Safe Speculative Replication

MIKE DAHLIN *University of Texas at Austin*
ARUN IYENGAR *IBM TJ Watson Research*
RAVI KOKKU *NEC Labs, Princeton, NJ*
AMOL NAYATE *IBM TJ Watson Research*
ARUN VENKATARAMANI *University of Massachusetts, Amherst*
PRAVEEN YALAGANDULA *HP Labs, Palo Alto, CA*

This article argues that commonly-studied techniques for speculative replication—such as prefetching or pre-pushing content to a location before it is requested there—are inherently unsafe: they pose unacceptable risks of catastrophic overload and they may introduce bugs into systems by weakening consistency guarantees. To address these problems, the article introduces SSR, a new, general architecture for Safe Speculative Replication. SSR specifies the mechanisms that control how data flows through the system and leaves as a policy choice the question of what data to replicate to what nodes. SSR's mechanisms (1) separate invalidations, demand updates, and speculative updates into three logical flows, (2) use per-resource schedulers that prioritize invalidations and demand updates over speculative updates, and (3) use a novel scheduler at the receiver to integrate these three flows in a way that maximizes performance and availability while meeting specified consistency constraints. We demonstrate the SSR architecture via two extensive case studies and show that SSR makes speculative replication practical for widespread adoption by (1) enabling self-tuning speculative replication, (2) cleanly integrating speculative replication with consistency, and (3) providing easy deployability with legacy infrastructure.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications, distributed databases*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*distributed programming*; D.4.4 [**Operating Systems**]: Communications Management—*network communication*; D.4.5 [**Operating Systems**]: Reliability—*fault tolerance*; D.4.7 [**Operating Systems**]: Organization and Design—*distributed systems*; H.3.4 [**Information Storage and Retrieval**]: System and Software—*distributed systems*; H.3.5 [**Information Storage and Retrieval**]: Online Information Services—*data sharing*

General Terms: Algorithms, Design, Performance, Reliability

Additional Key Words and Phrases: Prefetching, Data replication, Consistency

1. INTRODUCTION

[1]*Speculative replication*—prefetching [Chandra et al. 2001; Cho and Garcia-Molina 2000; Duchamp 1999; Griffioen and Appleton 1994a; Kokku et al. 2003; Padmanabhan and Mogul 1996a] or pre-pushing [Gao et al. 2003; Gwertzman and Seltzer 1995; Nayate et al. 2004; Venkataramani et al. 2002] content to a location before it is used there—has immense potential benefits for large-scale distributed systems operating over wide-area networks (WANs). WANs are characterized by two fundamental limitations: (i) high latency that limits the responsiveness of networked services and (ii) susceptibility to network partitions that limits the availability of networked services in spite of high end-system availability [Dahlin et al. 2003]. Speculative replication (SR) can potentially mitigate both of these limitations [Chandra et al. 2001; Cho and Garcia-Molina 2000; Duchamp 1999; Gao et al. 2003; Gwertzman and Seltzer 1995; Nayate et al. 2004; Padmanabhan and Mogul 1996a; Yu and Vahdat 2001]. Furthermore, two technology trends favor widespread use of SR. First, rapidly falling prices of bandwidth, storage, and computing [Gray 2003] argue for "wasting" inexpensive hardware resources to obtain improvements in human wait-time

[1]Some parts of this manuscript have been discussed in more detail in previously published papers [Kokku et al. 2003; Nayate et al. 2004; Venkataramani et al. 2002b].

and productivity [Chandra 2001]. Second, the bursty nature of WAN workloads [LeFebvre 2001; Robert Blumofe 2002] often results in overprovisioning of systems that, during non-peak periods, leaves abundant spare capacity for SR.

Despite its promise on paper, few systems today employ SR. Numerous studies show significant benefits from prefetching on the Web [Padmanabhan and Mogul 1996a; Gwertzman and Seltzer 1995; Duchamp 1999; Fan et al. 1999; Bestavros 1996a; Chen and Zhang ; Markatos and Chronaki 1998; Palpanas 1998] or in file systems [Cao et al. 1995a; Kotz and Ellis 1991; Kimbrel et al. 1996; Patterson et al. 1995b; Smith 1978], production systems seldom deploy prefetching unless significantly overprovisioned. In this article, we argue that two crucial challenges prevent widespread deployment of SR. First is the difficult resource management problem that SR introduces. Second is the complexity of integrating SR into systems with significant consistency constraints.

The resource management problem arises because SR increases the overall load on the system in order to improve performance and availability. SR systems must therefore manage a trade-off between the benefits of SR and the *interference* from SR that can degrade performance of regular requests. Unfortunately, choosing the right trade-off is not simple because (1) the trade-off is *non-monotonic*—increasing resource expenditure for SR does not always translate to increased benefits, (2) the cost/benefit trade-off is *complex* because it must balance a large number of different resources and goals, (3) the appropriate cost/benefit trade-off is *non-stationary*—the appropriate trade-off varies over time as workloads and available resources change, and (4) the costs of SR are *non-linear*—a small increase in SR can in some circumstances cause catastrophic damage to overall system performance. Self-tuning resource management is thus crucial to SR because static, hand-tuned replication policies are more complex to maintain, less able to benefit from spare system resources, and more prone to catastrophic overload if they are mis-tuned or during periods of unexpectedly high system load.

Consistency constraints pose a second challenge to SR. Ideally, SR could be transparently added to any system without requiring a re-design or introducing new bugs. But, by increasing the separation between when an object is fetched and when it is used, SR can introduce a larger window for inconsistency. On the other hand, if consistency is carefully integrated with speculative replication, speculatively replicating updates to replace stale versions can improve availability for a given level of consistency [Yu and Vahdat 2001] and thus provide better trade-offs within the consistency versus availablity limits defined by Brewer's CAP impossibility result [Gilbert and Lynch 2002].

In this article, we present Safe Speculative Replication (SSR). SSR makes SR practical for widespread adoption through three distinguishing features. First, it enables self-tuning SR, i.e., it obviates manually tuned thresholds for resource management, making SR simple, more effective, and safe. Second, it enables a simple uniform architecture to integrate self-tuning SR with general consistency constraints. Third, SSRs emphasis on deployability enables easy integration with legacy infrastructure.

SSR separates the mechanisms for controlling how information flows through a system from the policy question of what information to replicate to what nodes. At the core of the SSR architecture are three key mechanisms. First, SSR disentangles three distinct flows: (1) invalidations that notify a replica when an update has occurred, (2) demand bodies that provide data in response to a demand read, and (3) speculative bodies that speculatively provide data in advance of a demand read. Second, SSR isolates speculative load from demand load using a priority scheduler at each resource and thus eliminates the undesirable effects of interference and overload. Third, a scheduler at each SSR cache or replica coordinates application of updates from these three separate flows to maximize performance and availability while meeting consistency and stalenes constraints.

To evaluate our architectural proposal, we present extensive case studies of two SR systems, NPS and TRIP, built on top of SSR. NPS [Kokku et al. 2003] is a Non-interfering Prefetching System for the Web that shows how SSR's self-tuning SR simplifies deployment, improves benefits, and reduces risks. For example, for a fixed workload and network conditions, SSR nearly matches the best performance available to a hand-tuned threshold-based system. But, when network conditions or workloads change from the training conditions, SSR's performance can be factors of two to five better than that of the static threshold system. TRIP [Nayate et al. 2004] (Transparent Replication through Invalidation and Prefetching) is a data replication middleware for constructing data dissemination services, a class of services where writes occur at an origin server and reads occur at a number of replicas. TRIP maintains sequential consistency semantics and, to the best of our knowledge, is the first system to integrate strong consistency with SR for WAN systems. TRIP shows that for some important workloads in a WAN, using SR makes it possible to simultaneously provide excellent consistency, performance, and availablity in spite of the CAP dilemma [Brewer 2001]. For example, for one large-scale trace workload, TRIP provides sequential consistency, improves performance by over a factor of four compared to demand caching, and often extends by orders of magnitude the duration over which a replica is able to mask network disconnections by serving locally-replicated data. TCP Nice [Venkataramani et al. 2002b] is a key component used by both NPS and TRIP to prevent network interference. We rigorously evaluate Nice in isolation through theory, simulations, and real world experiments over networks varying in capacity over four orders of magnitude, and show that it maintains noninterference while allowing background traffic to reap significant fractions of spare resources.

Finally, an architectual proposal must be easily deployable to be practical. Conceptually, SSR employs a priority scheduler for every resource in the system to prevent interference. However, in practice, large-scale systems are often complex, multitiered, and multidomain in nature and consist of black box components that are difficult to modify. Our case studies show that simple end-to-end mechanisms that require minimal modification to existing infrastructure are sufficient in practice. For example, NPS requires no modification to existing Web servers, networks, or browsers, and TRIP replicates consistent data via a file system interface so that a cluster of Web servers supporting a data-dissemination workload can transparently be distributed across a WAN as edge servers.

In summary, we make the following contributions in this article.

(1) We show that the dominant SR model based on thresholds is fundamentally flawed and makes resource management complex, inefficient, and risky.

(2) We present SSR, a self-tuning architecture that makes SR practical for large-scale distributed systems with general consistency constraints.

(3) We present TCP Nice, a sender-based transport protocol for low-priority transfers that requires no support from routers.

(4) We present NPS, the first non-interfering Web prefetching system that can be deployed without any modification to existing networks, Web servers, and browsers.

(5) We present TRIP, the first system that cleanly integrates sequential consistency with SR,

showing that SR can benefit distributed systems with consistency constraints in spite of the CAP dilemma.

The rest of the paper is organized as follows. Section 2 explains why threshold-based SR is flawed, section 3 presents SSR, section 4 presents TCP Nice, sections 5 and 6 present the NPS and TRIP case studies, section 7 describes related work, and section 8 summarizes lessons learned, open problems, and our conclusions.

2. THRESHOLD SR CONSIDERED HARMFUL

This section argues that threshold-based SR, the most commonly proposed SR scheme, is fundamentally flawed. Thresholds present three serious problems. First, they make systems complex to design and manage; second, they arbitrarily limit the benefits that SR can deliver; third, they expose a system to the risk of catastrophic overload.

Before proceeding further, we caution the reader against a common misperception that because speculatively replicating an object only saves a read miss upon the first access to an object (considering that subsequent accesses may reuse locally cached copies), SR can only incrementally improve user-perceived response time. However, observe that misses often dominate overall response time because local hits are rapidly served from a fast, nearby cache. Thus, using SR to improve the hit rate from say 60% to 95% may yield a six-fold reduction in miss rate and significant improvements to user-perceived response time.

2.1 Background

SR expends resources such as network bandwidth, storage space, and computing cycles to provide benefits such as performance, availability, and freshness of data. In particular, SR fetches an object in anticipation of future need, but if this prediction is wrong, then the resources used to fetch the object are wasted.

Thresholds appear to be a natural way to control resource consumption and relate the cost of SR to its benefit, and replicate when expected benefit exceeds cost. Commonly, a prediction algorithm such as prediction-by-partial-matching (PPM) [Curewitz et al. 1993] estimates the probability p that an object will be requested at some location in the future; then, a prefetching or pre-pushing system replicates objects whose probability of access p at a location exceeds a threshold t. The threshold t is then tuned to limit resource consumption and to maximize expected net benefit.

Figure 1(a) illustrates the intuition behind this methodology for Web prefetching. The figure itself is a cartoon, but similar graphs may be found in the literature (e.g., Padmanabhan and Mogul [Padmanabhan and Mogul 1996a] Figure 6). Decreasing the value of the threshold, i.e., increasing the number of objects eligible for prefetching, makes prefetching more effective in reducing miss rates, but also increases the network bandwidth consumed. A natural and common approach to balancing these effects [Duchamp 1999; Fan et al. 1999; Griffioen and Appleton 1994b; Padmanabhan and Mogul 1996a] is to pick a threshold value beyond which the improvement in response time no longer justifies the bandwidth cost of prefetching.

