss, at which time each flow reduces its window size by a factor of γ. $W(t) = \mu\tau + B$ just before a loss and $W(t) = (\mu\tau + B) \cdot \gamma$ just after. Let t_0 be the beginning of one such period after a loss.

We consider first the more interesting case when $W(t_0) = (\mu\tau + B)\gamma < \mu\tau$. For ease of analysis we assume that the "Vegas β" parameter for the Nice flows is 0, *i.e.* the Nice flows additively decrease upon observing round-trip times greater than τ. Consider the case when $m > l$. The window dynamics in any period can be split into four intervals as described below.

In interval $[t_0, t_1]$ $W(t)$ increases from $W(t_0)$ to $\mu\tau$, at which point the queue starts building. Both Reno and Nice flows increase linearly and their dynamics can be represented as:

$$\begin{cases} \frac{dW_r(t)}{dt} & = & \frac{m}{\tau} \\ \\ \frac{dW_n(t)}{dt} & = & \frac{l}{\tau} \end{cases} \tag{1}$$

Since the boundary conditions are given, the above equations completely determine $W_r(t), W_n(t)$ in this interval. The next interval $[t_1, t_2]$ is marked by additive increase of W_r, but additive decrease of W_n as the "*Diff* $> \beta$" rule triggers the underlying Vegas controls for the Nice flows. The round-trip time experienced by each packet when the queue is non-empty is given by $W(t)/\mu$, and the window dynamics during interval $[t_1, t_2]$ are as follows:

$$\begin{cases} \frac{dW_r(t)}{dt} & = & \frac{m\mu}{W(t)} \\ \\ \frac{dW_n(t)}{dt} & = & -\frac{l\mu}{W(t)} \end{cases} \tag{2}$$

The end of this interval is the time t_2 when $W(t_2) = \mu\tau + q_t$, where q_t is the threshold queue size that begins multiplicative backoff for Nice flows. However, the rate of decrease of $W_n(t)$ is bounded by the rate of increase of increase of

$W_r(t)$. Thus, the dynamics of interval $[t_2, t_3]$ are governed by:

$$\begin{cases} \frac{dW_r(t)}{dt} & = \frac{m\mu}{W(t)} \\ \frac{dW_n(t)}{dt} & = -min(\frac{W_n(t)\mu}{2 \cdot W(t)}, \frac{m\mu}{W(t)}) \end{cases} \tag{3}$$

The end of the above interval marks the completion of the period. At this point $W(t_3) = \mu\tau + B$, and right after, each flow decreases its window size by a factor of γ, thereby entering into the next period.

In order to quantify the interference experienced by Reno flows because of the presence of Nice flows we make the following observations.

Lemma 1: The values of W_r and W_n at the beginning of periods stabilize after several losses, so that each period thereafter is of a fixed duration T.

Lemma 2: The total amount of data sent by the Reno flows depends only on the initial and final values of W_r in a period.

Lemma 3: The length of a period T in a system with m Reno flows and a non-zero number of Nice flows is shorter than that of a system consisting of only the Reno flows.

The proofs of the above lemmas are not difficult and appear in Appendix A. Using the above lemmas it is straightforward to solve for for the residual number of outstanding Nice packets $W_n(t_0 + nT)$, which we denote by δ, at the end of a period, and show that it is bounded as follows:

$$\delta \leq \frac{m}{\gamma} e^{(-(1+\frac{(B-q_t)\gamma}{m}))} \tag{4}$$

Using this value of δ, we can compute W_r at the end of a period. The interference I characterizes the fractional loss in throughput experienced by Reno flows because of the presence of Nice flows.

Theorem 1: The interference I is given by

$$I \leq \frac{2\delta}{(\mu\tau + B)(1-\gamma^2)} \tag{5}$$

where δ is bounded as in (4). The proof appears in Appendix A.

Interference is primarily characterized by δ, that is as $\delta \rightarrow 0$, interference I approaches 0.

The derivation of δ indicates that the multiplicative decrease phase in the Nice protocol is crucial for reducing interference: it contributes to the inverse exponential term in δ. Intuititively, multiplicative decrease allows any number of Nice flows to get out of the way of additively increasing demand flows. This analysis also indicates that our particular mechanism of sampling round trip times from each packet is not fundamental. What is important is that the protocol detect when when queue lengths exceed q_t. The protocol is sensitive to $B - q_t$, which intuitively reflects the time that Nice must back off before packet losses occur. The value of δ also depends on the ratio $\frac{B-q_t}{m}$, which suggests that as the number of demand flows approaches the maximum queue size the non-interference property starts to break down. This breakdown is not surprising as each flow barely gets to maintain one packet in the queue and TCP Reno is known to behave anamolously under such circumstances [37]. We show in Appendix A that the above bound on I holds even for the case when $m \ll l$. Allowing window sizes to multiplicatively decrease below one is crucial in this proof.

When $(\mu\tau + B)\gamma > \mu\tau$, Nice flows do not get any throughput. However, this condition implies that the network is already saturated and does not have any spare capacity.

4 NS Controlled Tests

The goal of our simulation experiments is to validate our hypotheses in a controlled environment. In particular, we wish to i) test the non-interference property of Nice and ii) determine if Nice gets any useful bandwidth for the workloads considered. By using controlled *ns* [1] simulations in this phase of the study we can stress test the system by varying network configurations and load to extreme values. We can also systematically compare the Nice algorithm against others. Overall, the experiments support our theses:

- Nice flows cause almost no interference irrespective of the number of flows.
- Nice gets a significant fraction of the available spare bandwidth.
- Nice outperforms other existing protocols, including Reno, Vegas, and Vegas with reduced α and β parameters.

4.1 Methodology

We use *ns* 2.1b8a for all of our simulation experiments. The topology used is a bar-bell one in which N TCP senders transmit through a shared bottleneck link L to an equal number of receivers. The router connecting the senders to L becomes the bottleneck queue. Routers perform drop-tail FIFO queueing and the buffer size is set to 50 packets, each of which is 1024 bytes in size. The propogation delay is set to 50ms. We vary the capacity of the link in order to simulate different amounts of spare capacity.

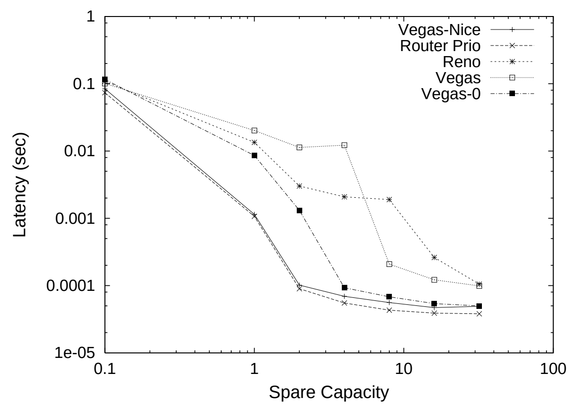


Figure 2: Spare capacity vs Latency

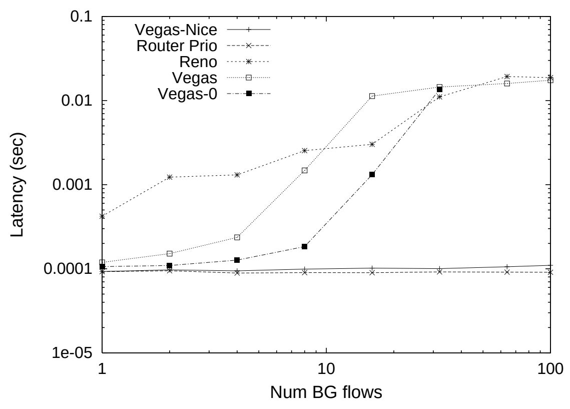


Figure 3: Number of BG flows vs Latency

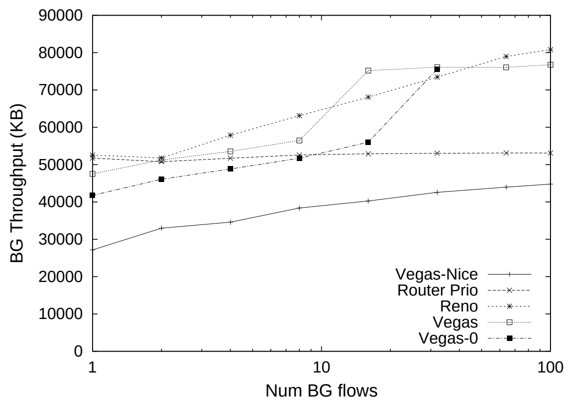


Figure 4: Number of BG flows vs BG throughput

We use a 15 minute section of a Squid proxy trace logged at UC Berkeley as the foreground traffic over L. The number of flows fluctuate as clients enter and leave the system as specified by the trace. On average there are about 12 active clients. In addition to this foreground load, we introduce permanently backlogged background flows. For the initial set of experiments we fix the bandwidth of the link to twice the average demand bandwidth of the trace. The primary metric we use to measure interference is the average round-trip latency of a foreground packet *i.e.*, the time between its being first sent and the receipt of the corresponding ack, inclusive of retransmissions. We use the total number of bytes transferred by the background flows as the measure of its utilization of spare capacity.

