TCP Nice: A Mechanism for Background Transfers

Arun Venkataramani Ravi Kokku Mike Dahlin *

Laboratory of Advanced Systems Research
Department of Computer Sciences
University of Texas at Austin, Austin, TX 78712
{arun, rkoku, dahlin}@cs.utexas.edu

Abstract

Many distributed applications can make use of large *background transfers* – transfers of data that humans are not waiting for – to improve availability, reliability, latency or consistency. However, given the rapid fluctuations of available network bandwidth and changing resource costs due to technology trends, hand tuning the aggressiveness of background transfers 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 transfers. Our goal is for the operating system to manage network resources in order to provide a simple abstraction of near zero-cost background transfers. Our system, TCP Nice, can provably bound the interference inflicted by background flows on foreground flows in a restricted network model. And our microbenchmarks and case study applications suggest that in practice it interferes little with foreground flows, reaps a large fraction of spare network bandwidth, and simplifies application construction and deployment. For example, 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

Many distributed applications can make use of large *background transfers* – transfers of data that humans are not waiting for – to improve service quality. For example, a broad range of applications and services such as data backup [29], prefetching [50], enterprise data distribution [20], Internet content distribution [2], and peer-to-peer storage [16, 43] can trade increased network bandwidth consumption and possibly disk space for improved service latency [15, 18, 26, 32, 38, 50], improved availability [11, 53], increased scalability [2], stronger consistency [53], or support for mobility [28, 41, 47]. 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 [8, 10] or by updating cached copies more frequently as data change [13, 50, 48]; 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 [9, 17, 37]; conversely network availability [11, 40, 54] 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 transfers. In particular, because background transfers compete with foreground requests, they can hurt overall performance and availability by increasing network congestion. Applications must therefore carefully balance the benefits of background transfers 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 [18, 26]) 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 transfers. A self-tuning background transport layer will enable new classes of applications by (1) simplifying applications, (2) reducing the risk of being too aggressive, and (3) making

*This work was supported in part by an NSF CISE grant (CDA-9624082), the Texas Advanced Technology Program, the Texas Advanced Research Program, and Tivoli. Dahlin was also supported by an NSF CAREER award (CCR-9733842) and an Alfred P. Sloan Research Fellowship.

it easier to reap a large fraction of spare bandwidth to gain the advantages of background transfers. 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 [9, 37]). 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 [10, 24, 33, 39, 45] and because networks are shared across applications, users, and organizations and therefore pose the most critical resource management challenge to aggressive background transfers.

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 [30] or TCP Vegas [7] by detecting congestion earlier, reacting to it more aggressively, and 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. Our analysis shows that all three features described above are essential for bounding interference.

Our microbenchmarks comprise both *ns* [36] 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, in one microbenchmark, 16 Nice background flows slow down the average demand document transfer time by less than 10% and reap over 70% of the spare network bandwidth. But in the same situation, 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 a simplified version of the Tivoli Data Exchange [20] 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 performance 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 [22] or rate-based [42]. Our protocol is strictly less aggressive than Reno, and we expect that it causes 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 [18, 50]. 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 respectively. 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 believed to be fundamental to the robustness of the Internet [12, 30].

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 [7], a protocol that differs from TCP-Reno in its congestion avoidance phase. By monitoring round-trip delays, 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 adjusts the window W once every round as follows ($minRTT$ is the minimum value of all measured round-trip delays and $observedRTT$ is the round-trip delay experienced by a distinguished packet in the previous round):

$E \leftarrow \frac{W}{minRTT}$ // Expected throughput

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

$Diff \leftarrow (E - A) \cdot minRTT$

if ($Diff < \alpha$)
  $W \leftarrow W + 1$
else if ($Diff > \beta$)
  $W \leftarrow W - 1$

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: (i) Vegas has been designed to compete for throughput approximately fairly with Reno, (ii) Vegas backs off when the number of queued packets from its flow increases, but it does not necessarily back off when the number of packets enqueued by other flows increase, (iii) 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 Internet 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 delay 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. Round-trip delays of packets are indicative of the current bottleneck queue size and the threshold represents the fraction of the total queue capacity that starts to trigger congestion. The Nice congestion avoidance mechanism incorporating the *interference trigger* with threshold t and fraction f can be written as follows ($curRTT$ is the round-trip delay experienced by each packet):

per ack operation:
  if $(curRTT > (1-t) \cdot minRTT + t \cdot maxRTT)$
    $numCong$++;
per round operation:
  if $(numCong > f \cdot W)$
    $W \leftarrow W/2$
  else $\{$
    $\ldots$ // 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.

Maintaining a window of less than one causes us to 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 dissipate. 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 [35].