2.2 Problems

Although the threshold approach to limit resource usage can work well in a controlled experimental environment [Duchamp 1999; Fan et al. 1999; Griffioen and Appleton 1994b; Padmanabhan and Mogul 1996a], in practice, thresholds can make a deployed SR system

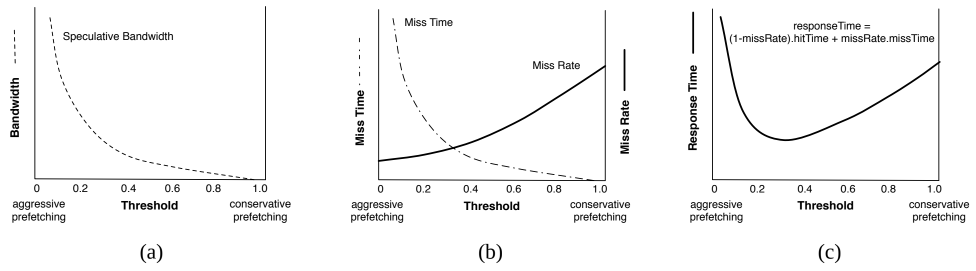


Fig. 1. Reducing the prefetch threshold (a) increases the bandwidth consumed by prefetching, as a result (b) reducing the threshold reduces miss rates but can increase miss times as network interference slows down demand requests. Since $responseTime = hitTime + missRate * missTime$ (c) for any given system configuration and workload, there is a critical threshold at which further reductions actually hurt overall response time.

complex, inefficient, and vulnerable to overload. The core of the problem is interference, which makes the relationship between various cost and benefit metrics nonmonotonic and difficult to map to a threshold. A simple threshold fails to capture the threshold v. net-benefit relationship for three reasons.

(1) **Non-monotonic** tradeoffs. As Figure 1 (b) and 1 (c) illustrate, lowering the threshold beyond a certain point can hurt performance because the ratio between the marginal improvement in hit rates and the marginal increase in resource consumption tends to fall as the threshold is lowered. As a result, for thresholds below some critical value, reducing thresholds will increase bandwidth more rapidly than it will increase hit rates and therefore will hurt rather than help overall performance.

   Note that these graphs understate the problem because they only illustrate *self-interference* in which prefetching can hurt the average response time of the application and node doing the prefetching. Another concern is *cross-interference* when prefetching by one node or application hurts the performance of another node or application.

(2) **Complex** tradeoffs. Determining this critical threshold is difficult because SR costs and benefits are not directly comparable. For example, in many systems the primary cost prefetching an object is network bandwidth, measured in bytes, while the primary benefit is improved response time, measured in seconds.

   An initally appealing strategy to balance costs and benefits is to convert both to a monetary value. We could, for example, extend the methodology used by Gray and Shenoy [Gray and Shenoy 2000] to analyze caching in WAN systems to analyze prefetching. A simple analysis (see extended version [Chandra 2001]) to compute a threshold where the marginal cost of resources for SR outweighs the marginal benefit in human wait-time reveals two insights. First, this approach is complex, requiring users or system administrators to provide accurate values for costs, benefits, and system constraints. Second, this approach yields 1-2 orders of magnitude more aggressive thresholds than typically proposed in the literature. We believe this result illustrates that even were designers willing to perform calculations such as these, the exercise would yield unrealistically low thresholds because they fail to address the other two problems discussed below.

(3) **Non-stationary** tradeoffs. The true interference cost of consuming a resource varies over a number of time scales. Workloads can vary at the granularity of seconds or less,

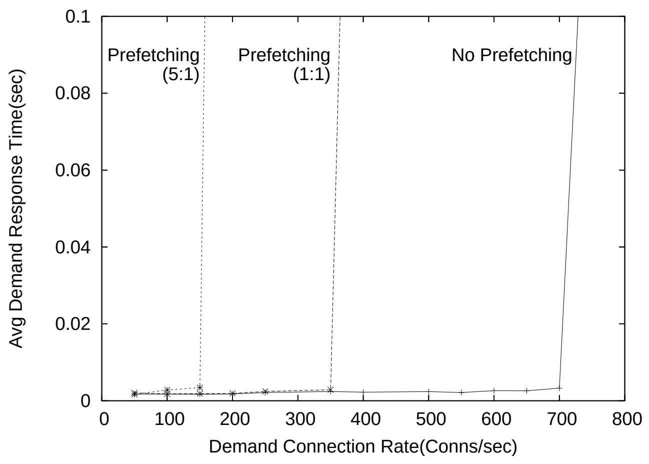


Fig. 2. Typical latency v. response time for demand requests with and without prefetching.

and speculatively replicating an object is much “cheaper” (i.e., cause less interference with demand requests) when the system is idle than when it is busy. For example, at the granulary of hours, speculative replication may be cheaper at night if workloads exhibit diurnal patterns of load. Finally, at the granularity of months and years, technology trends are causing the price of bandwidth, storage, and processing to fall significantly, so it may make sense to prefetch more aggressively next year than now.

This non-stationarity is thus a further challenge to threshold based systems. Even if the nonmonotonicity and complexity challenges were met and a perfect threshold set at one particular moment in time, that threshold will likely be incorrect at some points in the future.

(4) **Non-linear** tradeoffs. Threshold based schemes focus on controlling bandwidth consumption, but other important costs of prefetching—such as degraded response time for demand requests—may be non-linear with bandwidth. As a result, a small error in setting a threshold can result in an unacceptable risk of catastrophic overload of a system.

Network and server systems often behave like bounded queueing systems where, as load increases, the average response time for processing requests first increases gradually but then rises sharply as load approaches system capacity. In many systems, the performance degradation under overload is further magnified, compared to an ideal queueing system, by phenomena such as thrashing, high context switch overhead, and TCP’s exponentially backing off retransmission timers. As a result, a threshold that, say, doubles network bandwidth may impose negligible actual cost when load is low, but it may be catastrophic when system load is high.

The experiment presented in Figure 2 illustrates the problem. The graph shows the response time of requests on a Web server as a function of demand load while prefetching with different static thresholds. The label x on each line represents the number of prefetch requests issued for each demand request and roughly corresponds to a threshold of $1/x$, e.g., a value of 5 roughly corresponds to a static threshold of 0.2. In such a system static threshold prefetching offers a potentially unattractive tradeoff—it can provide significant improvements to performance [Duchamp 1999; Griffioen and Appleton

1994a; Padmanabhan and Mogul 1996a; Gwertzman and Seltzer 1995] during periods of low load, but it also makes the system significantly more vulnerable to unacceptable overload. We discuss this experiment in more detail in Section 5.

2.3 Implications

We believe the above four factors explain why prefetching systems have been more successful in laboratory experiments than in production deployments. The consequence of the first three factors is that it is difficult to choose the right trade-off for a threshold based system. The consequence of the fourth is that if one chooses an incorrect threshold, the negative consequences may be dramatic.

More broadly, the commonly-studied threshold prefetching approach (1) makes systems complex to design and manage because the threshold is difficult to set properly, (2) forces users to arbitrarily limit the benefits that SR can deliver by setting the threshold conservatively, and (3) exposes a system to the risk of catastrophic overload.

3. SSR: SELF-TUNING SR WITH CONSISTENCY

In this section, we present Safe Speculative Replication, a new SR architecture for large-scale distributed systems with general consistency constraints. SSR makes SR practical for widespread adoption through three distinguishing features. First, it enables *self-tuning* SR, i.e., it obviates manually tuned thresholds for resource management, making SR simple, more effective, and safe. Second, it enables a simple uniform architecture to integrate self-tuning SR with general consistency constraints. Third, SSR's emphasis on deployability enables easy integration with legacy infrastructure.

In the rest of this article, we show how SSR achieves these properties and discuss its implications for designing large-scale distributed systems. For simplicity of exposition, Section 3.1 first explains SSR's self-tuning resource management architecture. Then, Section 3.2 shows how a simple augmentation to this architecture integrates self-tuning SR with consistency constraints. Finally, the schedulers and case studies in Section 4, Section 5, and 6 describe how this architecture can be deployed in practice.

3.1 Basic Architecture

The key insight that enables the self-tuning property is a clean separation of *mechanisms* for scheduling resources for SR from *policies* for data selection and placement. As shown in Figure 3, SSR has two main components—a scheduler and a predictor—that address the mechanism and policy issues respectively.

3.1.1 *Predictor.* The predictor's task is to determine candidate objects for speculative replication. It does this by assigning priority values to objects that represent the estimated benefit of speculatively replicating an object at a location. The priorities thus impose a partial order on candidate objects. A typical priority metric is the probability of access of an object at a location.

In general, the predictor could employ an arbitrary ranking function to order objects by priority. Often predictors make use of history information. For example, a common and successful model is the partial-prediction based matching (PPM) model [Curewitz et al. 1993; Palpanas 1998] that develops a Markov model for predicting the objects likely to be accessed next given a history of the last n objects accessed at a location. One variation, PPN-$\frac{n}{w}$ [Kokku et al. 2003] predicts objects likely to be accessed in a window of the w

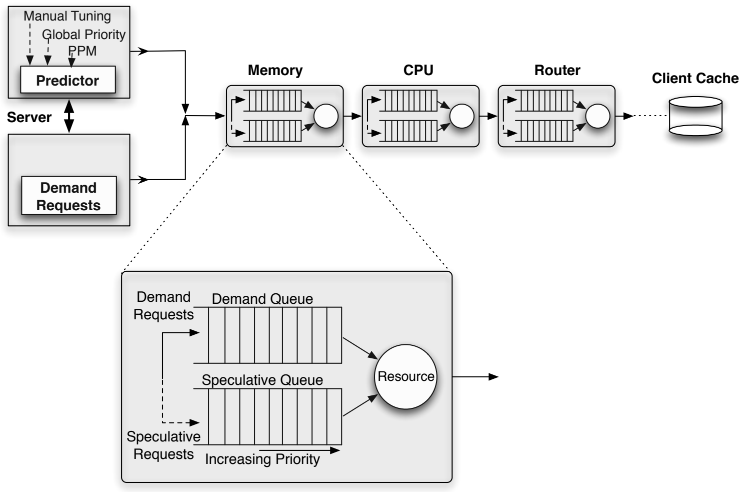


Fig. 3. SSR: Basic Architecture.

next references given a history of the last n references. More generally, predictors can use any relevant information such as domain-specific knowledge (e.g., for a news Web site, stories linked near the top of the page are more likely to be accessed than those linked at the bottom) or manually assigned hints (e.g., breaking news articles could be manually assigned a high priority even before the first access.) We term assignment of priorities to objects as a *speculative replication policy*, or simply a *prediction policy*.

3.1.2 *Scheduler.* The scheduler's task is to prevent interference between speculative and demand load. Conceptually, each resource in the system is equipped with a priority scheduler that maintains two queues, one each for demand and speculative load respectively. Demand load has strictly higher priority than speculative load, i.e., the scheduler assigns the resource to demand if there is any outstanding demand request and processes speculative load only if the demand queue is empty. Buffer space for the queues is allocated such that, during overload, speculative load is always discarded before demand.