We compare the performance of the background protocol to several other strategies for sending background flows. First, we compare to router prioritization that services a background packet only if there are no queued foreground packets. Router prioritization is the ideal strategy for background flow transmission. In addition, we compare to Vegas($\alpha = 1, \beta = 3$), Reno, Vegas($\alpha = 0, \beta = 0$), and rate-limited Reno, which sets a maximum transmission bandwidth on each flow.

4.2 Results

Experiment 1: In this experiment we fix the number of background flows to 8 and vary the spare capacity, S. To achieve a spare capacity S, we set the bottleneck link bandwidth $L = (1 + S) \cdot$ *averageDemandBW*, where *averageDemandBW* is the total number of bytes transferred in the trace divided by the duration of the trace. Figure 2 plots the average latency of foreground packets as a function of the spare capacity in the network. Different lines represent different runs of the experiments using different protocols for background flows. It can be seen that the Nice is hardly distinguishable from the router prioritization whereas, the other protocols cause a significant increase in foreground latency. Note that the Y-axis is on a log scale, which means that in some cases Reno and Vegas increase foreground packets latencies by two orders of magnitude!

Experiment 2: Sensitivity to number of BG flows In this experiment we fix the capacity of the network to $S = 1$ (L twice the bandwidth needed by demand flows), and we vary the number of background flows. Figure 3 plots of the latency of foreground packets against the number of background flows. Even with 100 background Nice flows, the latency of foreground packets is hardly distinguishable from the ideal case when routers do strict prioritization. On the other hand Reno and Vegas background flows can cause demand flows' latencies to increase by by orders of magnitude. Figure 4 plots the number of bytes the background flows manage to get across. A single background flow reaps about half the spare bandwidth available under router prioritization; this background throughput improves with increasing number of background flows but remains below router prioritization. The difference is the price we pay for ensuring non-interference with an edge-based algorithm. Note that although Reno and Vegas obtain better throughputs, even for a small number of flows they shoot up beyond the router prioritization line, which means they steal bandwidth from foreground traffic.

We also ran these experiments where we do not allow Nice's

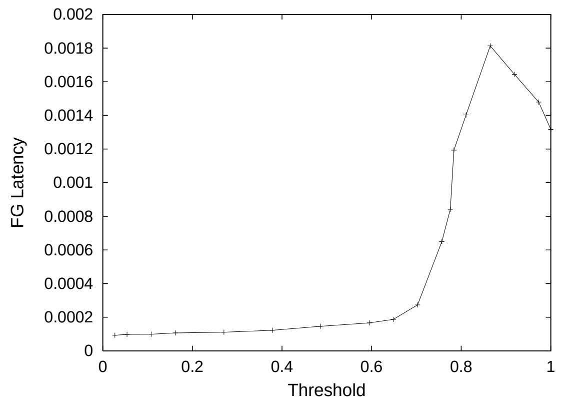


Figure 5: Threshold vs FG latency

congestion window to fall below 1 (graph omitted). In this case, when the number of background flows exceeds about 10, the latency of foreground flows begins to increase noticably; the increase was about a factor of two when $nBGFlows = 64$.

Experiment 3: Sensitivity to parameters In this experiment we trace the effect of the threshold and trigger fraction parameters described in Section 2.2. Figure 5 shows for the same trace as above, with $S = 1$ and 16 background flows, the latency of foreground packets as a function of the threshold. As expected, as the threshold value increases, the interference caused by Nice increases until the protocol finally reverts to Vegas behavior as the threshold approaches 1. It is interesting to note that there is large range of threshold values yielding low interference, which suggests that its value need not be manually tuned for each network. We examine the trigger fraction in the same way, and find no change in foreground latency as we vary this fraction from 0.1 to 0.9 (graph omitted).

Other results Due to space constraints, we state two other results here, but omit detailed discussions and graphs. The full discussion appears in Appendix B.

First, we also perform experiments with synthetically generated ON/OFF Pareto UDP traffic for the foreground, which is much burstier and less predictable than TCP foreground flows. We observe that Nice still causes lower interference than Reno or Vegas, but does not match router prioritization as closely. The utilization of spare capacity by Nice is also lower as compared to the trace workload case. This suggests that the benefits of Nice are reduced when traffic is unpredictable.

Second, we compare Nice to simple rate limited Reno flows. When the rate is tuned to approximate the spare capacity of the network, rate limiting performs well. Nice, however, outperforms rate limiting and does not require hand tuning.

5 Internet Microbenchmarks

This section presents a controlled experiment in which we evaluate our Nice implementation over a variety of Internet links. We seek to answer three questions. First, in a less controlled environment than our NS simulations, does Nice still avoid interference. Second, are there enough reasonably long periods of spare capacity on real links for Nice to reap reasonable throughput. Third, are any such periods of spare capacity spread throughout the day, or is the usefullness of background transfers restricted to nights and weekends?

Our experiments suggest that Nice works for a range of networks, including a modem, a cable modem, a transatlantic link, and a fast WAN. In particular, on these networks it appears that Nice avoids interfering with other flows and that it can achieve throughputs that are significant fractions of the throughputs that would be achieved by Reno throughout the day.

5.1 Methodology

Our measurement client program connects to a measurement server program at exponentially-distributed random intervals. At each connection time, the client chooses one of six actions: Reno/NULL, Nice/NULL, Reno/Reno, Reno/Nice, Reno/Reno8, Reno/Nice8.[2] Each action consists of a “primary transfer (denoted by the term left of the /) and zero or more “secondary transfers” (denoted by the term right of the /). Reno terms indicate flows using standard TCP-Reno congestion control. Nice terms indicate flows using Nice congestion control. For secondary transfers, NULL indicates actions that initiate no secondary transfers to compete with the primary transfer, and 8 indicates actions that initiate 8 (rather than the default 1) secondary transfers. The transfers are of large files whose sizes are chosen to require approximately 10 seconds for a single Reno flow to compete on the network under study.

In addition, during these actions and during periods of inactivity, clients ping our server to measure latency for individual packet transfers.

We position a server that supports Nice at UT Austin. We position clients (1) in Austin connected to the internet via a University of Texas 56.6K dial in modem bank (*modem*), (2) in Austin connected via a commercial ISP cable modem (*cable modem*), (3) in a commercial hosting center in London, England connected to multiple backbones including an OC12 and an OC3 to New York (*London*), and (4) at the University of Delaware, which connects to UT via an Abiline OC3 (*Delaware*). All machines run Linux. The server is a 450MHz Pentium II with 256MB of memory. The clients

[2]We also test standard Vegas in place of Reno for the large-transfer experiments and find that standard Vegas behaves essentially like Reno. These results are omitted due to space constraints.

range from 450-1000MHz and all have at least 256MB of memory. The experiment ran from Saturday May 11 2002 to Wednesday May 15 2002; we gathered approximately 50 probes per client/workload pair.

5.2 Results

Figure 6 summarizes the results of our large-transfer experiments. On each of the networks, the throughput of Nice/NULL is a significant fraction of that of Reno/NULL, suggesting that periods of spare capacity are often long enough for Nice to detect and make use of them. Second, we note that during Reno/Nice and Reno/Nice8 actions, the primary (Reno) flow achieves similar throughput to the throughput seen during the control Reno/NULL sessions. In particular, on a modem network, when Reno flows compete with a single Nice flow, they receive on average 97% of the average bandwidth they receive when there is no competing Nice flow. On a cable modem network, when Reno flows compete with eight Nice flows, they receive 97% of the bandwidth they would recieve alone. Conversely, Reno/Reno and Reno/Reno8 show the expected fair sharing of bandwidth among Reno flows, which reduces the bandwith achieved by the primary flow.