When a Nice flow signals congestion, it halves its current congestion window. In contrast Vegas reduces its window by one packet each round that encounters 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 utilization of spare capacity, but our design goals lead us to minimize interference even at the potential cost of utilization. Our experimental results show that even with these aggressively timid policies, we achieve reasonable levels of utilization in practice.

As in TCP Vegas, maintaining running measures of $minRTT$ and $maxRTT$ 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 [1, 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. The use of minimum and maximum values makes the prototype sensitive to outliers. Using the first and ninety-ninth percentile values could improve the robustness of this algorithm, but we have not tested this optimization. Route changes during a transfer can also contribute to inaccuracies in RTT estimates. However such changes are uncommon [40] and we speculate that they can be handled by maintaining exponentially decaying averages for $minRTT$ and $maxRTT$ estimates.

2.3 Prototype Implementation

We implement a prototype Nice system 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. In our implementation of Nice, we set the corresponding Vegas parameters α and β to 1 and 3 respectively. After the first round-trip delay estimate, maxRTT is initialized to $2 \cdot minRTT$.

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 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 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 analyze it formally to bound the reduction in throughput that Nice imposes on foreground flows. Our primary result is that under a simplified network model, 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.

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. Most significantly, our formal analysis assumes a simplified fluid approximation and synchronous network model, as described below. Also, 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. 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. Although these assumptions are restrictive, the insights gained in the analysis lead us to expect the protocol to work well under more general circumstances. The 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.

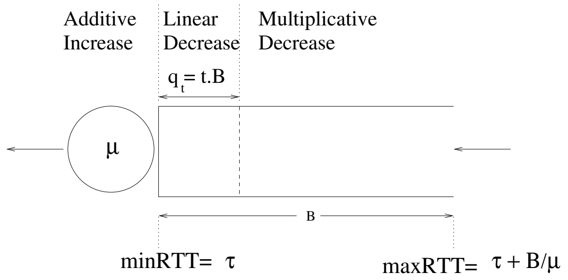


Figure 1: Nice Queue Dynamics

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 buffer capacity 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, each of which has one continuously backlogged flow between a source and a destination. Let t be the Nice threshold and $q_t = t \cdot B$ be the corresponding queue size that triggers multiplicative backoff for Nice flows. The connections are homogeneous, *i.e.* they experience the same propagation 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. Models assuming flow synchronization have been used in previous analyses [6]. We model only the congestion avoidance phase to analyze the steady-state behavior.

We obtain a bound on the reduction in the throughput of Reno flows due to the presence of Nice flows by analyzing the dynamics of the bottleneck queue. We achieve this goal by dividing the duration of the flows into *periods*. In each period we bound the decrease in the number of Reno packets processed by the router due to interfering Nice packets. In the following we give an outline of this analysis. The complete analysis with detailed proofs appears in the our technical report [49].

Let $W_r(t)$ and $W_n(t)$ denote respectively the total number of outstanding Reno and Nice packets in the network at time t. $W(t)$, the total window size, is $W_r(t)+W_n(t)$. We trace these window sizes across 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. Consider the case when $W(t_0) = (\mu\tau + B)\gamma < \mu\tau$ and $m > l$. 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. The window dynamics in any period can be split into three intervals as described below.

Additive Increase, Additive Increase: In this interval $[t_0, t_1]$ both Reno and Nice flows increase linearly. $W(t)$ increases from $W(t_0)$ to $W(t_1) = \mu\tau$, at which point the queue starts building.

Additive Increase, Additive Decrease: This 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 end of this interval is marked by $W(t_2) = \mu\tau + q_t$.

Additive Increase, Multiplicative Decrease: In this interval $[t_2, t_3]$, $W_n(t)$ multiplicatively decreases in response to observing queue lengths above q_t. However, the rate of decrease of $W_n(t)$ is bounded by the rate of increase of $W_r(t)$, as any faster a decrease will cause the queue size to drop below q_t. At the end of this interval $W(t_3) = \mu\tau + B$. At this point, 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 formulate differential equations to represent the variation of the queue size in a period. We then show that the values of W_r and W_n at the beginning of periods stabilize after several losses, so that the length of a period converges to a fixed value. It is then straightforward to compute the total amount of Reno flow sent out in a period. We show in the technical report [49] that the interference I, defined as the fractional loss in throughput experienced by Reno flows because of the presence of Nice flows, is given as follows.

Theorem 1: The interference I is given by

$$I \leq \frac{4m \cdot e^{(-\frac{B(1-t)\gamma}{m})}}{(\mu\tau + B)\gamma} \qquad (1)$$

The derivation of I indicates that all three design features of Nice are fundamentally important for reducing interference. The interference falls exponentially with $B(1-t)$ or $B - q_t$, which reflects the time that Nice has to multiplicatively back off before packet losses occur. Intuitively, multiplicative decrease allows any number of Nice flows to get out of the way of additively increasing demand flows. The dependence on the ratio $\frac{B}{m}$ 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 anomalously under such circumstances [35]. In a well designed network, when $B \gg m$, it can be seen that the dependence on the threshold t is weak, *i.e.* interference is small when t is, and careful tuning of the exact value of t in this region is unnecessary. Our full analysis shows 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.