Eliminating interference at each resource makes it unnecesary for the predictor to impose a predestined limit on how much can be speculatively replicated. A predictor could enqueue an arbitrarily large list of objects—in the extreme case including the entire content base—for speculative replication. The schedulers (1) ensure that speculative load does not interfere with demand requests, (2) allow any spare resource in the system to be aggressively used for SR, and (3) maximize SR benefit for a given resource consumption by processing speculative requests in priority order.

3.1.3 *Generalizations.* The basic architecture above can be extended to incorporate dynamically computed priorities where, as new information about access patterns becomes available, objects enqueued for SR can have their priorities modified and be reinserted. In practice, the queues for speculative load at downstream servers should be kept small in order to prevent outdated speculative objects from accumulating during periods of low spare capacity and stalling useful objects from being replicated in the future. Another

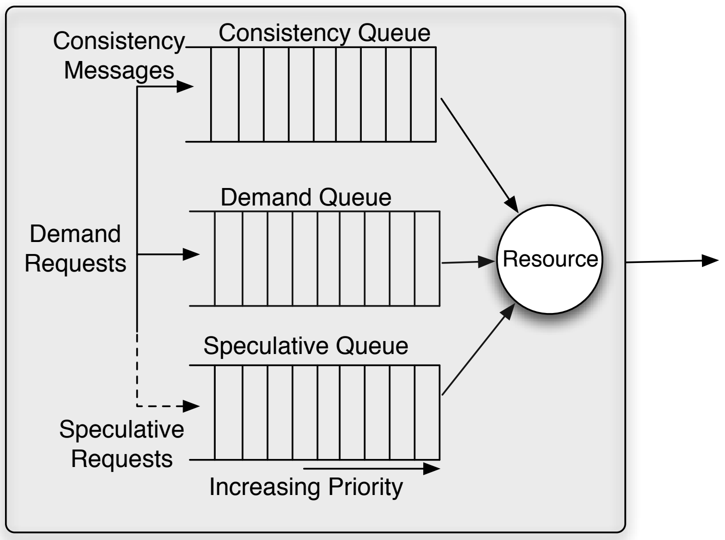


Fig. 4. SSR with Consistency

alternative is to decay the priorities of speculative objects in interior queues with time to prevent stalling of newer high-priority objects.

In the special case of scheduling client cache storage, both speculative and demand objects may utilize the same cache. When the cache becomes full, the lowest priority object, e.g., the demand-fetched or speculatively-fetched object with the highest expected time to next access is evicted. Replacement policies are not a focus of this paper; Patterson et al. [Patterson et al. 1995b] and Kimbrel et al. [Kimbrel et al. 1996] provide good discussions of how to share storage between demand-cached and SR data.

3.1.4 *Summary of Benefits.* SSR offers the following benefits over threshold SR.

(1) SSR simplifies the design, deployment, and maintenance of large-scale SR systems by cleanly separating the concerns of prediction and scheduling, which (a) simplifies prediction design by removing the need to account for resource limitations and (b) simplifies the resource management problem by eliminating the need to manually control how many resources are consumed by SR.
(2) SSR can yield greater benefits because there are no a priori limits to its resource usage. The predictor can enqueue an arbitrarily long list of objects for SR. When spare resources are abundant the system replicates aggressively to maximize benefits, but when few spare resources are available, the system throttles SR to avoid interference.
(3) SSR protects the system from the risk of overload as the noninterfering mechanism automatically adapts the intensity of SR to only use spare resources in the system.

3.2 SSR with Consistency

To provide cache consistency, a client cache should stop using an object when the object is updated at the server. SR can increase the delay between when an object is fetched and

	In-Order	Priority
Consistency	Required	high
Demand	Not required	high
SR	Not required	low

Table I. Per-channel properties.

when it is used, and thereby increase the need to carefully manage consistency. At the same time, SR offers the opportunity to improve a system's trade-offs between consistency and performance or availability by prefetching or pre-pushing updated content, thereby potentially obviating a refresh on demand. Note that the CAP impossibility result [Brewer 2001; Siegel 1992; Gilbert and Lynch 2002] says that a distributed system with strong consistency requirements can not achieve full availability over a partitionable network.

SSR deployments can provide *sequential consistency* [Lamport 1979] with bounded *staleness* for *information dissemination workloads* in which a central server is the source of all writes [Nayate et al. 2004]. By supporting the strong guarantee of sequential consistency, SSR enables *transparent replication*: the results of all read and write operations are consistent with an order that could legally occur in a centralized system, so—absent time or other communication channels outside of the shared state—a program that is correct for all executions under a local model with a centralized storage system remains correct when using SSR. In the special case of dissemination services, allowing a client's view of stored objects to be stale by a bounded amount retains sequential consistency. Nevertheless it alleviates the CAP dilemma [Brewer 2001; Siegel 1992] by allowing nodes to delay invalidating some of their data and thereby maximize the amount of consistent data stored locally.

SSR systems typically use invalidation-based consistency, which can provide better consistency at a lower cost than polling [Howard et al. 1988; Liu and Cao 1997; Yin et al. 1999] and which can scale to Internet-scale systems [Yin et al. 2002b]. There are two key ideas. First, SSR transmits consistency metadata on a logically separate channel from the bodies of updates. Second, SSR allows a receiver to buffer an invalidation before applying it to enable a tunable trade-off between availability and performance on one hand and consistency on the other.

As shown in Figure 4, consistency metadata, such as invalidates, two-phase commit protocol messages, accept stamps, version vectors etc., are propagated via a logically separate transmission channel from demand and SR bodies. As Table I summarizes, invalidations are propagated via an in-order, high-priority channel. An in-order channel is required to ensure that caches can maintain a consistent view of their storage. Conversely, demand and SR traffic can be safely reordered by the network which, for example, would allow the use of high-performance multi-path bulk transfer protocols for large objects [Kostic et al. 2003]. Or, for simplicity, implementations may choose to merge the demand and consistency channels and to have a separate low priority SR channel.

At the receiving cache or data replica, SSR must coordinate how it applies data from the three incoming streams (1) to ensure that data from these separate channels do not violate consistency and (2) to maximize the amount of valid data in the cache to maximize availability and performance. To meet the first goal, an SSR cache buffers any demand or SR bodies that arrive before the corresponding invalidation. To meet the second goal, an SSR cache buffers incoming invalidations until all preceeding invalidations have been applied and either (a) the corresponding body has arrived or (b) a configurable maximum delay Δ has passed since the invalidation was issued by the server. We discuss how these features are used in more detail in Section 6.

3.3 Is SSR Practical?

Ease of deployment is crucial for an architectural proposal to be practical. Although, SSR conceptually employs a priority scheduler to prevent interference at each resource in the system, in practice, it is difficult to modify large-scale legacy distributed systems. For example, Web servers are often multitiered, consisting not only of a highly optimized Web server, but also several other complex black box components such as application servers, virtual machines, backend database servers, a storage area network etc. Similarly, WANs often consist of several autonomous domains that make it difficult to enable any scheduling mechanism more sophisticated than simple best-effort delivery.

In the case studies we examine in subsequent sections, we show how using simple end-to-end mechanisms that require little or no modification to existing network and server infrastructure, it is possible to instantiate SSR for building useful WAN services or augmenting legacy distributed systems. Instead of considering each resource separately, we group a WAN system into three subsystems, namely, the server, network, and client, and develop noninterfering SR mechanisms by treating each as a black box. An example of such a mechanism for the network is TCP Nice, a sender-based low priority transport protocol explained in the next section. In our NPS (§5) case study, we present a noninterfering Web prefetching system based on TCP Nice that works with unmodified Web servers and client machines. In the TRIP case study (§6), we present a middleware that enables replication and prefetching in otherwise centralized applications with consistency requirements in a transparent manner, i.e., without requiring application programmers to be consistency-aware.

4. TCP NICE: A PRIORITY SCHEDULER FOR NETWORK RESOURCES

In this section, we present a low-priority scheduler for network resources. Our goal is to manage network resources in order to provide a simple abstraction of low-priority or *background* transfers that do not interfere with on-demand or *foreground* transfers. Because it is difficult to modify network routers across thousands of administrative domains, we provide this abstraction by modifying only the transport protocol at the sender.

Our solution, TCP Nice [Venkataramani et al. 2002c], dramatically reduces the interference inflicted by background flows on foreground flows. Nice does so by modifying TCP congestion control to be more sensitive to congestion than traditional protocols such as Reno [Jacobson 1988] or Vegas [Brakmo et al. 1994] 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.

4.1 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 the network has spare capacity.

4.1.1 *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 [Chiu and Jain 1989; Jacobson 1988].

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 [Brakmo et al. 1994], 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. Finally, to mainatin TCP-friendliness, Vegas performs a multiplicative decrease upon a packet loss. Vegas estimates the number of packets queued at the bottleneck router by bounding the difference between the actual and expected throughput corresponding to the current window size and observed round-trip delay. 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 upon the "β 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.

4.1.2 *TCP Nice.* The Nice extension adds three components to AIMD congestion control: 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 a fixed fraction f of the packets that Nice sends in a round encounter delays exceeding $(1 - t) \cdot \textit{minRTT} + t \cdot \textit{maxRTT}$ for a fixed fraction t, our detector signals congestion. After the first rount-trip delay estimate, *maxRTT* is initialized to $2 \cdot \textit{minRTT}$. Round-trip delays of packets are indicative of the current bottleneck queue size and the t represents the fraction of the total queue capacity that starts to trigger congestion. The Nice congestion avoidance mechanism incorporating the *interference trigger* with constants t and fraction f can be written as follows (*curRTT* is the round-trip delay experienced by each packet):

```
per ack operation:
    if ($\textit{curRTT} > (1 - t) \cdot \textit{minRTT} + t \cdot \textit{maxRTT}$)
        $\textit{numCong}$++;
per round operation:
    if ($\textit{numCong} > f \cdot W$)
        $W \leftarrow W/2$
    else {
        ...   // TCP-friendly congestion avoidance follows
    }
```

If the congestion condition does not trigger, Nice falls back on TCP-friendly congestion avoidance rules like AIMD in Reno or delay-based AIAD and loss-based AIMD in Vegas. Nice is agnostic to the specific choice of the TCP-friendly foreground protocol.

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 transmit a packet 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 noninterference 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 [Morris 1997].

When a Nice flow signals congestion, it halves its current congestion window. In contrast, Vegas reduces its window by one packet for 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 mathematical analysis as well as our experimental results show that even with these aggressively timid policies, we achieve reasonble levels of utilization in practice.

As in TCP Vegas, maintaing 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 [Acharya and Saltz 1996; Sanghi et al. 1993] 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. Therefore, we use the fifth and ninety-fifth percentile values and observe that it improves the robustness of this algorithm. Route changes during a transfer can also contribute to inaccuracies in RTT estimates. However such changes are rare [Paxson 1996] within the lifetime of a typical connection. However, since Nice is designed for long-lived background flows, it is important to address this situation, and we do so by maintaining exponentially decaying averages for *minRTT* and *maxRTT* estimates.

4.1.3 *Prototype Implementation.* We implemented 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 the congestion detector. In our implementation, we retain the Vegas parameters α and β to 1 and 3 respectively. We remark that the detector in the previous section may be implemented on top of any existing TCP-friendly protocol. We chose Vegas only because our initial attempts at designing Nice were centered around tweaking Vegas to be less interfering.

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 $f = 0.5$ packets during a round encounter delays exceeding $t = 0.2$. Our analysis and experiments suggest that the interference is insensitive to the exact value of t so long as it is comfortably less than 0.5. The fraction f does not enter directly into our analysis, and our experimental studies interference is insensitive to the exact value of f; it only serves as a reliable indicator of increasing delays. 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 [Aggarwal et al. 2000] 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.