Figure 7 shows the hourly average bandwidth achived by the primary flow for the different combinations listed above. Our hypothesis is that Nice can achieve useful amounts of throughput throughout the day, and the data appear to support this statement.

6 Case Study Applications

6.1 HTTP Prefetching

Many studies have published promising results that suggest that prefetching (or pushing) content could significantly improve web cache hit rates by reducing compulsory and consistency misses [19, 21, 28, 29, 33, 34, 39, 48]. Few such systems have been deployed.

Typically, prefetching algorithms are tuned with a *threshold* parameter to balance the potential benefits of prefetching data against the bandwidth costs of fetching it and the storage cost of keeping it until its next use. An object is prefetched if the estimated probability that the object will be referenced before it is modified exceeds the threshold. Extending Gray and Shenoy's analysis of demand caching [27], Chandra calculates reasonable thresholds given network costs, disk costs, and human waiting time values and concludes that most algorithms in the literature have been far too conservative in setting their thresholds [13]. Furthermore, the 80-100% per year improvements in network [13, 38] and disk [20] capacity/cost mean that a value that is correct today may be off by an order of magnitude in 3-4 years.

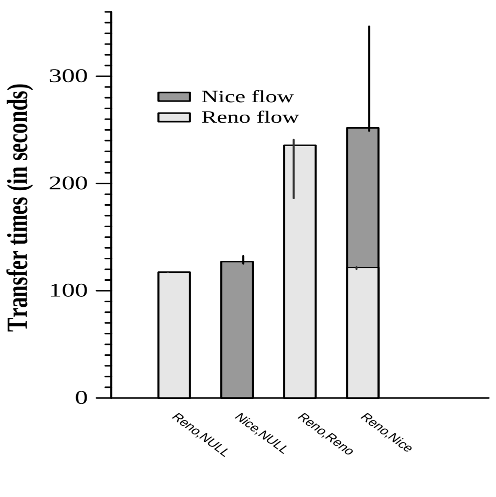
Our prototype protocol is similar to the one proposed by Padmanabhan and Mogul [39]: when serving requests, servers piggy back lists of suggested objects in a new HTTP reply header. Clients receiving a prediction list discard old predictions and then issue prefetch requests of objects from the new list. This division of labor allows servers to use global information and application-specific knowledge to predict access patterns, and it allows clients to filter requests through their caches to avoid repeatedly fetching an object.

To evaluate prefetching performance, we implement a standalone client that reads a trace of HTTP requests, simulates a local cache, and issues demand and prefetch requests. Our client is written in Java and pipelines requests across HTTTP/1.1 persistent connections [23]. To ensure that demand and prefetch requests use separate TCP connections, our server directs prefetch requests to a different port than demand requests. The disadvantage of this approach is that it does not fit with the standard HTTP caching model. Our modified client recognizes that URLs with two different ports are the same, but to simplify deployment we plan to modify our Nice implementation to allow a server to switch a single connection between Reno and Nice congestion control.

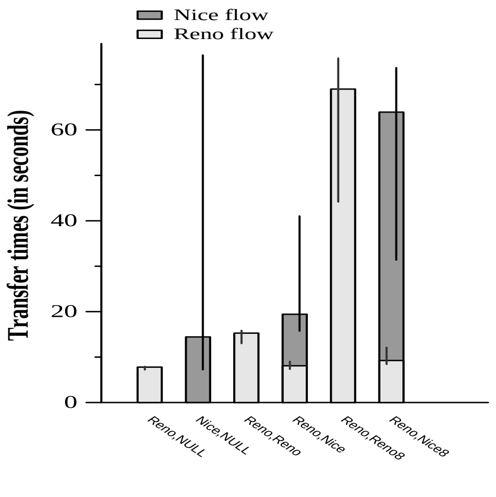
In this experiment, we generate predictions at clients (using knowledge from the trace to simulate server knowledge) rather than sending predictions across the network. This simplification allows us to use an unmodified Apache server. It slightly reduces network traffic for prefetching, but the impact on overall performance should be small. If servers have large prediction lists to send to clients, they can send small numbers of predictions in the headers of demand replies and "chain" the rest of the predictions in headers of prefetch replies.

We use Squid proxy traces from 9 regional proxies collected during January 2001 [51]. Each trace record includes the URL, the anonomized client IP address, and the time of the request. We study network interference near the server by examining subsets of the trace corresponding to a popular groups of related servers – *cnn* (e.g., cnn.com, www.cnn.com, cnnfn.com, etc.). Future work will examine cross interference near clients when prefetching from multiple unrelated servers.

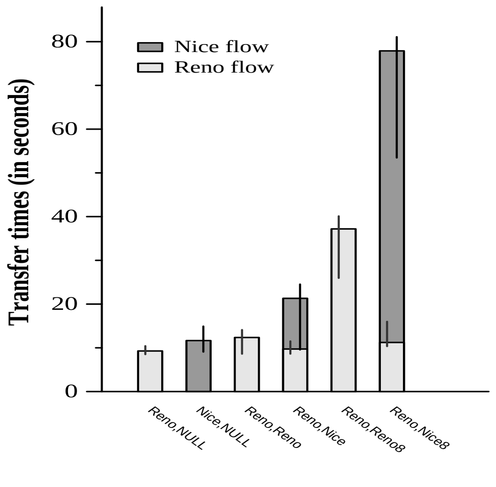
This study compares relative performance for different resource management algorithms for a given set of prefetching algorithms. It does not try to identify optimal prefetching algorithms; nor does it attempt to precisely quantify the absolute improvements available from prefetching. We use a simple prediction by partial matching (PPM) algorithm [18] PPM-n/w that uses a client's n most recent requests to the server group for non-image data to predict cachable (i.e., non-dynamically-generated) URLs that will appear during a subsequent window that ends after the w'th non-image request to the server group. This algorithm is limited because it uses neither link topology information [21] nor server-



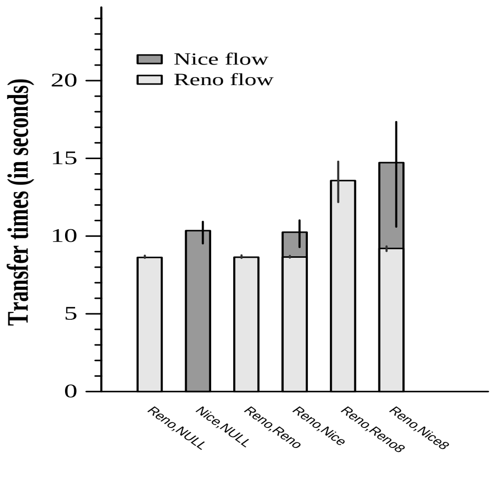
(a) modem



(b) cable modem



(c) London



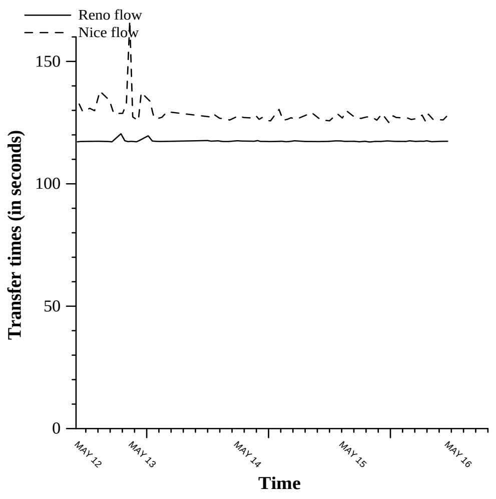
(d) Delaware

Figure 6: Large flow transfer performance. Each bar represents the average transfer time observed for the specified combination of primary/secondary transfers. Empty bars represent the average time for a Reno flow. Solid bars represent the average time for a Nice flow. The narrow lines are errorbars depicting the minimum and maximum values observed during multiple runs of each combination.