4 ns Controlled Tests

The goal of our simulation 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* [36] 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 performs better than other existing protocols, including Reno, Vegas, and Vegas with reduced α and β parameters.

4.1 Methodology

We use *ns* 2.1b8a for our simulation experiments. The topology used is a bar-bell 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 except in experiments with RED turned on. The buffer size is set to the bandwidth delay product. Packets are 512 bytes in size and the propagation delay is set to 50ms. We vary the capacity of the link in order to simulate different amounts of spare capacity.

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 fluctuates as clients enter and leave the system as specified by the trace. On average there are

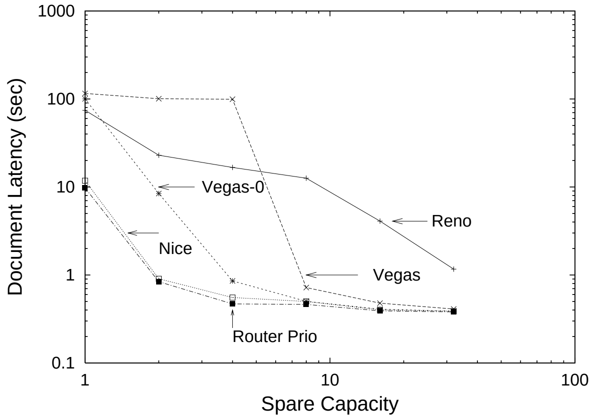


Figure 2: Spare capacity vs Latency

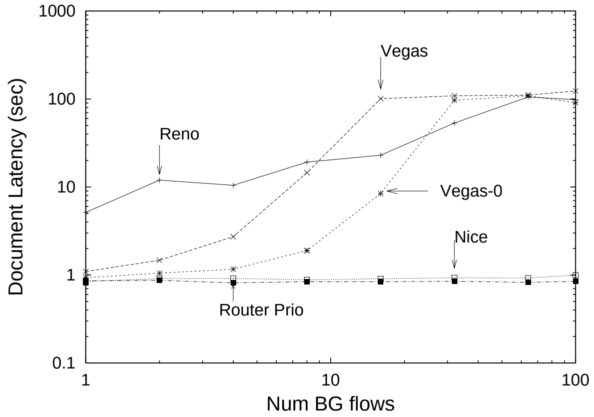


Figure 3: Number of BG flows vs Latency

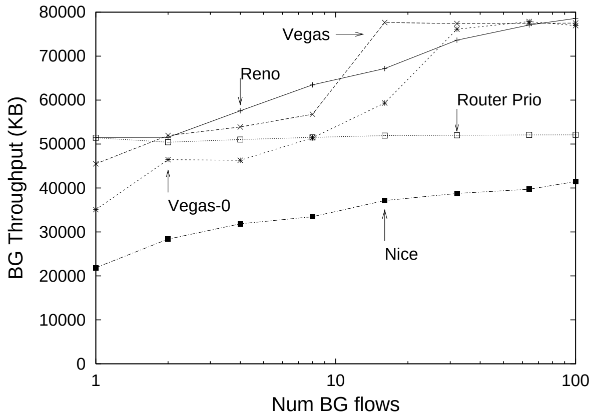


Figure 4: Number of BG flows vs BG throughput

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 transfer latency of a document *i.e.*, the time between its first packet being sent and the receipt of the ack corresponding to the last packet. We use the total number of bytes transferred by the background flows as the measure of its utilization of spare capacity.

Unless otherwise specified, the values of the $threshold$ and $fraction$ for Nice are set to 0.1 and 0.5 respectively. We compare the performance of Nice to several other strategies for sending background flows. First, we compare with 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, as background flows never interfere with foreground flows. In addition, we compare to Reno, Vegas($\alpha = 1, \beta = 3$), Vegas($\alpha = 0, \beta = 0$).

4.2 Results

Experiment 1: In this experiment we fix the number of background flows to 16 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 document transfer latency for foreground traffic 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 Nice is hardly distinguishable from 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 document transfer latencies by over an order of magnitude.

Experiment 2: Sensitivity to number of BG flows In this experiment we fix the spare capacity S of the network to 1 and vary the number of background flows. Figure 3 plots the latency of foreground document transfers against the number of background flows. Even with 100 background Nice flows, the latency of foreground documents is hardly distinguishable from the ideal case when routers provide strict prioritization. On the other hand, Reno and Vegas background flows can cause foreground latencies to increase by orders of magnitude. Figure 4 plots the number of bytes the background flows manage to transfer. 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 end-to-end algorithm. Note that although Reno and Vegas obtain better throughput, even for a small number of flows they go beyond the router prioritization line, which means they steal bandwidth from foreground traffic.