4.2 Formal 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 analyse 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 maxi-

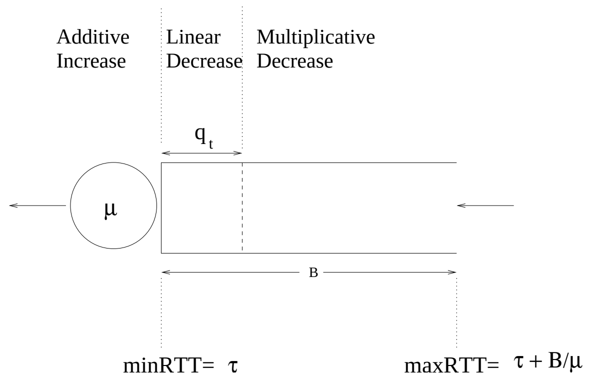


Fig. 5. Queue dynamics at a bottleneck router with Nice

mum 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.

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 5. 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, all of which attempt to transfer a single large file from the source to the 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 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. Models assuming flow synchronization have been used in previous analyses [Bonald 1998]. We model only the congestion avoidance phase to analyze the steady-state behaviour.

We obtain a bound on the reduction in the throughput of Reno flows due to the presence of Nice flows by analysing 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 extended technical report [Venkataramani et al. 2002a].

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. The total window size, $W(t)$, is $W_r(t) + W_n(t)$. Let the number of Reno and Nice flows be denoted by m and l respectively. 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 $m > l$. The window dynamics in any period can be split into three intervals as described below.

(1) *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.

(2) *Additive Increase, Additive Increase or Decrease*: This interval $[t_1, t_2]$ is marked by additive increase of W_r, but additive decrease of W_n if the underlying congestion avoidance mechanism is Vegas and additive increase if it is Reno. The end of this interval is marked by $W(t_2) = \mu\tau + q_t$.

(3) *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 [Venkataramani et al. 2002a] that the interference of Nice is bounded as follows.

Theorem 1: The interference I, defined as the fractional loss in throughput experienced by TCP flows in the presence of Nice flows, is bounded as follows:

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

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 intuitively reflects the time that Nice has to multiplicatively back off before packet losses occur. Intuititively, 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 anamolously under such circumstances [Morris 1997]. 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 comfortably less than 1, 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.

4.3 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, does Nice 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.

4.3.1 *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. 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.

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 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.

4.3.2 *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

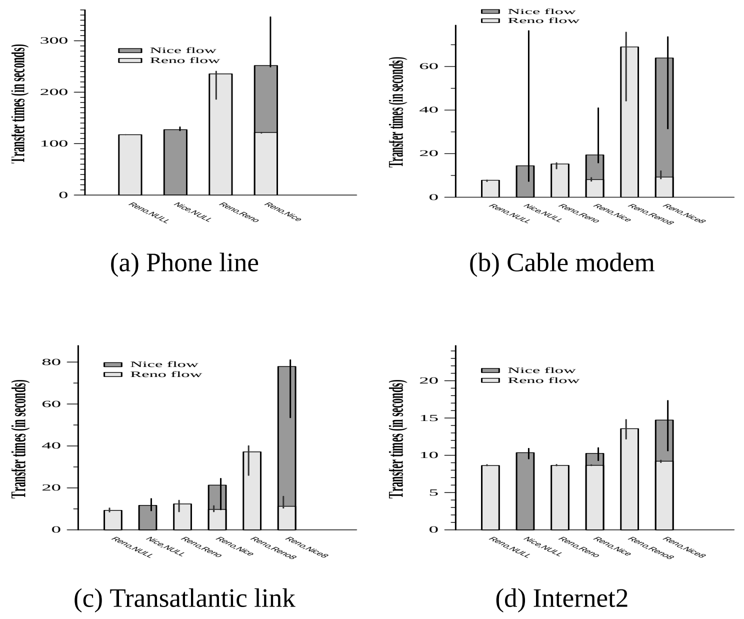


Fig. 6. Large flow transfer performance. Each bar represents the average transfer time observed for the specified combination of primary/secondary transfers. Empty bars represent the average time for a Reno flow. Solid bars represent the average time for a Nice flow. The narrow lines depict the minimum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

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.

Other Internet experiments (refer extended version [Venkataramani et al. 2002d]) conducted over a week-long period show that Nice can achieve useful amounts of throughput in a noninterfering manner throughout the day. In addition to experimentation over real networks, we also stress tested Nice via simulations over a wide range of parameters for spare capacity, foreground traffic profiles, and active queue management schemes that support the above results.

4.3.2.1 *Summary.* In summary, Nice achieves our goals of noninterference and practical levels of utilization of spare capacity over real networks varying in capacity by four orders of magnitude. The key ideas that enable these properties are early congestion detection and more aggressive backoff than AIMD TCP. It is somewhat surprising that a transport protocol operated only at the sender end can emulate two levels of differentiated services, a service that is otherwise considered a part of the network layer requiring implemention at every single router.

5. WEB PREFETCHING CASE STUDY

In this section, we present a concrete instantiation of the SSR architecture, NPS – a **N**oninterfering and deployable Web **P**refetching **S**ystem. We group resources into three subsystems — the server, network, and client — that also represent administratively separate domains. Imposing a priority scheduler for each resource prevents interference, however, the challenge is to build deployable schedulers. For example, one systematic "clean-slate" solution might be to extend HTTP to include a "GET-PREFETCH" request that Web browsers process in the background, enable differentiated services for low priority transport in routers, and employ resource containers [Banga et al. 1999] at Web servers to prevent prefetch traffic from interfering with demand traffic. Unfortunately, this strawman warants an unacceptable level of change to legacy infrastructure. The increasing complexity of Web servers [Gribble et al. 2002; IBM ; Intel ; Resonate Inc ; Zeus Technology], the lack of architectural consensus among network service providers [Anderson et al. 2005; GENI], and the large number of Web browsers in use makes clean-slate solutions impractical. Our goal therefore is to instantiate SSR in a manner that involves no change to HTTP, Web browsers, routers, and Web servers.

NPS treats the server and network as a black box and employs an external probing mechanism to estimate spare capacity and limit the prefetch load accordingly. In both cases, the probing mechanism uses response time as an indicator of spare capacity. For the network, this mechanism is TCP Nice introduced in the previous section. Similarly, for the server, increasing response times for prefetch requests indicate high demand load and cause the prefetching module to aggressively back off, while low response times actuate a measured increase in prefetching intensity. Finally, to avoid client interference, NPS uses simple heuristics to control resources used by prefetching. To work with existing browsers, NPS modifies HTML pages to include JavaScript code to issue prefetch requests, and wraps the server infrastructure with simple external modules that require no knowledge of, or no modifications to the internals of existing servers.

Our experiments with an NPS prototype under traces of real web workloads indicate that it is both efficient and non interfering under different network and server load conditions. For example, in our experiments, when the network and the server are lightly loaded, NPS and a manually tuned threshold-based prefetching scheme improve the response times by 28% and 41% respectively. On the other hand, when the network gets heavily loaded with little spare capacity, we observe that the same manually tuned scheme causes response times to increase by a factor of 7 due to interference, whereas NPS self-tunes prefetch intensity to contain this increase to less than 30%. In the rest of this section, we explain in detail the design and implementation of the above mechanisms in NPS.

5.1 NPS design

There appears to be a consensus among researchers on a high level architecture for prefetching in which a server sends a list of objects to a client and the client issues prefetch requests for the objects on the list [Chen and Zhang 2001; Markatos and Chronaki 1998; Padmanabhan and Mogul 1996b] . This division of labor allows servers to use global object access patterns and service-specific knowledge to determine what should be prefetched, and it allows clients to filter requests through their caches to avoid repeatedly fetching objects. NPS follows this organization and seeks to meet two other important requirements: (1) self tuning resource management and (2) deployability without modifying existing protocols,

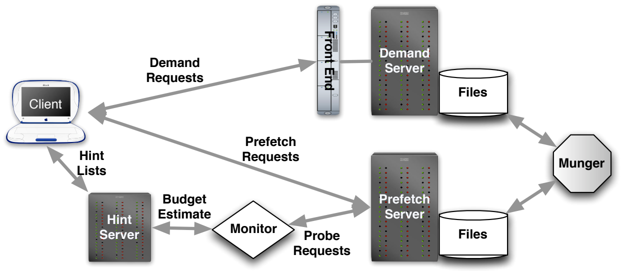


Fig. 7. NPS architecture

clients, proxies, or servers.

Figure 7 provides an overview of our design. A server machine hosts both a *Demand Server* and a *Prefetch Server*. Each is an unmodified HTTP server, but they are running on different ports in order to ensure that prefetch replies and demand replies are sent on different network connections. A *Hint Server* sends suggestions of what to prefetch to *Clients*. The *Monitor* is coupled with the *Hint Server* to throttle the number of prefetch hints issued to clients. The *Client* is an unmodified Web browser that issues both demand and prefetch requests.

NPS is designed to avoid interference at the server, network, and client as follows.

(1) To avoid interference at the server, the *Monitor* monitors end-to-end server load and throttles the number of prefetch hints issued by the *Hint Server* to restrict the prefetch load *Clients* can impose. We discuss this process in Section 5.2.1.

(2) To avoid interference in the underlying network, NPS (1) ensures that prefetch and demand requests appear on different network connections by using separate *Demand Server* and *Prefetch Server* processes on different ports and (2) uses TCP Nice for prefetch connections.

(3) To control resource usage at the client, NPS uses some simple heuristics. First, we ensure that a client does not start prefetching until after it has finished loading the demand page. Second, to limit cache pollution, we set prefetched objects to expire from the cache earlier than they would otherwise. A better approach to limiting cache pollution would be to use techniques from Transparent Informed Prefetching [Patterson et al. 1995a] to balance caching and prefetching, but this would require modification of existing browsers. Initial measurements suggest that our simple heuristics work reasonably well [Kokku et al. 2003], however, we do not examine resource management at the client in detail in this article.

Deployability concerns constrain our design: our system is designed to work with unmodified Web servers, Web clients, and network routers. We induce an unmodified Web *Client* to prefetch using a *Munger* that modifies Web pages to include JavaScript that causes the *Client* to (i) repeatedly fetch lists of URLs to prefetch from the *Hint Server*, and (ii) prefetch those URLs from the *Prefetch Server*. A challenge is to get an unmodified browser to allow objects prefetched from one server (the *Prefetch Server*) to act as cache hits for

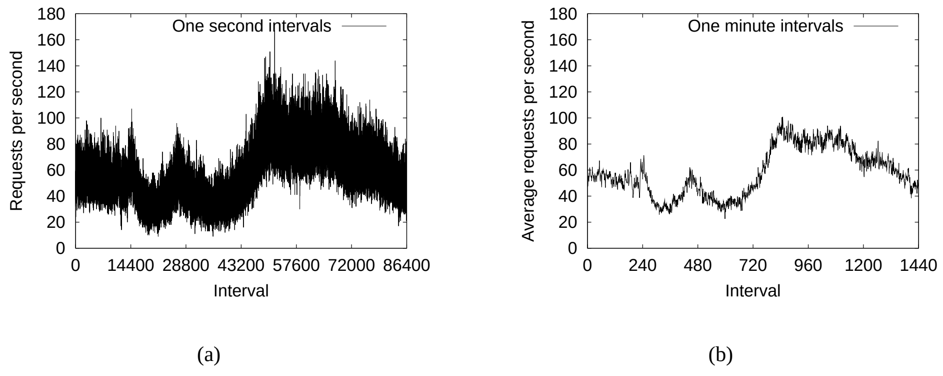


Fig. 8. Server loads averaged over (a) 1-second and (b) 1-minute intervals for the IBM sporting event workload.

requests to a different server (the *Demand Server*). The *Front End* and the *Munger* conspire to re-write URLs to make this work. The details of how NPS re-writes Web pages to (a) trigger prefetching and (b) allow cross-server cache hits are outside the scope of this article but appear in an extended version of a paper describing NPS [Kokku et al. 2003], where we also discuss how this design would differ if deployability using unmodified clients and servers were less of a concern [Kokku et al. 2003]. Note that our approach works *as is* even in the presence of proxy caches between clients and servers; the first client behind a proxy would prefetch from the server (loading both its cache and the proxy cache), and the rest of the clients would prefetch from the proxy cache (loading their caches).