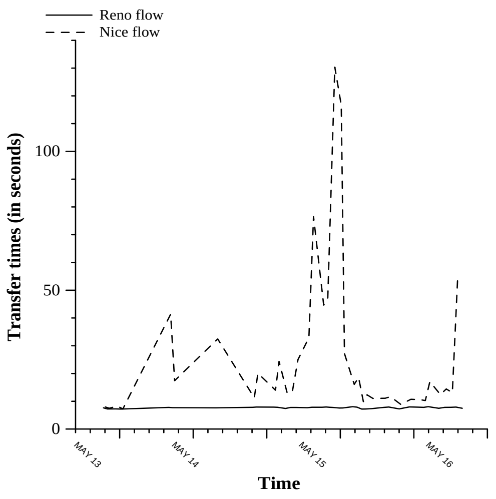
specific semantic knowledge. For simplicity, we assume that all non-dynamically-generated data (e.g., data not including a suffix indicating that a program was executed) are cachable and unchanging for the 1-hour duration of our experiments. Also, to allow us to vary demand, we break the trace into per-client, per-hour sections and treat each section as coming from a different client during the same simulated hour. Prefetching algorithms and server workloads are likely to vary widely; we believe that these assumptions yield a simple system that falls within the range prediction effectiveness that a simple service might experience.

We use two variations of our PPM-n/w algorithm. The *conservative* variation uses parameters similar to those found in the literature for HTTP prefetching. It uses $n = 2$, $w = 5$ and sets the prefetch threshold to 0.25 [21]. To prevent prefetch requests from interfering with demand requests, it pauses 1 second after a demand reply is received before issuing requests. The *aggressive* variation uses $n = 2$, $w = 10$, and truncates prefetch proposal lists with a threshold probability of 0.00001. It issues prefetches immediately after receiving them.

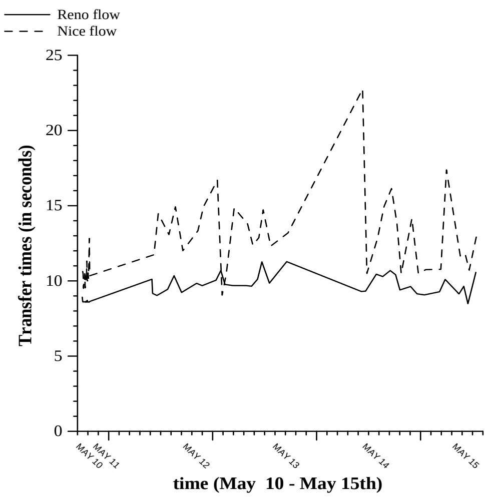
We use 2 client machines connected to a server machine via a cable modem. On each client machine, we run 8 virtual clients, each with a separate cache and separate HTTP/1.1 demand and prefetch connections to the server. In order for the demand traffic to consume about 10% of the cable modem bandwidth, we select the 6 busiest hours from the 30-Jan-2001 trace and divide trace clients from each hour



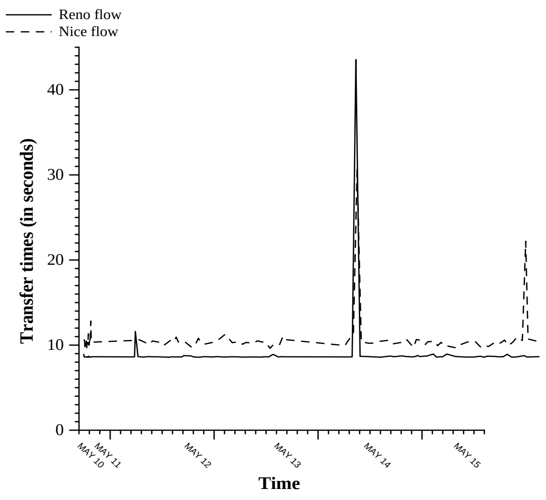
(a) modem



(b) cable modem



(c) London



(d) Delaware

Figure 7: Large flow transfer performance over time.

randomly across 4 of the virtual clients. In each of our seven trials, all the 16 virtual clients run the same prefetching algorithm: *none*, *conservative-Reno*, *aggressive-Reno*, *conservative-Nice*, *aggressive-Nice*.

Figures 8 shows the average cumulative demand transfer times perceived by the clients for all the files they are fetching on demand from the CNN server. In Figure 8, we note that when clients do conservative prefetching using either protocol - Nice or Reno - the latency reductions are comparable. However, when they start aggressively prefetching using Reno, the latency blows up by an order of magnitude. Clients using aggressive Nice prefetching however continue to see further latency reductions. The figure shows that Nice is very effective in using up the spare bandwidth for prefetching without affecting the demand requests.

Figure 9 represents the effect of prefetching over a modem (the setup is same as above except with the cable modem re-

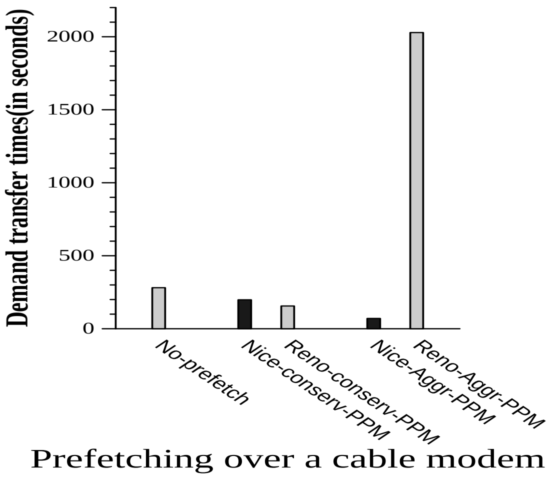


Figure 8: Average response time for prefetching for the cnn server-group.

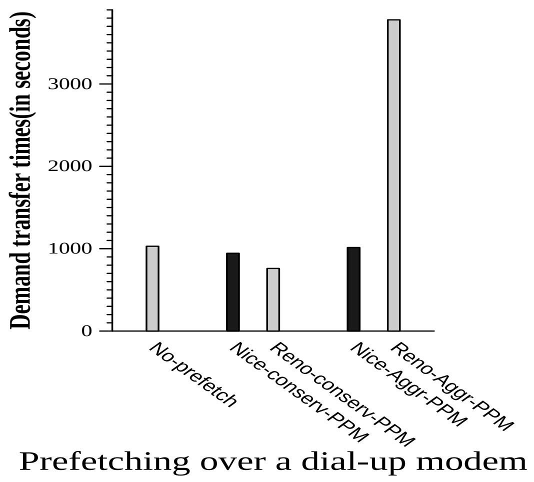


Figure 9: Average response time for prefetching for the cnn server-group.

placed by a modem), an environment where the amount of spare bandwidth available is not much. This figure shows that while the Reno and Nice protocols are comparable in benefits when doing conservative prefetching, aggressive prefetching using Reno hurts the clients significantly by increasing the latencies three-fold. Nice on the other hand, does not worsen the latency even though it does not gain much.

6.2 Tivoli Data Exchange

We study the Tivoli Data Exchange [4] system for replicating data across large numbers of hosts. This system distributes data and programs across thousands of client machines using a hierarchy of replication servers. Both non-interference and good throughput are important metrics. In particular, these data transfers should not interfere with interactive use of target machines. And because transfers may be large, may be time critical, and must go to a large number of clients using a modest number of simultaneous connections, each data transfer should complete as quickly as possible. For example, after Congress makes last minute changes to tax laws, the IRS must rapidly distribute new documentation to auditors. The system must cope with complex topologies including thousands of clients, LAN/WAN/modem links, and mobile clients whose bandwiths change drastically over time. The system currently uses two parameters at each replication server to tune the balance between non-interference and throughput. One parameter throttles the maximum rate that the server will send a single client; the other throttles the maximum total rate across all clients.