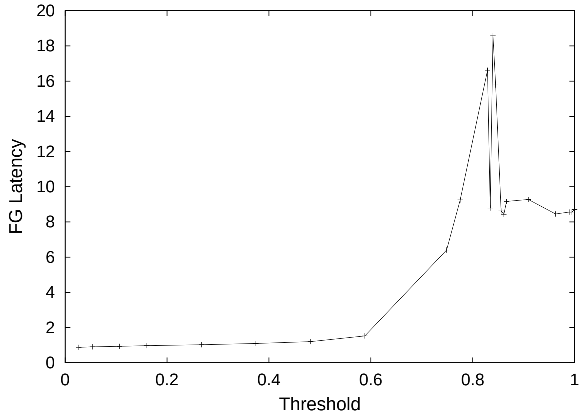


Figure 5: Threshold vs FG latency

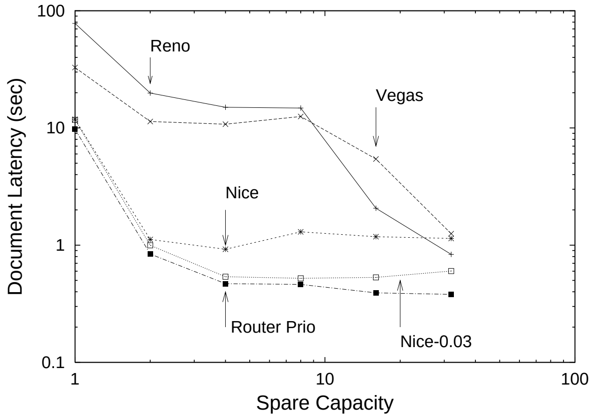


Figure 6: Spare capacity vs Latency with RED

We also examine experiments where we do not allow Nice's 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 noticeably; the increase is about a factor of two when the number of background flows is 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 the document transfer latencies as a function of the threshold for the same trace as above, with $S = 1$ and 16 background flows. 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 little change in foreground latency as we vary this fraction from 0.1 to 0.9 (graph omitted).

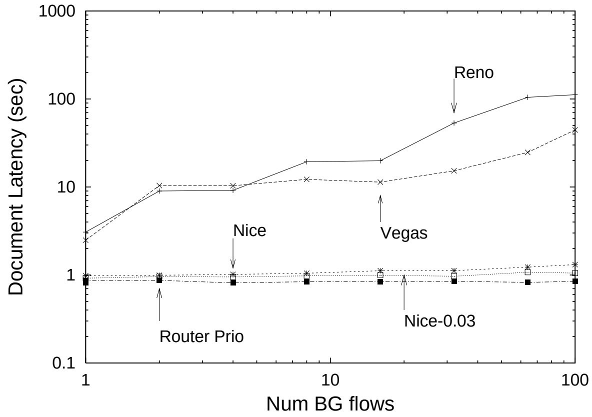


Figure 7: Number of BG flows vs Latency with RED

Experiment 4: Nice with RED queueing We repeat experiments 1 and 2, but with routers performing RED queueing. The RED parameters are set as recommended in [23] with the "gentle" mode on. The minimum and maximum RED thresholds are set to one-third and two-third of the buffer size. Packets are probabilistically marked with ECN support from the senders. Figure 6 plots the foreground document transfer latency against the spare capacity with 16 background flows. Though Nice still performs as much as an order of magnitude better than other protocols, it causes up to a factor of 2 increase in document transfer latencies for large spare capacities. As figure 7 indicates, under RED, Nice closely approximates router prioritization regardless of the number of flows when the spare capacity is one, i.e. the demand workload consumes half of the network capacity.

The relatively poor performance of Nice under RED when spare capacities are large appears to reflect the sensitivity of Nice's interference I to bottleneck queue length (Equation 1). Whereas Nice flows damage foreground flows when drop-tail queues are completely full, under RED, interference can begin when the bottleneck queue occupancy reaches RED's minimum threshold min_{th}. One solution may be to reduce Nice's $threshold$ parameter. The *Nice-0.03* lines in Figures 6 and 7 plot Nice's interference under RED when $threshold = 0.03$ instead of the default value of 0.10. Future work is needed to better understand Nice's interaction with RED queuing.

Other results Due to space constraints, we state two other results here, but omit detailed discussions and graphs. The full discussion appears in the extended version [49].

First, we also perform experiments with synthetically generated ON/OFF Pareto UDP foreground traffic, 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 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

In this section 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 usefulness 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 throughput that are significant fractions of the throughput 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.[1] 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.