5.2 Server Interference

An ideal system for avoiding server interference would cause no delay to demand requests in the system and utilize all spare resources on servers for prefetching. Such a system must tolerate and exploit changing workload patterns. However, Web traffic at a server is often bursty at several different time scales [Crovella and Bestavros 1996] and peak rates typically exceed the average rate by an order or two of magnitude [Mogul 1995]. For example, Figure 8 shows the request load on an IBM server hosting a major sporting event during 1998 averaged over 1-second and 1-minute intervals. It is crucial for a prefetching system to be responsive to such fluctuations to balance utilization and risk of interference.

Our approach to adapting to server load in a deployable way is to use an end-to-end monitoring unit that estimates the overall load (or spare capacity) on the server by occasionally probing the server with representative requests and measuring the response times of the replies. Low response times indicate that the server has spare capacity and high response times indicate that the server is loaded. Based on such an estimate, the monitor utilizes the spare capacity on the server by controlling the number and aggressiveness of prefetching clients.

An advantage of end-to-end monitoring is that it requires no modifications to existing servers. The choice of the probing interval, however, strikes a tradeoff between the precision of the monitor and the load imposed by the probes themselves.

In the following subsections, we discuss the issues involved in designing an end-to-end monitor, present our simple monitor, and evaluate its efficacy in comparison to server scheduling.

5.2.1 *End-to-end Monitor Design.* The monitor estimates the server's spare capacity and sets a *budget* of prefetch requests permitted for an interval. The hint server adjusts the load imposed by prefetching on the server by ensuring that the sum across the hint lists returned to clients does not exceed the budget. Our monitor design must address two issues: (i) budget estimation and (ii) budget distribution across clients.

5.2.1.1 *Budget estimation.* The monitor periodically probes the server with HTTP requests to representative objects and measures the response times. The monitor increases the budget when the response times are below the objects' *threshold* values and decreases the budget otherwise.

As probing is an intrusive technique, choosing an appropriate rate of probing is a challenge. A high rate makes the monitor more reactive to load on the server, but also adds extra load on the server. On the other hand, a low rate makes the monitor react slowly, and can potentially lead to interference to the demand requests. Similarly, the exact policy for increasing and decreasing the budget must balance the risk of causing interference against underutilization of spare capacity.

5.2.1.2 *Budget distribution.* The goal of this task is to distribute the budget among the clients such that (i) the load due to prefetching on the server is contained within the budget for that epoch and is distributed uniformly over the interval, (ii) a significant fraction of the budget is utilized over the interval, and (iii) clients are responsive to changing load patterns at the server. The two knobs that the hint server can manipulate to achieve these goals are (i) the size of the hint list returned to the clients and (ii) the subset of clients that are given permission to prefetch. This flexibility provides a freedom to choose from many policies.

5.2.2 *Monitor Prototype.* Our prototype uses simple, minimally tuned policies for budget estimation and budget distribution. Future work may improve the performance of our monitor.

The monitor probes the server in epochs, each approximately 100 ms long. In each epoch, the monitor collects a response time sample for a representative request. In the interest of being conservative – choosing non-interference even at the potential cost of reduced utilization – we use an additive increase (increase by 1), multiplicative decrease (reduce by half) policy. AIMD is commonly used in network congestion control [Jacobson 1988] to conservatively estimate spare capacity in the network and be responsive to congestion. If in five consecutive epochs, the five response time samples lie below a threshold, the monitor increases the budget by 1. While taking the five samples, if any sample exceeds the threshold, the monitor sends another probe immediately to check if the sample was an outlier. If the new sample exceeds the threshold, indicating a loaded server, the monitor decreases the budget by half and restarts collecting the next five samples.

In our simple prototype, we manually supply the representative objects's threshold response times. However, setting this value is straightforward because of the predictable pattern in which response times vary with load on server systems – a nearly constant value of response time for low load followed by a sharp rise beyond the "knee" for high load. As part of our future work, we intend to make the monitor automatically pick thresholds in a self-tuning manner.

The hint server distributes the current budget among client requests that arrive in that epoch. We choose to set the hint list size to the size of one document (a document corresponds to a HTML page and all embedded objects). Our policy lets clients to return quickly for more hints and thus be more responsive to changing load patterns on the server. Note that returning larger hint lists would reduce the load on the hint server, but it would reduce the system's responsiveness and its ability to avoid interference. We control the number of simultaneously prefetching clients, and thus the load on the server, by returning to some clients a hint list of zero size and a directive to wait until the next epoch to fetch the next hint list. For example, if B denotes the budget in the current epoch, and N the expected number of clients in that epoch, D the number of files in a document, and τ the epoch length, the hint server accepts a fraction $p = min(1, \frac{B \cdot \tau}{N \cdot D})$ of requests to prefetch on part of clients in that epoch and returns hintlists of zero length for other requests. Note that other designs are possible. For example, the monitor can integrate with the prefetch prediction algorithm to favor prefetching by clients for which the predictor can identify high-probability items and defer prefetching by clients for which the predictor identifies few high-value targets.

Since the hint server does not a priori know the number of client requests that will come in an epoch, it estimates that value with the number of requests that come in the previous epoch. If more than the estimated number of requests arrive in a epoch, the hint server replies with list of size zero and a directive to retry in the next epoch to those extra requests. If fewer clients arrive, some of the budget can get wasted. However, in the interest of avoiding interference, we choose to allow such wastage of budget.

5.2.3 *Evaluation of the Monitor.* In this subsection, we evaluate the monitor in isolation primarily concerning ourselves with its interference and utilization of spare capacity, and less with the effectiveness of the prediction policy in consuming spare capacity usefully. (In the next section, we evaluate end-to-end interference as well as response-time benefits of NPS in its entirety.) We therefore abstract away prediction policies by prefetching dummy data. Our experiments compare the effectiveness of different resource management alternatives in avoiding server interference against the case when no prefetching is done with respect to the following metrics: (i) *cost:* the amount of interference in terms of demand response times and (ii) *benefit:* utilization of spare capacity.

We consider the following resource management algorithms for this set of experiments:

(1) No-Prefetching: Ideal case, when no prefetching is done or when we use a separate prefetching infrastructure.
(2) No-Avoidance: Prefetching with no interference avoidance with fixed aggressiveness. We set the aggressiveness by setting *pfrate*, which is the number of documents prefetched for each demand document. For a given service, a given prefetch threshold will correspond to some average *pfrate*. We use fixed *pfrate* values of 1 and 5.
(3) Scheduler: As a simple local server scheduling policy, we choose *nice*, the process scheduling utility in Unix. We again use fixed *pfrate* values of 1 and 5. This simple server scheduling algorithm is only intended as a comparison; more sophisticated local schedulers may better approximate the ideal case.
(4) Monitor: We perform experiments for two threshold values of 3ms and 10ms.

For evaluating algorithms 2 and 4, we set up one server serving both demand and prefetch requests. Our prototype implementation of algorithm 3 requires that the demand and prefetch requests be serviced by different processes. We use two different servers listening on two ports on the same machine, with one server run at a lower priority using the Linux *nice*. For experiment 4, we also run two servers on the same machine, but both run

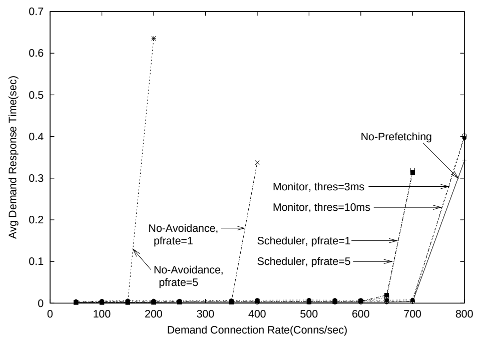


Fig. 9. Effect of prefetching on demand throughput and response times with various resource management policies

at the same priority. Our experimental setup includes Apache HTTP server [Apache HTTP Server Project] running on a 450MHz Pentium II, with 128MB of memory. To generate the client load, we use httperf [Mosberger and Jin 1998] running on four different Pentium III 930MHz machines. All machines run the Linux operating system.

We use two workloads in our experiments. Our first workload generates demand requests to the server at a constant rate. The second workload is a one hour subset of the IBM sporting event server trace, whose characteristics are shown in Figure 8 and discussed in more detail in [Challenger et al. 1999]. We scale up the trace in time by a factor of two, so that requests are generated at twice the original rate, as the original trace barely loads our server.

5.2.3.1 *Constant workload.* Figure 9 shows the demand response times with varying demand request arrival rate. The graph shows that both Monitor and Scheduler algorithms closely approximate the ideal behavior of No-Prefetching in not affecting the demand response times. Conversely, the No-Avoidance algorithm with fixed *pfrate* values significantly damages both the demand response times and the maximum demand throughput.

Figure 10 shows the bandwidth achieved by the prefetch requests and their effect on the demand throughput bandwidth. The figure shows that No-Avoidance adversely affects the demand throughput bandwidth. Conversely, both Scheduler and Monitor reap spare bandwidth for prefetching without much decrease in the demand bandwidth. Further, at low demand loads, a fixed pfrate prevents No-Avoidance from utilizing the full available spare bandwidth. The problem of too little prefetching when demand load is low and too much prefetching when demand load is high illustrates the fundamental drawback of existing threshold strategies. As hoped, the Monitor tunes prefetch aggressiveness of the clients to utilize most of the spare capacity.

5.2.3.2 *IBM server trace.* In this set of experiments, we compare the performance of the four algorithms for the IBM server trace. Figure 11 shows the demand response times and prefetch bandwidth in each case. The graph shows that the No-Avoidance case affects the demand response times significantly as *pfrate* increases. The Scheduler and Monitor cases have less adverse effects on the demand response times.

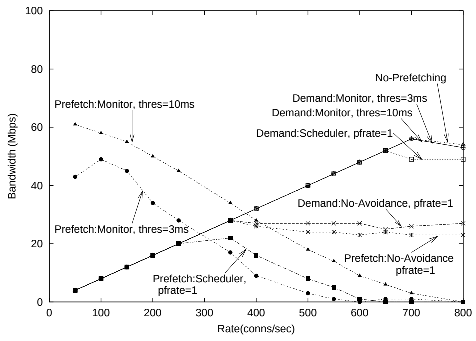


Fig. 10. Prefetch and demand throughput achieved by various algorithms

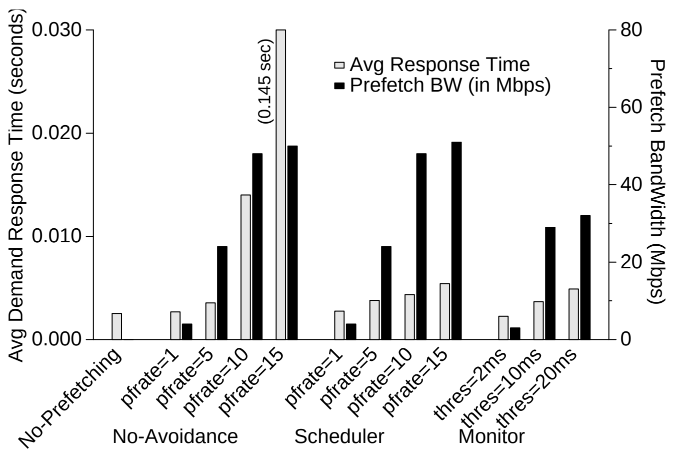


Fig. 11. Performance of No-Avoidance, Scheduler and Monitor schemes on the IBM server trace

These experiments show that resource management is an important component of a prefetching system because overly aggressive prefetching can significantly hurt demand response time and throughput while timid prefetching gives up significant bandwidth. They also illustrate a key problem with constant non-adaptive magic numbers in prefetching such as the threshold approach that is commonly proposed. The experiments also provide evidence of the effectiveness of the monitor in tuning prefetch aggressiveness of clients to reap significant spare bandwidth while keeping interference at a minimum.