Choosing these rate limiting parameters requires some knowledge of network topology and may have to choose between overwhelming slow clients and slowing fast clients (e.g., distributing a 300MB Office application suite would nearly a day if throttled to use less than half a 56.6Kb/s mo-

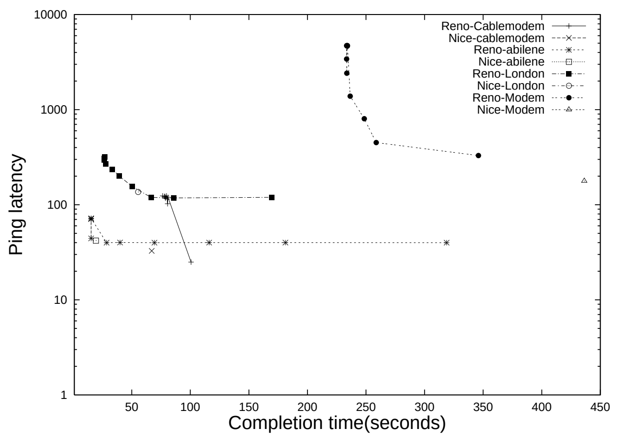


Figure 10: Tivoli: Nice picks the ideal send rate.

dem). One could imagine a more complex system that allows the maximum bandwidth to be specified on a per-client basis, but such a system would be complex to configure and maintain.

Nice provides an attractive self-tuning abstraction. Using it, a sender can just send at the maximum speed allowed by the connection. We are in the process of installing and benchmarking a series of experiments using the Tivoli software. Due to time constraints, we report preliminary results using a standalone server and client.

The server and clients are the same as in the Internet measurements described in Section 5. We initiate large transfers from the server and during that transfer measure the ping round trip time between the client and the server. When running Reno, we vary the client throttle parameter and leave the total server bandwidth limit to an effectively infinite value. When running nice, we set both the client and server bandwidth limits to effectively infinite values.

Figure 10 shows a plot of ping latencies (representative of interference) as a function of the completion time of transfers to clients over different networks. With Reno, completion times decrease with increasing throttle rates, but increase ping latencies as well. However Nice picks good sending rates without the need for any manual tuning, and ensure low ping latencies.

7 Related work

TCP congestion control has seen an enormous body of work since Jacobson's seminal paper on the topic [31]. This work seeks to maximize utilization of network capacity, to share the network fairly among flows, and to prevent pathological scenarios like congestion collapse. In contrast our primary goal is to ensure minimal interference with regular network traffic; though high utilization is important, it is a distinctly subordinate goal in our algorithm. Our algorithm is always less aggressive than standard TCP: it reacts the same way to losses and in addition, it reacts to increasing delays. Therefore, the work to ensure network stability under TCP applies to Nice as well.

Nice is designed on top of TCP Vegas [11]. We modify Vegas because it provides a framework for using round-trip times to control congestion windows. Other round-trip delay-based congestion control mechanisms have also been proposed [32, 49, 50] These approaches differ from Nice in that they attempt to compete fairly with TCP.

The GAIMD [52] and binomial [9] frameworks provide generalized families of AIMD congestion control algorithms to allow protocols to trade smoothness for responsiveness in a TCP-friendly manner. The parameters can also be tuned to make a protocol less aggressive than TCP. We considered using these frameworks for constructing a background flow algorithm, but we were unable to develop the types of strong non-interference guarantees we seek using these frameworks. One area for future work is developing similar generalizations of Nice in order to allow different background flows to be more or less aggressive compared to one another while all remain completely timid with respect to competing foreground flows.

Prioritizing packet flows would be easier with router support. As noted in Section 4, router prioritization queues such as those proposed for DiffServe service differentiation architectures are capable of completely isolating foreground flows from background flows while allowig background flows to consume nearly the entire available spare bandwidth. Unfortunately, these solutions are of limited use for someone trying to deploy a background replication service today because few applications are deployed soley in environments where router prioritization is installed or activated. A key conclusion of this study is that an end-to-end strategy need not rely on router support to make use of available network bandwidth without interfering with foreground flows.

Router support can also be used to relay network congestion information to end-points. Examples of this approach are random early detection (RED) [25], explicit congestion notification (ECN) [3] and Packeteer's rate controlling scheme based on acknowledgement streaming [2]. These systems raise two issues in the context of Nice. First, by supplying better congestion information, routers may improve the performance of protocols like Nice. Second, our theoretical analysis of Nice assumes drop-tail queues. Future work is needed to quantify how other dropping approaches affect Nice.

Applications limit the network interference they cause in various ways. For example

- Coarse-grain scheduling (e.g., diurnal patterns): Background transfers can be scheduled during hours where there is little foreground traffic. For example, network backup is commonly scheduled for early mornings. Dykes et. al [22] observe appreciable savings in latency by performing updates of prefetched data during the night time alone. Maltzahn et. al [36] discusses bandwidth smoothing techniques by selectively distributing traffic across the course of a day.
- Rate limiting: Senders can pace the rate at which bytes are sent with simple logic, and receivers can limit senders by limiting their maximum advertised TCP receive window. For example, Crovella et. al [19] propose window based rate controlling approaches for prefetching data and show that not only can overall latency be improved using this approach, but the traffic shaping also leads to less bursty traffic and smaller queue lengths. The rate controlling approach spreads prefetched data in the time interval between the end of the previous request and the beginning of the next. For example, the Tivoli Data Exchange [4] system limits per-destination and total bandwidth consumption by each distribution server.
- Application tuning: Applications can limit the amount of data they send by varying application-level parameters. For example, many prefetching algorithms estimate the probability that an object will be referenced and only prfetch that object if its probability exceeds some threshold [21, 28, 48, 39].

We believe self-tuning support for background replication has at least three advantages over existing application-level approaches. Nice operates over fine time scales, so it can provide lower interference (by reacing to spikes in load) as well as higher average throughput (by using a large fraction of spare bandwidth) than static hand-tuned parameters. This property reduces the risk and increases the benefits available to background replication while simplifying design. Also, it appears that Nice provides useful bandwidth throughout the day in many environments. Finally, it is not clear how an engineer should go about setting these application parameters.

The congestion manager CM [8] provides an interface between the transport and the application layers to share information across connections and for handling applications using different transport protocols. Such a system could be used to avoid self-interference, but it is not clear how to use CM information to reduce cross interference.

Microsoft XP's Background Intelligent Transfer Service (BITS) provides three levels of lowered priority levels to minimize interference with the user's interactive sessions. It uses a rate throttling approach based on the amount of spare network bandwidth available. Further technical specifications of how BITS works is not publically available.

8 Conclusions

This paper presents an end-to-end congestion control algorithm optimized to support background transfers. Surprisingly, an end-to-end protocol can nearly approximate the ideal router-prioritization strategy by (a) almost eliminating interference with demand flows and (b) reaping significant fractions of available spare network bandwidth.

This algorithm is designed to support massive replication of data and services, where hardware (e.g., bandwidth, disk space, and processor cycles) is consumed to help humans be more productive. Massive replication systems should be designed as if bandwidth were essentially free. TCP Nice provides a reasonable approximation of such an abstraction.

References

[1] The network simulator – ns2. `http://www.isi.edu/nsnam/ns/`.

[2] Packeteer. http://www.packeteer.com.

[3] Explicit congestion notification. ftp://ftp.isi.edu/in-notes/rfc3168.txt, 2002.

[4] Tivoli data exchange data sheet. http://www.tivoli.com/products/documents/datasheets/data_exchange_ds.pdf, 2002.

[5] Anurag Acharya and Joel Saltz. A study of internet round-trip delay. Technical Report CS-TR-3736, 1996.

[6] Amit Aggarwal, Stefan Savage, and Thomas Anderson. Understanding the performance of TCP pacing. In *Infocom (3)*, 2000.

[7] Akamai, inc. home page. www.akamai.com.

[8] D. Andersen, D. Bansal, D. Curtis, S. Seshan, and H. Balakrishnan. System support for bandwidth management and content adaptation in internet applications, 2000.

[9] D. Bansal and H. Balakrishnan. Binomial Congestion Control Algorithms. In *INFOCOM 2001*, pages 631–640, 2001.

[10] T. Bonald. Comparision of tcp reno and tcp vegas via fluid approximation. In *INRIA, Research Report*, Nov 1998.

[11] L. Brakmo, S. O'Malley, and L. Peterson. Tcp vegas: New techniques for congestion detection and avoidance. In *Proceedings of the ACM SIGCOMM '94 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 24–35, August 1994.

[12] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *Proceedings of IEEE Infocom*, 1999.

[13] B. Chandra. Web workloads influencing disconnected service access. Master's thesis, University of Texas at Austin, May 2001.