[1] 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.

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 Abilene OC3 (*Delaware*). All machines run Linux. The server is a 450MHz Pentium II with 256MB of memory. The clients 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 8 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 receive alone. Conversely, Reno/Reno and Reno/Reno8 show the expected fair sharing of bandwidth among Reno flows, which reduces the bandwidth achieved by the primary flow.

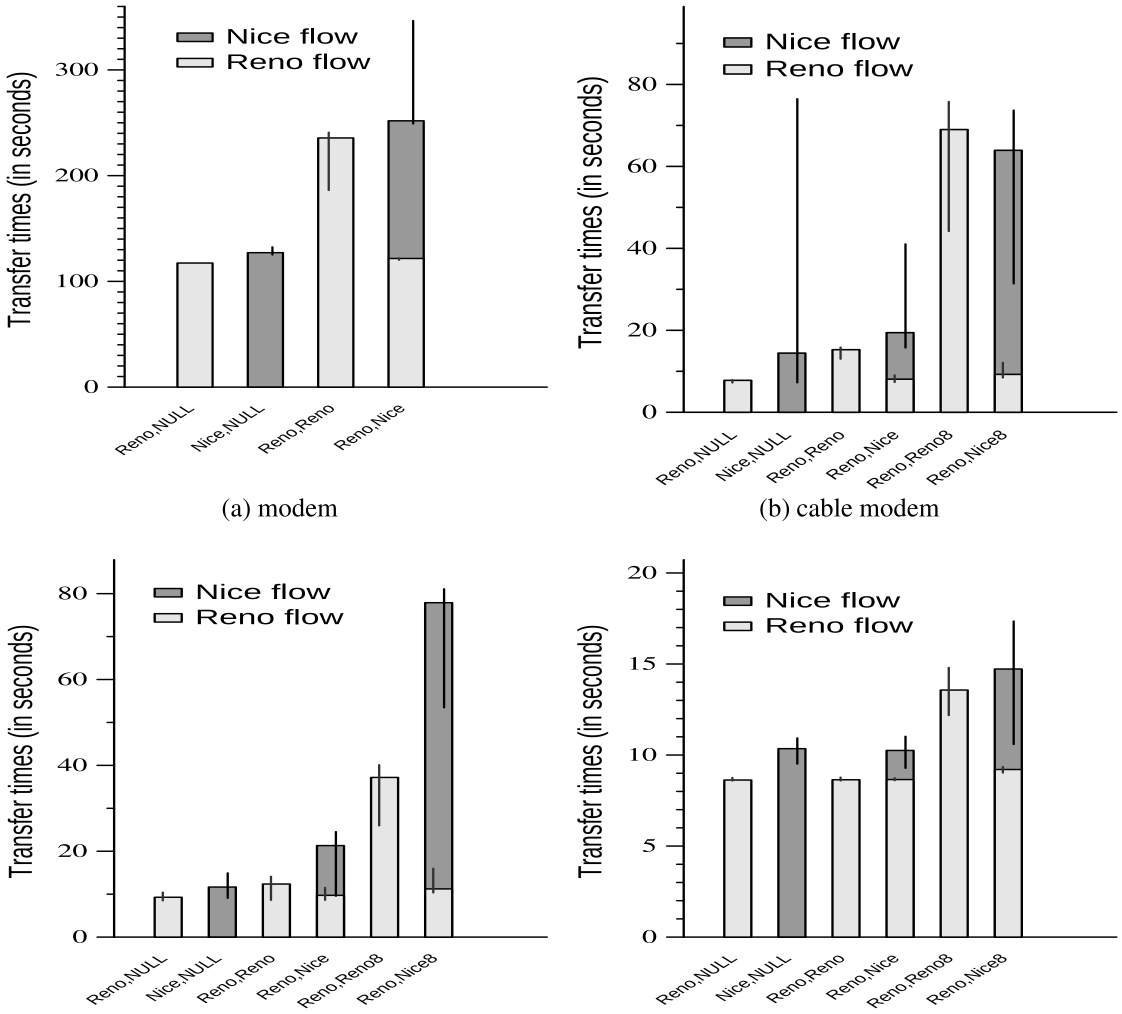
Figure 9 shows the hourly average bandwidth achieved by the primary flow for the different combinations listed above. Our hypothesis is that Nice can achieve useful amounts of throughput throughout the day, and the data appear to support this statement.

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 [15, 18, 26, 27, 32, 38, 50].