5.3 NPS Prototype and End-to-End Evaluation

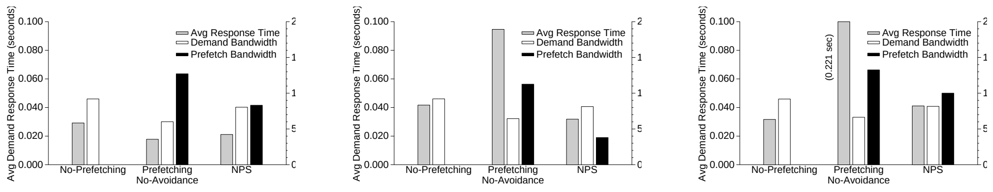
Our prototype uses the architecture whose prefetching mechanism is shown in Figure 7. We use Apache 2.0.39 as the server, hosted on a 450MHz Pentium II, serving demand requests on one port and prefetch requests on the other. As an optimization, we implement the frontend as a module within the Apache server rather than as a separate process. The hint server is implemented in Java and runs on a separate machine with 932 MHz Pentium III proessor, and connects to the server over a 100 Mbps LAN. The hint server uses prediction lists generated offline using the PPM algorithm [Padmanabhan and Mogul 1996b] over a complete 24 hour IBM server trace. The monitor runs as a separate thread of the hint server on same machine. The content munger is also written in Java and modifies the content offline. The munger takes 2.6 ms per KB on an average for each file on a 2.4 GHz Pentium 4. We have successfully tested our prefetching system with popular web browsers inluding Netscape, Internet Explorer, and Mozilla.

5.3.1 *End-to-End Performance.* In this section, we evaluate NPS under various setups and evaluate the importance of each component in our system. In all setups, we consider three cases: (1) No-Prefetching, (2) No-Avoidance scheme with fixed *pfrate*, and (3) NPS (with Monitor and TCP Nice). In these experiments, the client connects to the server over a wide area network through a commercial cable modem link. On an unloaded network, the round trip time from the client to the server is about 10 ms and the bandwidth is about 1 Mbps.

We use httperf to replay a subset of the IBM server trace. The trace is one hour long and consists of demand accesses made by 42 clients. This workload contains a total of 14044 file accesses of which 7069 are unique; the average demand network bandwidth is about 92 Kbps over this hour. We modify httperf to simulate the execution of JavaScript. Also, we modify httperf to implement a large cache per client that never evicts a file that is fetched or prefetched during a run of an experiment. In No-Avoidance case, we set the pfrate to 70, i.e. it gets a list of 70 files to prefetch, fetches them and stops. This pfrate corresponds to a prefetch threshold of 0.1 for the PPM algorithm; i.e. each file included in the list has probability of access greater than 0.1. This pfrate is hand tuned such that neither the server nor the network becomes a bottleneck even for the No-Avoidance case. The hint server gives out hint lists of size 10 each time a client requests more candidate objects to prefetch. Note that many of the files given as hints could be cache hits at the client for either the No-Avoidance or the NPS case.

5.3.1.1 *Unloaded resources.* In this experiment, we use the setup explained above. Figure 12(a) shows that when the resources are abundant, both No-Avoidance and NPS cases improve the average response times by 41% and 28% respectively. Observe that it is the relative improvement that matters—the absolute improvement appears small because of our experimental setup where the round-trip time between the clients and the server is about 10ms; the demand response times are significantly higher (hundreds of milliseconds to seconds) in real Web scenarios. NPS achieves less improvement than No-Avoidance because of its conservativeness in the interest of non-interference. The graph also shows the prefetch bandwidth consumed by No-Avoidance and NPS.

5.3.1.2 *Loaded server.* This experiment demonstrates the effectiveness of the monitor as an important component of NPS. To create a loaded server condition, we use a client machine connected on a LAN to the server running httperf that replays a heavier subset of the IBM trace and also prefetches like the WAN client. Figure 12(b) plots the average demand response times and the bandwidth used in the three cases. As expected, even though the server is loaded, the clients prefetch aggressively in the No-Avoidance case, thus causing the demand response times to increase by more than a factor of 2 rather than



(a) Unloaded Resources (b) Loaded Server (c) Loaded Network

Fig. 12. Effect of prefetching on demand response times.

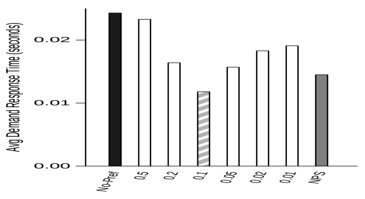


Fig. 13. Manual Tuning

decrease. However, despite the heavy load on the server, NPS (with the help of the monitor) is able to detect and exploit periods of relatively low load and improve the average response times by about 25%.

5.3.1.3 *Loaded network.* This experiment demonstrates the effectiveness of TCP Nice as a building block of NPS. In order to create a heavily loaded network with little spare capacity, we set up another client machine running httperf that shares the cable modem connection with the original client machine, replays the same trace, and also prefetches like the original client. Figure 12(c) plots the average demand response times, demand bandwidth, and prefetch bandwidth in all three cases. The results show that when the network is loaded, No-Avoidance causes significant interference to demand requests, thereby increasing the average demand response times by a factor of 7. Although NPS doesn't show any improvements, it contains the increase in demand response times to less than 30%, which shows the effectiveness of TCP Nice in limiting network interference. The damage is because TCP Nice is primarily designed for long flows.

5.3.2 *NPS vs. Manual Tuning.* Although Figure 12(a) shows that No-Avoidance with fixed *pfrate* performs well, setting the right value of *pfrate* can be tedious. Figure 13 shows the different thresholds for the PPM algorithm we tested to tune the *pfrate*. On the other hand, NPS self-tunes and performs close to the manually tuned threshold.

5.3.3 *Summary.* In summary, it is feasible to build practical systems using the SSR approach as demonstrated with our case study of the NPS Web Prefetching System. Just like Nice uses delay-based probes to infer foreground load and backs off aggressively, the server monitor in NPS uses increasing response times of representative probe requests as an indicator of high demand load and throttles prefetching aggressively to ensure noninterference. We refer the interested reader to [Kokku et al. 2003] for a discussion of the design space and nontrivial implementation details to make NPS interoperate with legacy browsers and Web servers.

6. DATA DISSEMINATION CASE STUDY

In this section, we investigate the feasibility of the SSR approach for distributed systems with consistency constraints. We present a case study of the TRIP (Transparent Replication through Invalidation and Prefetching) system, a data replication middleware that integrates self-tuning updates and sequential consistency to replicate large-scale information dissemination services. TRIP pursues the aggressive goal of supporting *transparent* service replication whereby a centralized service can be automatically converted to a service distributed across nodes on a wide area network without introducing new semantic or performance bugs. TRIP must therefore provide two key properties.

(1) TRIP must provide *self-tuning updates* to maximize performance and availability given the system resources available at any moment. Self-tuning updates are crucial for transparent replication because static replication policies are more complex to maintain, less able to benefit from spare system resources, and more prone to catastrophic overload if they are mis-tuned or during periods of high system load.

(2) TRIP must provide *sequential consistency* [Lamport 1979] with a tunable maximum staleness parameter to reduce application complexity. Weaker consistency guarantees can introduce subtle bugs [Frigo and Luchangco 1998], and as Internet-scale applications become more widespread, ambitious, and complex, simplifying the programming model becomes increasingly desirable [Hill 1998]. If we can provide sequential consistency, then we can take a single machine's or LAN cluster's service threads that access shared state via a file system or database and distribute these threads across WAN edge servers without re-writing the service and without introducing new bugs.

Not only is each of these properties important, but their combination is vital. Sequential consistency prevents the use of stale data, which could hurt performance and availability, but prefetching replaces stale data with valid data. Conversely, prefetching means that data are no longer fetched when they are used, so a prefetching system must rely on its consistency protocol for correct operation.

Unfortunately, providing sequential consistency in a large scale system while providing good availability [Brewer 2001] and performance [Lipton and Sandberg 1988] is fundamentally difficult. We therefore restrict our attention to replicated *dissemination services*, in which updates occur at one origin server and multiple edge server replicas treat the underlying replicated data as read-only and perform data caching, fragment assembly, per-user customization, and advertising insertion. Although this case is restrictive, it represents an important class of services. For example, Akamai's Edge Side Include [akamai 2001] and IBM's Sport and Event replication system [Challenger et al. 1999] both focus on improving the performance, availability, and scale of dissemination services. Furthermore, the TRIP middleware can be used to build key parts of general applications that distribute content such as file systems, distributed databases, and publish-subscribe systems. For example, our edge TPC-W implementation [Gao et al. 2003] uses different replication strategies for different objects, and TRIP could be used for *dissemination objects* like the TPC-W catalog of items for sale.

Figure 14 provides a high level view of the environment we target. An origin server and several replicas (also called content distribution nodes or edge servers) share data, and logical clients – either on the same machine or another – access the service via the replicas,

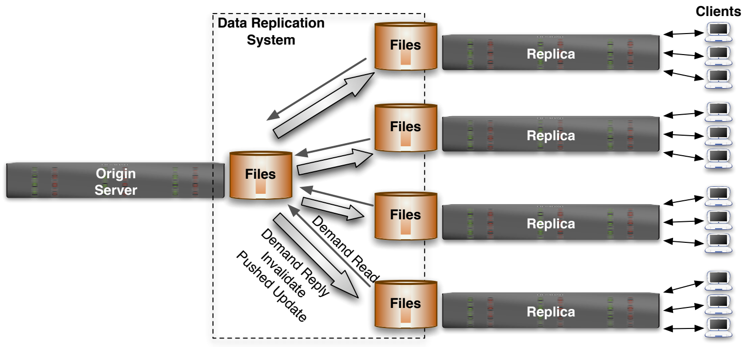


Fig. 14. Edge server architecture.

which run service-specific code to dynamically generate responses to requests [akamai 2001; Awadallah and Rosenblum 2002; Cao et al. 1998; Foster et al. 2002]. The system typically uses some application-specific mechanism [Challenger et al. 1999; Karger et al. 1997; Yoshikawa et al. 1997] to direct client requests to a good (e.g., nearby, lightly loaded, or available) replica. The design of such a redirection infrastructure is outside the scope of the paper; instead, we focus on the data replication middleware that provides shared state across the origin server and replicas. We focus on supporting on the order of 10 to 100 long-lived replicas that each have sufficient local storage to maintain a local copy of the full set of their service's shared data. Although our protocol remains correct under other assumptions about the number of replicas, replica lifetimes, and whether replicas replicate all shared data or only a subset, optimizing performance in other environments may require different tradeoffs.

In the rest of this subsection, we first discuss the consistency requirements for transparent information dissemination. We then describe our design for providing self-tuning updates and sequential consistency. Finally, we evaluate the approach via simulation and prototype measurements.