[14] B. Chandra, M. Dahlin, L. Gao, A. Khoja, A. Razzaq, and A. Sewani. Resource management for scalable disconnected access to web services. In *WWW10*, May 2001.

[15] B. Chandra, M. Dahlin, L. Gao, and A. Nayate. End-to-end WAN Service Availability. In *Proceedings of the Third USENIX Symposium on Internet Technologies and Systems*, 2001.

[16] Chiu and Jain. "analysis of increase and decrease algorithms for congestion avoidance in computer networks". *Journal of Computer networks and ISDN*, 17(1):1–14, June 1989.

[17] J. Cho and H. Garcia-Molina. Synchronizing a database to improve freshness. In *Proceedings of 2000 ACM International Conference on Management of Data*, May 2000.

[18] J. Cleary and I. Witten. "data compression using adaptive coding and partial string matching". *IEEE Trans. Commun.*, 1984.

[19] M. Crovella and P. Barford. The network effects of prefetching. In *Proceedings of IEEE Infocom*, 1998.

[20] M. Dahlin. "technology trends data". http://www.cs.utexas.edu/users/dahlin/techTrends/data/diskPrices/data, January 2002.

[21] D. Duchamp. Prefetching Hyperlinks. In *Proceedings of the Second USENIX Symposium on Internet Technologies and Systems*, October 1999.

[22] S. Dykes and K. A. Robbins. "a viability analysis of cooperative proxy caching". In *INFOCOM 2001*, 2001.

[23] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http/1.1. Technical Report RFC-2616, IETF, June 1999.

[24] S. Floyd, M. Handley, J. Padhye, and J. Widmer. "equation-based congestion control for unicast applications: the extended version". Technical Report TR-00-003, ICSI, March 2000.

[25] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.

[26] P. Goyal, X. Guo, and H.M. Vin. "a hierarchical cpu scheduler for multimedia operating systems". In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages 107–122, October 1996.

[27] J. Gray and P. Shenoy. Rules of Thumb in Data Engineering. In *"Proc. 16th Internat. Conference on Data Engineering"*, pages 3–12, 2000.

[28] J. Griffioen and R. Appleton. Automatic Prefetching in a WAN. In *IEEE Workshop on Advances in Parallel and Distributed Systems*, October 1993.

[29] J. S. Gwertzman and M. Seltzer. The case for geographical push-caching. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 51–57, May 1995.

[30] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[31] V. Jacobson. "congestion avoidance and control". In *Proceedings of the ACM SIGCOMM '88 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, 1988.

[32] R. Jain. A Delay-Based Approach for Congestion Avoidance in Interconnected Heterogeneous Computer Networks. *ACM Computer Communication Review*, 19(5), October 1989.

[33] Madhukar Korupolu and Mike Dahlin. Coordinated placement and replacement for large-scale distributed caches. In *Workshop On Internet Applications*, June 1999.

[34] T. M. Kroeger, D. E. Long, and J. C. Mogul. Exploring the bounds of web latency reduction from caching and prefetching. In *USENIX Symposium on Internet Technologies and Systems*, 1997.

[35] C. Lumb, J. Schindler, G. Ganger, D. Nagle, and E. Riedel. "towards higher disk head utilization: Extracting free bandwidth from busy disk drives". In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, October 2000.

[36] C. Maltzahn, K. Richardson, D. Grunwald, and J. Martin. On bandwidth smoothing, 1999.

[37] R. Morris. "tcp behavior with many flows". In *International Conference on Network Protocols*, 1997.

[38] A. Odlyzko. "internet growth: Myth and reality, use and abuse". *Journal of COmputer Resource Management*, pages 23–27, 2001.

[39] V. N. Padmanabhan and J. C. Mogul. Using predictive prefetching to improve World-Wide Web latency. In *Proceedings of the SIGCOMM'96*, 1996.

[40] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. In *Proceedings of the Fifteenth ACMSymposium on Operating Systems Principles*, 1995.

[41] V. Paxson. End-to-end Routing Behavior in the Internet. In *Proceedings of the ACM SIGCOMM '96 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, August 1996.

[42] G. Popek, R. Guy, T. Page, and J. Heidemann. Replication in the Ficus Distributed File System. In *Proceedings of the Workshop on the Management of Replicated Data*, pages 5–10, November 1990.

[43] R. Rejaie, M. Handley, and D. Estrin. "rap: An end-to-end rate-based congestion control mechanism for realtime streams in the internet". In *Proceedings of IEEE Infocom*, 1999.

[44] Dheeraj Sanghi, Ashok K. Agrawala, Olafur Gudmundsson, and Bijendra N. Jain. Experimental assessment of end-to-end behavior on internet. In *INFOCOM (2)*, pages 867–874, 1993.

[45] P. Shenoy and H. Vin. "cello: A disk scheduling framework for next-generation operating systems". In *Proceedings of the SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, June 1998.

[46] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the Fifteenth ACMSymposium on Operating Systems Principles*, pages 172–183, December 1995.

[47] A. Venkataramani, M. Dahlin, and P. Weidmann. Bandwidth constrained placement in a WAN. In *Proceedings of the 20th International Conference on Distributed Computing Systems*, Aug 2001.

[48] A. Venkataramani, P. Yalagandula, R. Kokku, S. Sharif, and M. Dahlin. "potential costs and benefits of long-term prefetching for content-distribution". In *Proceedings of the 2001 Web Caching and Content Distribution Workshop*, June 2001.

[49] Z. Wang and J. Crowcroft. A New Congestion Control Scheme: Slow Start and Search (Tri-S). *ACM Computer Communication Review*, 21:32–43, January 1991.

[50] Z. Wang and J. Crowcroft. "a dual-window model for flow and congestion control". *The Distributed Computing Engineering Journal*, 1(3):162–172, May 1994.

[51] D. Wessels. Squid internet object cache. http://squid.nlanr.net/Squid, Jan 1998.

[52] Y. Yang and S. Lam. General AIMD Congestion Control. In *ICNP*, Nov 2000.

[53] H. Yu and A. Vahdat. "the costs and limits of availability for replicated services". In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, October 2001.

[54] Y. Zhang, V. Paxson, and S. Shenkar. The Stationarity of Internet Path Properties: Routing, Loss, and Throughput. Technical report, AT&T Center for Internet Research at ICSI, http://www.aciri.org/, May 2000.

9 Appendix A

In this section we give proofs of the claims in Section 3 and derivation of the upper bound for intereference stated in equation (5). We then show that these derivations also hold true for the case when the number of Reno flows $m < l$.

9.1 Proofs

Lemma 1: The values of W_r and W_n at the beginning of periods stabilize after several packet losses, so that each period thereafter is of a fixed duration T.

Proof: Let t_0 mark the beginning of a period just after a packet loss so that $W(t_0) = (\mu\tau + B)\gamma < \mu\tau$. Let t_1, t_2, t_3 mark respectively the end of the three intervals constituing the current period. The beginning of the interval $[t_2, t_3]$ initiates multiplicative decrease on part of the Nice flows. The dynamics are given by equation (15). Note that for $t < t_2$, $W_n(t) \geq l$ as each Nice flow maintains a window of at least one packet.
If $W_n(t_2) > 2m$, W_n multiplicatively decreases at a rate just enough to counter the rate of increase of W_r, thereby keeping W fixed at $\mu\tau + q_t$ till time $t_2^{'}$ such that $W_n(t_2^{'}) = 2m$. For the rest of the period $[t_2^{'}, t_3]$ the system dynamics are given by:

$$\begin{cases} \frac{dW_r(t)}{dt} = \frac{m\mu}{W(t)} \\ \frac{dW_n(t)}{dt} = -\frac{W_n(t)\mu}{2 \cdot W(t)} \end{cases} \tag{6}$$

These equations yield a unique solution to $W_r(t_3)$ and $W_n(t_3)$, and therefore $W_r(t_3{}^+)$ and $W_n(t_3{}^+)$. The dynamics in every period thereafter exactly emulates the current one.
If $W_n(t_2) < 2m$, it is straightforward to argue that W_n at the beginning of the third interval in every period thereafter keeps monotonically increasing or decreasing. However, as stated above above, this value is bounded below by l and above by $2m$. Thus, the length of the period T and the system dynamics of a period converge.