Typically, prefetching algorithms are tuned with a *threshold* parameter to balance the potential benefits of prefetching data against the bandwidth costs of fetching



(a) modem

(b) cable modem

(c) London

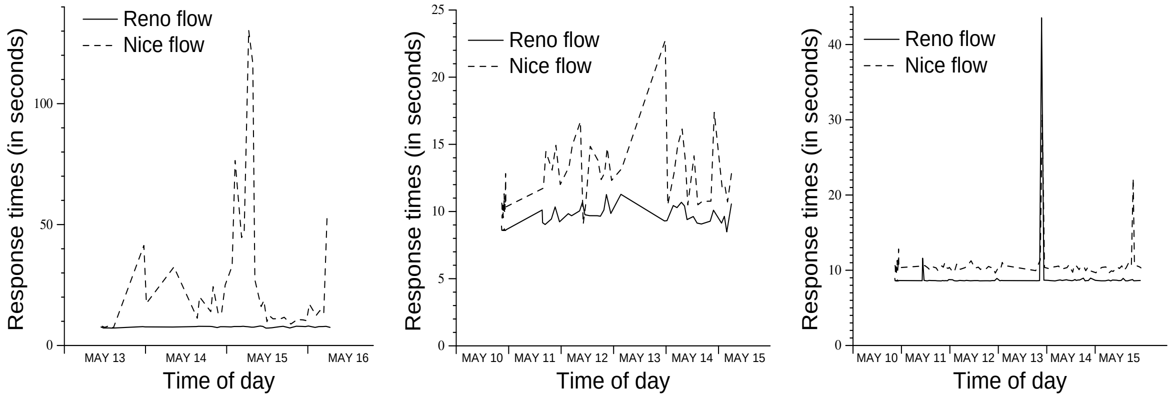
(d) Delaware

Figure 8: 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 depict the minimum and maximum values observed during multiple runs of each combination.

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 [25], 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 [9]. Furthermore, the 80-100% per year improvements in network [9, 37] and disk [17] capacity/cost mean that a value that is correct today may be off by an order of magnitude in 3-4 years.

In this case study, we build a prefetching protocol similar to the one proposed by Padmanabhan and Mogul [38]: 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 [21].



(a) cable modem (b) London (c) Delaware

Figure 9: Large flow transfer performance over time.

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. We discuss how to deploy such a protocol without modifying HTTP in a separate study [31].

We use Squid proxy traces from 9 regional proxies collected during January 2001 [51]. 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.). 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.

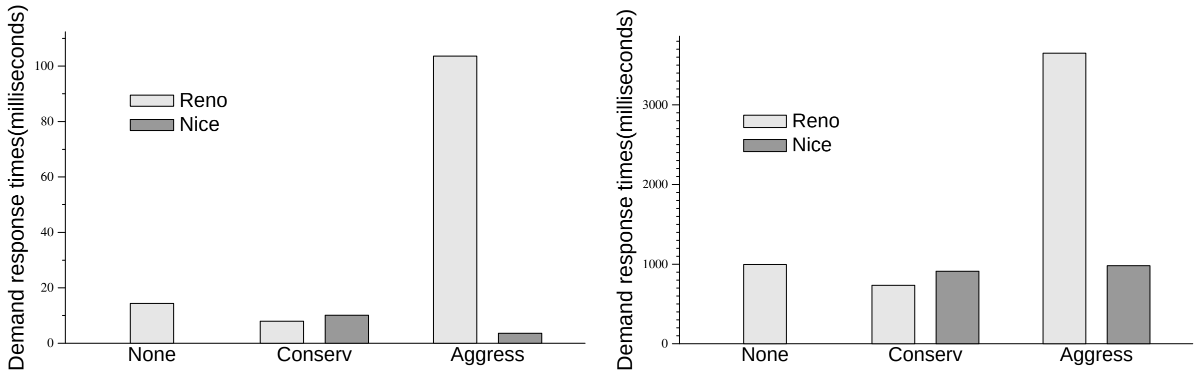
We use a simple prediction by partial matching algorithm [14] PPM-n/w that uses a client's n most recent requests to the server group for non-image data to predict cachable (i.e., non-dynamically-generated) URLs that will appear during a subsequent window that ends after the w'th non-image request to the server group. 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 [18]. 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 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*.

Figure 10(a) shows the average demand response times perceived by the clients. 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 effective in using spare bandwidth for prefetching without affecting the demand requests.

Figure 10(b) represents the effect of prefetching over a modem (the setup is same as above except with the cable modem replaced by a modem), an environment where the amount of spare bandwidth available is minimal. 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.

We conclude that Nice simplifies the design of prefetching applications. Applications can aggressively prefetch data that might be accessed in the future. Nice prevents interference if the network does not have spare bandwidth and improves application performance if it does.



(a) cable modem (b) dial-up modem

Figure 10: Average demand transfer time for prefetching for the cnn server-group.

6.2 Tivoli Data Exchange

We study a simplified version of the Tivoli Data Exchange [20] 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. 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 modem). 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.