6.1 Consistency Requirements

Evaluating the semantic guarantees of large-scale replication systems requires careful distinctions between *consistency*, which constrains the order that updates across multiple memory locations become *observable* [Frigo and Luchangco 1998] to nodes in the system, *coherence*, which constrains the order that updates to a single location become observable but does not additionally constrain the ordering of updates across different locations, and *staleness*, which constrains the real-time delay between when an update completes and when it becomes observable. Adve discusses the distinction between consistency and coherence in more detail [Adve and Gharachorloo 1996].

To support transparency, we focus on providing sequential consistency. As defined by Lamport, "The result of any execution is the same as if the [read and write] operations by all processes were executed in some sequential order and the operations of each individual processor appear in this sequence in the order specified by its program." [Lamport 1979] Sequential consistency is attractive for transparent replication because the results of all read and write operations are consistent with an order that could legally occur in a centralized system, so – absent time or other communication channels outside of the shared state – a program that is correct for all executions under a local model with a centralized storage system is also correct for a distributed storage system.

Typically, providing sequential consistency is expensive in terms of latency [Lipton and Sandberg 1988; Burns et al. 2000] or availability [Brewer 2001]. However, we restrict our study to *dissemination services* that have one writer and many readers, and we enforce *FIFO consistency* [Lipton and Sandberg 1988] under which writes by a process appear to all other processes in the order they were issued, but different processes can observe different interleavings between the writes issued by one process and the writes issued by another. Note that for applications that include only a single writer, FIFO consistency is identical to sequential consistency or the weaker causal consistency.

Although sequential consistency provides strong semantic guarantees at replicas, clients of those replicas may observe unexpected behaviors in at least two ways due to communication channels outside of the shared state.

First, because sequential consistency does not specify any real-time requirement, a client may observe stale (but consistent) data. For example, a network partition between the origin server and replica could cause the client of a stock ticker service to observe the anomalous behavior of a stock price not changing for several minutes. We note that in this scenario, physical time acts as a communications channel outside of the control of the data replication middleware that allows a user to observe anomalous behavior from the replication system. Hence, we allow systems to enforce timeliness constraints on data updates by providing Δ-*coherence*, which requires that any read reflect at least all writes that occurred before the current time minus Δ. [Singla et al. 1997] By combining Δ-coherence with sequential consistency, TRIP enforces a tunable staleness limit on the sequentially consistent view. The Δ parameter reflects a per-service trade-off between availability and worst case staleness: reducing Δ improves timeliness guarantees but may hurt availability because disconnected edge servers may need to refuse a request rather than serve overly stale data.

Second, some redirection infrastructures [Challenger et al. 1999; Karger et al. 1997; Yoshikawa et al. 1997] may cause a client to switch between replicas, allowing it to observe inconsistent state. For example, consider two replicas r_1 and r_2 where r_2 processes messages more slowly than r_1, and updates u_1 and u_2 such that u_1 *happens before* [Lamport 1978] u_2. If a client of r_1 sees update u_2, switches to r_2 (which has not seen u_1 yet) and sees data that should have been modified by u_1 but is not, it observes an inconsistency. In [Belaramani et al. 2006], we discuss how to adapt Bayou's session consistency protocol [Terry et al. 1994] to our system to ensure that each client observes a sequentially consistent view regardless of how often the redirection infrastructure switches the client among replicas.

6.2 TRIP Design

Figure 15 provides a high-level view of TRIP's realization of the SSR architecture for synchronizing a replica's data store with the origin server's. When the origin server writes an object (number ①in the figure), it immediately sends an invalidation to each replica ②, updates the local checkpoint ③, and enqueues the body of the update in a priority queue of speculative updates for each replica ④. In contrast with the immediate transmission of invalidations on a normal-priority lossless network connection ⑤, each priority queue drains by sending its highest-priority update to its replica via a low-priority network channel when the network path between the origin server and replica has spare capacity ⑥.

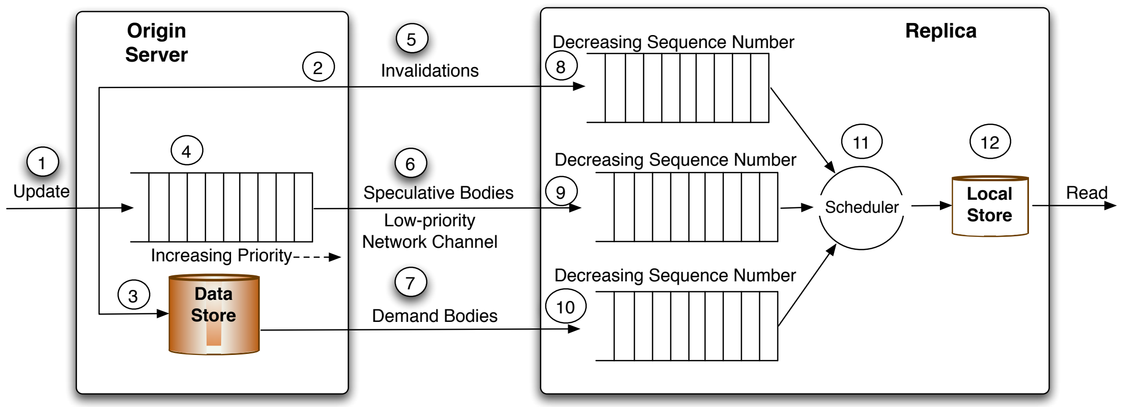


Fig. 15. Transparent Replication through Invalidation and Prefetching (TRIP) architecture.

When the origin server receives a demand read request, it replies with a a demand body from its checkpoint using a normal-priority network channel ⑦.

At the replica, invalidations ⑧, speculative updates ⑨, and demand bodies ⑩ that arrive are buffered rather than being immediately applied to the replica's local data store. A scheduler ⑪ at each replica decides when to apply information from these three buffers to the local store ⑫ in order response time to enforce sequential consistency, enforce Δ-coherence, maximize availability, and minimize response time.

This TRIP design embodies the SSR architecture's approach to safe speculative replication using two key ideas

(1) *Self-tuning resource management.* Sending speculative updates via a priority queue that drains via a low-priority network channel ensures that prefetch traffic does not consume network resources that regular TCP connections could use. In particular, when a lot of "spare bandwidth" is available, the queue drains quickly and nearly all updates are sent as soon as they are inserted. But, when little "spare bandwidth" is available, the buffer sends only high priority updates and absorbs repeated writes to the same object.

(2) *Scheduled application of messages.* Each replica buffers the invalidation, speculative update, and demand update messages it recives and a scheduler decides when to apply these messages to the local store. The scheduler applies messages in an order that meets consistency requirements, it delays applying some messages to improve availability and performance, and it enforces a maximum delay to meet timeliness constraints.

In the next three subsections, we provide additional details on how replicas process updates and how the system copes with machine/network failures. Then Subsection 6.2.3 describes our prototype implementation adn discusses several limitations of the basic algorithm as well as possible optimizations available within this framework.

6.2.1 *Replica Updates.* The core of each replica is a novel *scheduler* that coordinates the application of invalidations, updates, and demand read replies to the replica's local state. The scheduler has two conflicting goals. On one hand, it would like to delay applying invalidations for as long as possible to minimize the amount of invalid data and thereby maximize local hit rate, maximize availability, and minimize response time. On the other hand, it must enforce sequential consistency and Δ-coherence, so it must enforce two constraints:

C1 A replica must apply all invalidations with sequence numbers less than N to its storage before it can apply an invalidation, update, or demand reply with sequence number N.[2]

C2 A replica must apply an invalidation with timestamp t to its storage no later than $t + \Delta - maxSkew$.

Note that Δ specifies the maximum staleness allowed between when an update is applied at the origin server and when the update affects subsequent reads, and *maxSkew* bounds the clock skew between the origin server and the replica.

To meet these goals, when the origin server receives an update, it assigns the update a sequence number and a real-time timestamp, and it sends invalidations, speculative updates, and demand updates as described earlier. Then, a replica enqueues all incoming invalidation and update messages sorted by sequence number, and it applies the following rules to decide when to apply updates and invalidations:

(1) A scheduler applies a demand or speculative update message as soon as it has received and applied all invalidations with lower sequence numbers.
This rule allows a replica to simultaneously apply an invalidation with the corresponding update body. A replica that activates this rule thereby replaces an obsolete version of an object as soon as it can do so without opening a window during which the object would be invalid and could cause a read miss. Note that applying an update normally entails updating the local sequence number and body for the object, but if the locally stored sequence number already exceeds an update's sequence number, the replica must discard the update because a newer demand reply or invalidation has already been processed.

(2) The scheduler applies an invalidation to its local store when (1) an update with the same sequence number is available in the speculative or demand queue of pending updates, (2) the invalidation's deadline arrives at $timestamp + \Delta - maxSkew$, or (3) an update with a higher sequence number is available in the demand queue of pending updates.
The first rule minimizes staleness by aggressively applying an update as soon as both the update and invalidation are available; the second rule enforces the worst-case staleness guarantee; and the third rule minimizes demand response time by immediately applying all invalidations that are preventing a demand read reply from being applied.

6.2.2 *Disconnections and failures.* When a replica becomes disconnected from the server due to a network partition or server failure, the replica attempts to service requests from its local store as long as possible. If the local copies of most objects are valid, a replica may be able to mask the disconnection for an extended period. However, to enforce Δ-coherence, a replica must block all reads if it has not communicated with the origin server for Δ seconds. We use a heartbeat protocol to ensure liveness when the network is available. But, if a read miss occurs during a disconnection, it logically blocks until the connection is reestablished and the server satisfies the demand miss.

In a web service environment, blocking a client indefinitely is an undesirable behavior. Therefore, TRIP provides three ways for services to give up some transparancy in order to gain control of recovery in the case where a replica blocks because it is disconnected from the origin server. First, TRIP can reply to read requests from the calling edge server program by returning an error code. Because this approach requires that the edge server

[2]We show in [Nayate et al. 2003] that enforcing condition C1 yields sequential consistency.

program be designed to expect such an error code, it prevents the replication layer from being fully transparent. Second, TRIP can (1) signal the redirection layer [Challenger et al. 1999; Karger et al. 1997; Yoshikawa et al. 1997] to stop sending requests to this replica and (2) signal the local web server infrastructure to close all existing client connections and to respond to subsequent client requests with HTTP redirects [Fielding et al. 1999] to different replicas. Although this approach requires web servers to be augmented with the ability to handle signals from the replication layer, we do not expect these changes to be invasive. Third, during the common case of a connected replica, the replica enforces a desired $\Delta_{connected}$ staleness bound, but to improve availability and response time, if the replica detects a network disruption, it falls back on enforcing a maximum acceptable $\Delta_{disconnected}$ staleness bound with $\Delta_{connected} < \Delta_{disconnected}$. Increasing Δ during disconnections allows the system to further delay applying pending invalidations and thus maximize the amount of valid local data and maximize the amount of time the replica can operate before suffering a miss.

6.2.3 *Prototype implementation.* Our prototype is implemented in Java, C, and C++ on a Linux platform, but we expect the server code to be readily portable to any standard operating system and the replica code to be portable to any system that supports mounting an NFS server. Our implementation makes use of two subsystems that are outside the scope of this project and that we do not discuss in detail: a protocol for limiting the clock skew between each replica and the origin server [David L. Mills 1992] and a policy for prioritizing which documents to push to which replicas [Gwertzman and Seltzer 1995; Venkataramani et al. 2001].

The rest of this section discusses internal details and design decisions in the server and replica implementations and then details several limitations and opportunities for future enhancements.