Corollary 1: After the first packet loss, the residual number of outstanding Nice packets δ just before a packet loss is bounded above by $W_n(t_3)$, obtained by solving (6) with the following boundary conditions:
Initial: $W_n(t_2^{'}) = 2m, \;\; W(t_2^{'}) = \mu\tau + q_t$
Final: $W(t_3) = \mu\tau + B$

Lemma 2: The total amount of flow sent by the Reno flows in a period depends only on the initial and final values of W_r in the period.

Proof: The total amount of flow sent out by the Reno flows is given by integrating the instantaneous sending rate over the duration of the period. The instantaneous sending rate at time t is obtained by dividing the current window size $W_r(t)$ by the current value of the round-trip delay $W(t)/\mu$. Thus, the total amount of Reno flow R sent out in a period beginning at t_0 is given by:

$$R = \frac{1}{m}\int_{t_0}^{t_0+T} \frac{W_r(t)\mu}{W(t)} dt \tag{7}$$

From (13),(14) and (15) we observe that the rate of increase of W_r throughout a period is given by:

$$\frac{dW_r(t)}{dt} = \frac{m\mu}{W(t)} \tag{8}$$

From (7) and (8) we get:

$$\begin{aligned} R &= \frac{1}{m}\int_{W_r(t_0)}^{W_r(t_0+T)} W_r(t)dW_r(t) \\ &= \frac{W_r^2(t)}{2m} \Big|_{t_0}^{t_0+T} \end{aligned} \tag{9}$$

Hence proved.

Lemma 3: The length of a period T in a system with m Reno flows and a non-zero number of Nice flows is shorter than that of a system consisting of only the Reno flows.

Proof: The proof of this lemma follows straighforwardly from the dynamics of $W_r(t)$ alone. Let $T^{'}$ denote the length of a period when only m Reno flows (and no Nice flows) are present, and $W_r^{'}(t)$ denote the total number of outstanding (Reno) packets at time t. Rewriting (8) for $W_r^{'}(t)$:

$$W_r^{'}(t)dW_r^{'}(t) \quad = \quad m\mu dt \tag{10}$$

Assume that a period begins at time t_0. Integrating both sides of (10) and swapping sides we get:

$$\begin{aligned} T^{'} &= \frac{1}{m\mu}\int_{t_0}^{t_0+T^{'}} W_r^{'}(t)dW_r^{'}(t) \\ &= \int_{(\mu\tau+B)\gamma}^{(\mu\tau+B)} W_r^{'}dW_r^{'} \end{aligned} \tag{11}$$

Similarly, using (8) we obtain for T:

$$\begin{aligned} T &= \int_{t_0}^{t_0+T} W(t)dW_r(t) \\ &= \int_{t_0}^{t_0+T} W(t)dW(t).\frac{dW_r(t)}{dW(t)} \\ &= \int_{(\mu\tau+B)\gamma}^{(\mu\tau+B)} WdW.\frac{dW_r}{dW} \end{aligned} \tag{12}$$

Notice that the multiplicative term $\frac{dW_r}{dW} < 1$ throughout the period, as $W = W_r + W_n$. Therefore $T < T^{'}$.

9.2 Bounding Interference

Interference is calculated as the fractional loss in throughput obtained by Reno flows due to the presence of Nice flows. In order to compute this throughput, we first need the residual number of outstanding Nice packets δ just before the end of a period. We use Corollary 1 and solve equatiion (6). Dividing the second differential equation by the first we obtain:

$$\begin{aligned} \frac{dW_n(t)}{dW_r(t)} &= -\frac{W_n(t)}{2m} \\ \Rightarrow \frac{dW_n(t)}{W_n(t)} &= -\frac{dW_r(t)}{2m} \end{aligned}$$

Integrating both sides we obtain:

$$\begin{aligned} \int_{2m}^{\delta}\frac{dW_n}{W_n} &\leq -\frac{1}{2m}\int_{\mu\tau+q_t-2m}^{\mu\tau+B} dW_r \\ \Rightarrow \quad \log(\tfrac{\delta}{2m}) &= -\tfrac{1}{2m}[B - q_t + 2m] \\ \Rightarrow \quad \delta &\leq 2m.\, e^{(-(1+\frac{(B-q_t)}{2m}))} \end{aligned}$$

If the multiplicative decrease of Nice flows is by a factor of γ instead of 2 as assumed in (15), we obtain the bound stated in (4).

The damage fraction may now be easily computed using Lemma 2. The asymptotic throughput obtained by Reno flows is the total flow sent out in a period divided by the length of the period T. By Lemma 2, the total flow sent out by the Reno flows is a period depends only on the initial and final values of $W_r(t)$ in a period. Thus, the throughput P obtained by Reno flows is computed using (9) as:

$$P \quad = \quad \frac{1}{T}\cdot\frac{[(\mu\tau+B-\delta)^2-((\mu\tau+B-\delta)\gamma)^2]}{2m}$$

The throughput obtained by the Reno flows in the absence of any Nice flows is given by:

$$Q \quad = \quad \frac{1}{T^{'}}\cdot\frac{(\mu\tau+B)^2(1-\gamma^2)}{2m}$$

The interference I defined as the fractional loss in throughput is given by $\frac{Q-P}{Q}$. By Lemma 3, $T^{'} > T$, which yields:

$$\begin{aligned} I &\leq \frac{[(\mu\tau+B-\delta)^2-(\mu\tau+B)^2]}{(\mu\tau+B)^2(1-\gamma^2)} \\ &\leq \frac{2\delta}{(\mu\tau+B)(1-\gamma^2)} \end{aligned}$$

Hence Theorem 1 follows.

9.3 Case $m < l$

In this case we simply give the differential equations governing the dynamics of the window sizes claim that Lemmas 1 to 3 hold in this case as well. The verification of the same is left as an exercise to the reader.

In interval $[t_0, t_1]$ $W(t)$ increases from $W(t_0)$ to $\mu\tau$, at which point the queue starts building up. Both Reno and Nice flows increase linearly and their dynamics can be represented as:

$$\begin{cases} \frac{dW_r(t)}{dt} = \frac{m}{\tau} \\ \frac{dW_n(t)}{dt} = \frac{l}{\tau} \end{cases} \tag{13}$$

The next interval $[t_1, t_2]$ is marked by additive increase of W_r, but additive decrease of W_n as the "*Diff* $> \beta$" rule triggers the underlying Vegas controls for the Nice flows. However, the rate of decrease of $W_n(t)$ is bounded by the rate of increase of $W_r(t)$. The two therefore exactly balance each other and the total window size $W(t)$ remains constant at $\mu\tau$. Moreover, in the additive decrease phase, each Nice flow maintains a minimum window of 1, which implies that $W_n(t) \geq l$ in this phase. The round-trip time experienced by each packet when the queue is non-empty is given by $W(t)/\mu$. Thus, the window dynamics during interval $[t_1, t_2]$ are as follows:

$$\begin{cases} \frac{dW_r(t)}{dt} = \frac{m\mu}{W(t)} \\ \frac{dW_n(t)}{dt} = -\frac{m\mu}{W(t)}, \text{ if } W_n(t) > l \\ \qquad\quad\; = 0 \text{ otherwise} \end{cases} \tag{14}$$

The end of this interval is the time t_2 when $W(t_2) = \mu\tau + q_t$, where q_t is the threshold queue size that begins multiplicative backoff for Nice flows. However, again the rate of