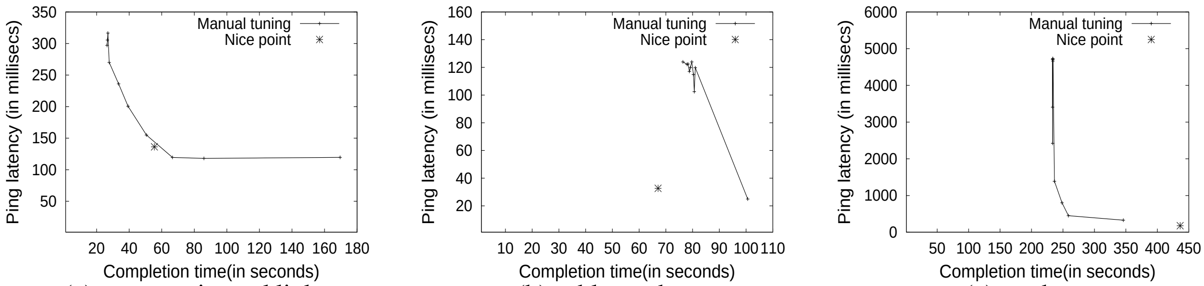
Nice can provide an attractive self-tuning abstraction. Using it, a sender can just send at the maximum speed allowed by the connection. We report preliminary results using a standalone server and client. The server and clients are the same as in the Internet measurements described in Section 5. We initiate 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 11 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. Furthermore, the optimal rates vary widely across different networks. However Nice picks sending rates for each connection without the need for manual tuning that achieve minimum transfer times while maintaining acceptable ping latencies in all cases.

7 Related work

TCP congestion control has seen an enormous body of work since Jacobson's seminal paper on the topic [30]. 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 AIMD TCP: it reacts the same way to losses and in addition, it reacts to increasing delays. Therefore, the work to ensure network stability under AIMD TCP applies to Nice as well.

The GAIMD [52] and binomial [4] 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



(a) transcontinental link (b) cable modem (c) modem

Figure 11: Each continuous line represents completion times and corresponding ping latencies with varying send rates. The single point is the send rate chosen by Nice.

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 DiffServ [5] service differentiation architectures are capable of completely isolating foreground flows from background flows while allowing 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 solely 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.

Applications can limit the network interference they cause in various ways:
(a) *Coarse-grain scheduling*: Background transfers can be scheduled during hours where there is little foreground traffic. Studies [19, 34] show that prefetching data during off-peak hours can reduce latency and peak bandwidth usage.
(b) *Rate limiting*: Spring et. al [46] discuss prioritizing flows by controlling the receive window sizes of clients. Crovella et. al [15] propose a combination of window-based rate control and pacing to spread out prefetched traffic to limit interference. They show that such shaping of traffic leads to less bursty traffic and smaller queue lengths.
(c) *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 prefetch that object if its probability exceeds some threshold [18, 26, 38, 50].

It is not clear how an engineer should go about setting such application-specific parameters. We believe that self-tuning support for background transfers has at least three advantages over existing application-level approaches. Nice operates over fine time scales, so it can provide lower interference (by reacting 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 transfers while simplifying application design. Our experiments also demonstrate that Nice provides useful bandwidth throughout the day in many environments.

Existing transport layer solutions can be used to tackle the problem of self-interference between a single sender/receiver's flows. The congestion manager CM [3] provides an interface between the transport and the application layers to share information across connections and for handling applications using different transport protocols. Microsoft XP's Background Intelligent Transfer Service (BITS) provides support for transfers of lower priority to minimize interference with the user's interactive sessions by using a rate throttling approach. In contrast to these approaches, Nice handles both self- as well as cross-interference by modifying the sender side alone.

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.

Our Internet experiments suggest that there is a significant amount of spare capacity on a wide variety of Internet links. Nice provides a mechanism to improve application performance by harnessing this capacity in a non-interfering manner. Our case studies demonstrate that Nice can simplify application design by eliminating the need to hand-tune parameters to balance utilization and interference. Inspired by the results in this paper, we have built a self-tuning prefetching system [31] based on Nice that avoids interference at the server and in the network, and is deployable with simple modifications to a web server.

One application of Nice is to support massive replication of data and services, where spare resources (e.g., bandwidth, disk space, and processor cycles) are 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] Anurag Acharya and Joel Saltz. A study of internet round-trip delay. Technical Report CS-TR-3736, University of Maryland, 1996.

[2] Akamai, Inc. http://www.akamai.com.

[3] D. Andersen, D. Bansal, D. Curtis, S. Seshan, and H. Balakrishnan. System support for bandwidth management and content adaptation in internet applications. In *OSDI*, pages 213–226, 2000.

[4] D. Bansal and H. Balakrishnan. Binomial Congestion Control Algorithms. In *Infocom*, 2001.

[5] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An architecture for differentiated services, 1998.

[6] T. Bonald. Comparision of TCP Reno and TCP Vegas via fluid approximation. INRIA Research Report 3563, Nov 1998.

[7] Lawrence S. Brakmo and Larry L. Peterson. TCP vegas: End to end congestion avoidance on a global internet. *IEEE Journal on Selected Areas in Communications*, 13(8):1465–1480, 1995.

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

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

[10] 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.

[11] B. Chandra, M. Dahlin, L. Gao, and A. Nayate. End-to-end WAN Service Availability. In *USITS*, 2001.

[12] 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.

[13] J. Cho and H. Garcia-Molina. Synchronizing a database to improve freshness. In *SIGMOD*, 2000.

[14] J. Cleary and I. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, 1984.

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

[16] F. Dabek, F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *SOSP*, 2001.

[17] M. Dahlin. http://www.cs.utexas.edu/users/dahlin/techTrends/data/diskPrices/data, Jan 2002.

[18] D. Duchamp. Prefetching Hyperlinks. In *USITS*, 1999.

[19] S. Dykes and K. A. Robbins. A viability analysis of cooperative proxy caching. In *Infocom*, 2001.

[20] Tivoli Data Exchange. http://www.tivoli.com/products/documents/datasheets/data_exchange_ds.pdf.

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

[22] 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.

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

[24] P. Goyal, X. Guo, and H.M. Vin. A hierarchical cpu scheduler for multimedia operating systems. In *OSDI*, pages 107–122, October 1996.

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

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

[27] J. S. Gwertzman and M. Seltzer. The case for geographical push-caching. In *HotOS*, 1995.

[28] 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.

[29] N. Hutchison, S. Manley, M. Federwisch, G. Harris, D. Hitz, S. Kleiman, and S. O'Malley. Logical vs. physical file system backup. In *OSDI*, 1999.

[30] V. Jacobson. Congestion avoidance and control. In *SIGCOMM*, 1988.

[31] R. Kokku, P. Yalagandula, A. Venkataramani, and M. Dahlin. A non-interfering deployable web prefetching system. Technical Report TR-02-51, Computer Sciences, UT Austin, May 2002.

[32] T. M. Kroeger, D. E. Long, and J. C. Mogul. Exploring the bounds of web latency reduction from caching and prefetching. In *USITS*, 1997.

[33] C. Lumb, J. Schindler, G. Ganger, D. Nagle, and E. Riedel. Towards higher disk head utilization: Extracting free bandwidth from busy disk drives. In *OSDI*, 2000.

[34] C. Maltzahn, K. Richardson, D. Grunwald, and J. Martin. On bandwidth smoothing. In *4th International Web Caching Workshop*, 1999.

[35] R. Morris. Tcp behavior with many flows. In *International Conference on Network Protocols*, 1997.

[36] The network simulator – ns-2. http://www.isi.edu/nsnam/ns.

[37] A. Odlyzko. Internet growth: Myth and reality, use and abuse. *Journal of Computer Resource Management*, pages 23–27, 2001.

[38] V. N. Padmanabhan and J. C. Mogul. Using predictive prefetching to improve World-Wide Web latency. In *SIGCOMM*, 1996.

[39] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. In *SOSP*, 1995.

[40] V. Paxson. End-to-end Routing Behavior in the Internet. In *SIGCOMM*, 1996.

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

[42] R. Rejaie, M. Handley, and D. Estrin. RAP: An end-to-end rate-based congestion control mechanism for realtime streams in the internet. In *Infocom*, 1999.

[43] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *SOSP*, 2001.

[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 *SIGMETRICS*, 1998.

[46] Neil T. Spring, Maureen Chesire, Mark Berryman, Vivek Sahasranaman, Thomas Anderson, and Brian N. Bershad. Receiver based management of low bandwidth access links. In *Infocom*, 2000.

[47] 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 *SOSP*, 1995.

[48] A. Venkataramani, M. Dahlin, and P. Weidmann. Bandwidth constrained placement in a WAN. In *PODC*, 2001.

[49] A. Venkataramani, R. Kokku, and M. Dahlin. System support for background replication. Technical Report TR-02-30, Computer Sciences, UT Austin, May 2002.

[50] A. Venkataramani, P. Yalagandula, R. Kokku, S. Sharif, and M. Dahlin. Potential costs and benefits of long-term prefetching for content-distribution. *Computer Communications Journal*, 25(4):367–375, 2002.

[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*, 2000.

[53] H. Yu and A. Vahdat. The costs and limits of availability for replicated services. In *SOSP*, 2001.

[54] Y. Zhang, V. Paxson, and S. Shenkar. The Stationarity of Internet Path Properties: Routing, Loss, and Throughput. Technical report, ICSI Center for Internet Research, May 2000.