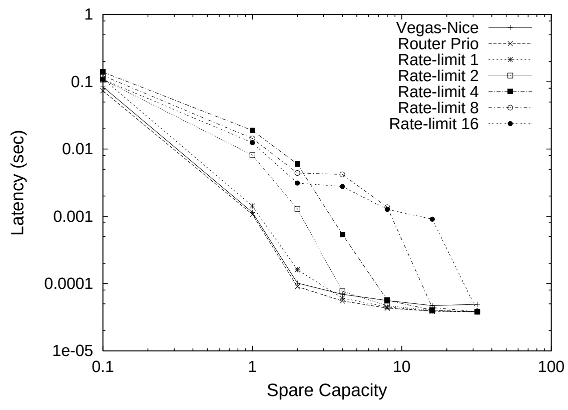


Figure 11: Spare capacity vs Latency

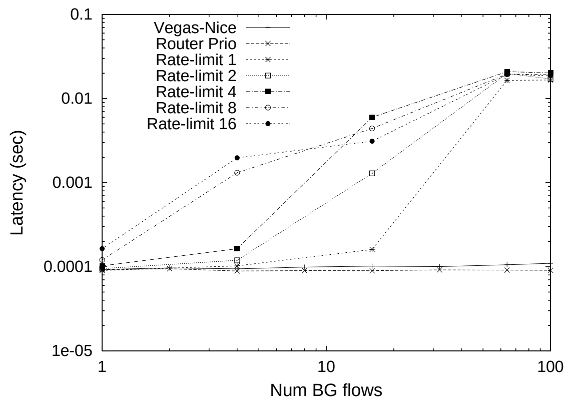


Figure 12: Number of BG flows vs Latency

decrease of $W_n(t)$ is bounded by the rate of increase of increase of $W_r(t)$. Thus, the dynamics of interval $[t_2, t_3]$ are governed by:

$$\begin{cases} \frac{dW_r(t)}{dt} & = & \frac{m\mu}{W(t)} \\ \frac{dW_n(t)}{dt} & = & -min(\frac{W_n(t)\mu}{2 \cdot W(t)}, \frac{m\mu}{W(t)}) \end{cases} \tag{15}$$

The end of the above interval marks the completion of the period. At this point $W(t_3) = \mu\tau + B$, and right after, each flow decreases its window size by a factor of γ, thereby entering into the next period.

Using Lemmas 1 to 3, it is easily shown that the interference bound given in (5) continues to hold.

10 Appendix B

In this section we show the results of the remaining *ns* simulation experiments.

10.1 Rate Limiting

Experiment 4a: In this experiment we compare Nice to simple rate-limited Reno flows. The foreground traffic is again modeled by the Squid trace and the experiment performed is identical to experiment 1.

Fig 11 plots the average latency of foreground packets as a function of the spare capacity in the network. The various lines represent rate-limited background flows with the limits corresponding to a window size of 1,2,4 and 16. *It can be seen that even a flow with a rate limit of 1 inflicts slightly greater interference than Nice!* This result is not surprising as Nice is equipped to reduce its window size below one when it deems necessary to minimize interference. All other flows with higher rates perform much worse and result in upto two orders of magnitude of increase in latency.

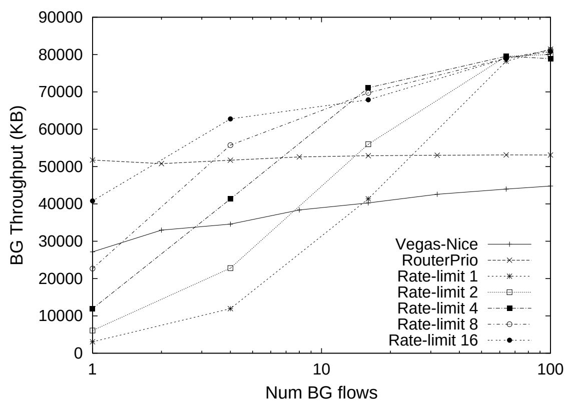


Figure 13: Number of BG flows vs BG throughput

Experiment 4b: In this experiment we fix the capacity of the network to $S = 1$ (L twice the bandwidth needed by demand flows), and we vary the number of background flows. This experiment is identical to experiment 2. Fig 12 plots of the latency of foreground packets against the number of background flows. We observe that even flows limited to a window size of 1 inflict upto two orders of magnitude of increase in latency when there are 64 background flows present. Nice on the other hand is hardly distinguishable from the router prioritization line even for a 100 background flows. Fig 13 plots the number of bytes the background flows manage to get across. We observe that a single Nice background flow gets more throughput than a flow rate limited to a window size of 8. This single Nice flow obtains about 10 times as much throughput as a flow rate-limited to a window of one but still causes lower intereference as was seen in the previous graph. With increasing number of flows, the rate-limited flows show a linear (X-axis is on a log-scale) increase in throughput while the throughput obtained by Nice increases much slower. However, all the rate-limited flows, sooner or later cross the router prioritization line, which means that they steal bandwidth from the foreground flows. Nice on the other hand remains below the router prioritization line always and gets between 60-80% of the spare bandwidth.

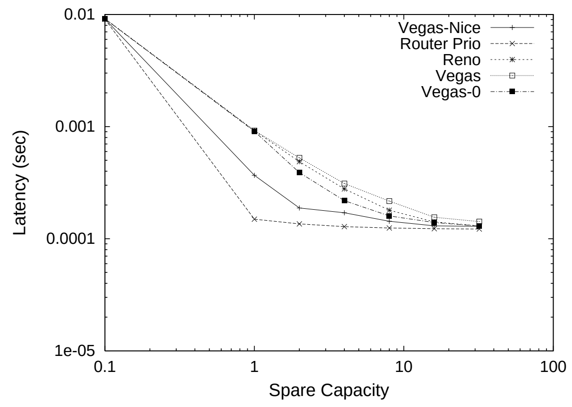


Figure 14: Spare capacity vs Latency

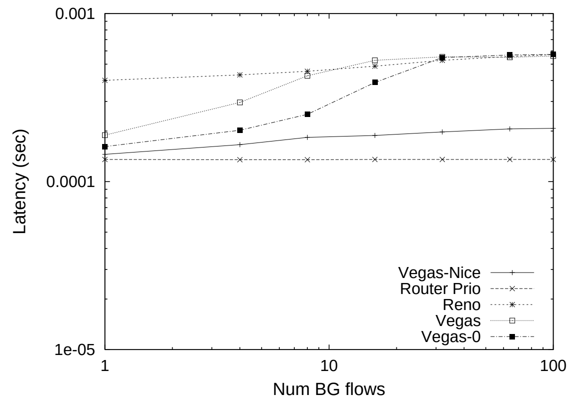


Figure 15: Number of BG flows vs Latency

10.2 On/Off Pareto FG Traffic

Experiment 5a: In this experiment we model the foreground traffic as a set of UDP sources transmitting in an on/off manner in accordance with a Pareto distribution. The burst time and idle time were each set to 250ms, and the value of the *shape* parameter set to 1.5 . The experiments performed were identical to the ones involving trace-based traffic *i.e.* spare capacity and the number of background flows were varied.

Fig 14 plots the average latency of foreground packets as a function of the spare capacity in the network. We observe that though the Nice flows cause less latency overhead than Reno or Vegas, the numbers are not as impressive as in the case when the foreground traffic was a trace following TCP. These numbers suggest that Nice is not well-suited to environments where the traffic is unpredictable. They also support our thesis that Nice works by using round-trip delay estimates in the current round to predict the state of the network in the next. As expected it doesn't work well when the traffic is unpredictable.

Experiment 5b: Sensitivity to number of BG flows In this experiment we fix the capacity of the network to $S = 2$ (L has four time the bandwidth needed by demand flows),

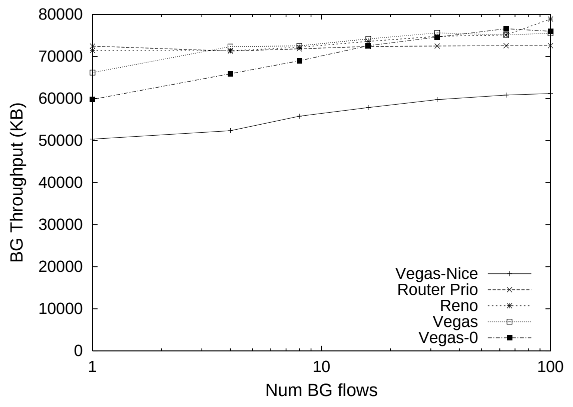


Figure 16: Number of BG flows vs BG throughput

and we vary the number of background flows. Fig 15 plots of the latency of foreground packets against the number of background flows. Again we observe that though Nice outperforms Reno and Vegas, it doesn't match router prioritization as closely. However, Nice continues to show graceful degradation with the number of background flows because of it's ability to decrease it's window size below one. Fig 16 plots the number of bytes the background flows manage to get across. We observe that a single Nice flow obtains about 70% of the spare bandwidth available under router prioritization; this background throughput improves with increasing number of background flows but remains below router prioritization. Thus, Nice reaps a significant fraction of the spare capacity even when the foreground traffic is unpredicatble. This result is not surprising as the interference it causes is also considerable.