6.2.3.1 *Origin Server.* The origin server uses the local file system for file storage, and to simplify handing failures, the origin server uses a custom persistent message queue [Corporation 1995] for sending updates and invalidations to each replica. Because our protocol only uses the update channel to push update data, the origin server does not store out-bound updates to persistent storage and considers it permissible to lose these updates across crashes. To provide a low-priority network channel for updates that does not interfere with other network traffic, we use an implementation of TCP-Nice [Venkataramani et al. 2002b].

6.2.3.2 *Replica.* The replica implements a single *read* method to access shared data. The simplicity of this interface allows us to use TRIP as a building block for a variety of replicated applications that require sophisticated interfaces. For example, publish/subscribe systems can be implemented by having the publisher perform write calls to publish data to the matching service, and the matching service can later make read calls to request data to serve to clients. Chen et al. [Chen et al. 2003] show an approach that can be adopted to compute priorities for pages in a publisher/subscriber model. For our prototype, however, we build TRIP to export a subset of the interface used by the NFS file system via a local user-level NFS file server [Mazires 2001], allowing the replica to mount this local file server as if it were a normal NFS server. Shared objects are accessed as if they are stored in a standard file system. For simplicity, we respond to reads of invalidated data during disconnections by returning an NFS error code to the calling program.

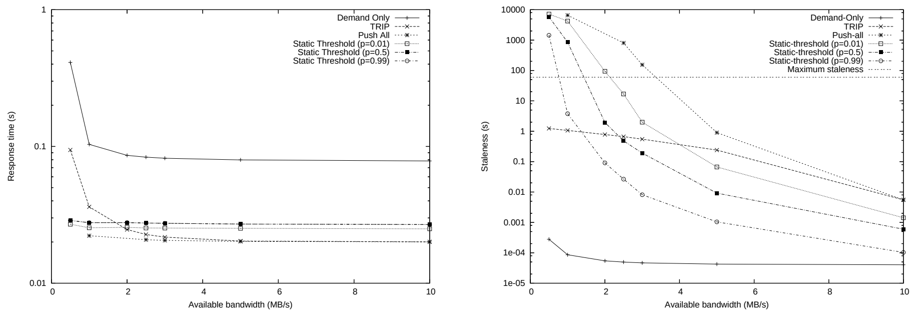
A remaining design choice is how to handle a second read request r_2 for object o_2 that arrives when a first read request r_1 for object o_1 is blocked and waiting to receive a demand reply from the origin server. Allowing r_2 to proceed and potentially access a cached copy of o_2 risks violating sequential consistency [Adve and Gharachorloo 1996] if program order specifies that r_1 *happens before* r_2. On the other hand, if r_1 and r_2 are issued by independent threads of computation that are not synchronized, then the threads are logically concurrent and it would be legal to allow read r_2 to "pass" read r_1 in the cache [Lamport 1979; Frigo and Luchangco 1998]. The TRIP design therefore affords two options. *Conservative* mode preserves transparancy but requires a read issued while an earlier read is blocking on a miss to block. *Aggressive* mode compromises transparency because it requires knowledge of application internals, but it allows a cached read to pass a pending read miss. Our prototype implements the *Conservative* approach to maximize transparency, so it may give up some performance.

6.2.3.3 *Limitations.* Our current protocol faces two limitations that could be addressed with future optimizations. First, as described in Section 6.1 our current protocol can allow a client that switches between replicas to observe violations of sequential consistency. We speculate in [Nayate et al. 2003] that a system could shield a client from such inconsistency by adapting Bayou's session guarantees protocol [Terry et al. 1994]. Second, our protocol sends each invalidation to all replicas even if a replica does not currently have a valid copy of the object being invalidated. We take this approach for simplicity, although our protocols could be extended to more traditional caching environments where replicas maintain small subsets of data by adding callback state [Howard et al. 1988].

6.3 TRIP Evaluation

We evaluate TRIP using two approaches: by employing a trace-driven simulator and evaluating a prototype. Our primary results are that (1) self-tuning prefetching can dramatically improve the response time of serving requests at replicas compared to demand-based strategies, (2) although a Push All strategy enjoys excellent response times by serving all requests directly from replicas' local storage, this strategy is fragile in that if update rates exceed available bandwidth for an extended period of time, the service must either violate its Δ-consistency guarantee or become unavailable, (3) when prefetching is used, delaying application of invalidation messages by up to 60 seconds provides a modest additional improvement in response times, and (4) by maximizing the amount of valid data at replicas, prefetching can improve availability by masking disconnections between a replica and the origin server.

6.3.1 *Simulation Methodology.* Our trace-driven simulator models an origin server and twenty replicas and assumes that the primary bottleneck in the system is the network bandwidth from the origin server. To simplify analysis and comparisons among algorithms, we assume that the bandwidth available to the system does not change throughout a simulation. We also assume that bandwidth consumed by control information (invalidate messages, message queue acknowledgments, meta data, etc.) is insignificant compared to the bandwidth consumed transferring objects; we confirm using our prototype that control messages account for less than 1% of the data transferred by the system. Transferring an object over the network thus consumes a link for $objectsize/bandwidth$ seconds, and the delay from when a message is sent to when it is received is given by $nwLatency + messageSize/bandwidth$. We simulate a round-trip time ($2 * nwLatency$)



(a) Average client-perceived response times (b) Average staleness of data served by replicas

Fig. 16. Average response times and data staleness observed at a client

of 200ms +/- 90% between the origin server and a replica.

We compare TRIP's *FIFO-Delayed-Invalidation/Priority-Delayed-Update* algorithm with three algorithms: *Demand Only*, which delivers invalidates eagerly in FIFO order but does no prefetching, *Push All* which eagerly pushes all updates to all replicas in FIFO order, and *Static Threshold* which pushes, in FIFO order, each update with a predicted probability of reference above a specified threshold. We assume that the system requires (1) sequential consistency, which all of these algorithms provide and (2) a Δ-coherence guarantee of Δ = 60 seconds, which *Demand Only* naturally meets, which *TRIP* systematically enforces, and which *Push All* or *Static Threshold* may or may not meet depending on available bandwidth.

We evaluate our algorithms using a trace-based workload of the Web site of a major sporting event [Challenger et al. 1999] hosted at several geographically distributed locations. In order to simplify simulations we ignore those entries in our trace files that contain dynamic/malformed requests, result in invalid server return codes, or that appear out of order.

Prediction policy. Our interface allows a server to use any algorithm to choose the priority of an update, and this article does not attempt to extend the state of the art in prefetch prediction. A number of standard prefetching prediction algorithms exist [Duchamp 1999; Griffioen and Appleton 1993] or the server may make use of application-specific knowledge to prioritize an item. Our simple default heuristic for estimating the benefit/cost ratio of one update compared to another is to first approximate the probability that the new version of an object will be read before it is written as the observed read frequency of the object divided by the observed write frequency of the object and then to set the relative priority of the object to be this probability divided by the object's size [Venkataramani et al. 2001]. This algorithm appears to be a reasonable heuristic for server push-update protocols: it favors read-often objects over write-often objects and it favors small objects over large ones.

6.3.2 *Simulation Results.*

6.3.2.1 *Response Times and Staleness.* In Figure 16(a), we quantify the effects of different replication strategies on client-perceived response times as we vary available bandwidth. We assume that client requests for valid objects at the replica are satisfied in 20ms, whereas requests for invalidated objects are forwarded from the replica to the origin over a network with an average round-trip latency of 200ms as noted above. To put these results in perspective, Figure 16(b) plots the average staleness observed by a request. We define staleness as follows. If a replica serves version k of an object after the origin site has already (in simulated time) written version j ($j > k$), the staleness of a request is the difference between when the request arrived at the replica and when version $k + 1$ was written. To facilitate comparison across algorithms, this average staleness figure includes non-stale requests in the calculations. We omit due to space constraints a second graph that shows the (higher) average staleness observed by the subset of reads under each algorithm that receives stale data.

We also show in Figures 16(a) and 16(b) the latency and staleness yielded when using the static-threshold-prefetching algorithm, which prefetches objects when the predicted likelihood of their being accessed exceeds a statically chosen threshold. We plot the behavior of this algorithm when it is tuned to prefetch objects that have a greater than 1%, 50% and 99% estimated chance of being accessed (denoted *Static Threshold* (p = 0.01), (p = 0.5), and (p = 0.99), respectively, on the graph). We note that *Push All* and *Demand Only* represent extreme cases of this algorithm with thresholds of 0 (push an update regardless of its likelihood of being accessed) and 1 (only push an update if it is certain to be accessed), respectively.

The data indicate that the simple *Push All* algorithm provides much better response times than the *Demand Only* strategy, speeding up responses by a factor of at least four for all bandwidth budgets examined. However, this comparison is a bit misleading as Figure 16(b) indicates: for bandwidth budgets below 2.1MB/s, *Push All* fails to deliver all of the updates and serves data that become increasingly stale as the simulation progresses. We note that under such a bound requirement, Push All replicas would be forced to either violate this freshness guarantee or become unavailable when the available bandwidth falls below about 3MB/s.

The *Static Threshold* lines illustrate precisely the problem with threshold-based speculative replication. When the system has less than 2MB/s available bandwidth, the *Static Threshold* algorithm yields lower response times than the *TRIP* algorithm, but it does so by violating the staleness guarantees. Conversely, when the system has more than 2MB/s bandwidth available, TRIP is able to replicate more aggressively and provides better performance than the threshold-based approach, as the *static Threshold* algorithm fails to utilize it to reduce response times.

Even at low bandwidths, *TRIP* gets significantly better response times than the *Demand Only* algorithm because (a) the self-tuning network scheduler allows prefetching to occur during lulls in demand traffic even for a heavily loaded system [3] and (b) the priority queue at the origin server ensures that the prefetching that occurs is of high benefit/cost items. TRIP's ability to exploit lulls in demand bandwidth also constitutes the reason that when the system has 2MB/s available bandwidth TRIP can outperform static-threshold while still retaining its timeliness guarantees.

6.3.2.2 *Variations of TRIP.* Due to space constraints, we omit a graph that plots response times for two variations of TRIP. In the first variation, we reduce the Δ parameter to 0 to evaluate the behavior of TRIP when we require replicas to apply all invalidate messages immediately. Under this scenario we find that values of Δ below 60s inflict a modest cost on response times, but this cost falls as available bandwidth increases. For example, at 1MB/s of available bandwidth, the Δ = 60s case yields 12.6% lower response times

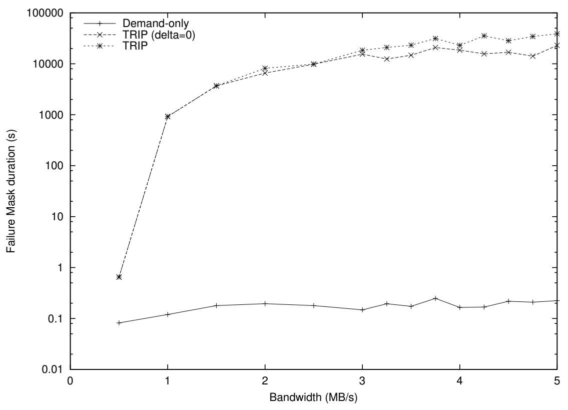


Fig. 17. Mask duration as bandwidth varies.

than the Δ = 0s case. However, our second variation of TRIP, TRIP-aggressive, which sacrifices some transparency and assumes that parallel read requests are independent, can result in substantial benefits. For example, for a system with 500KB/s of available bandwidth, this optimization improves response time by a factor of 2.5. But, this benefit falls as available bandwidth increases, suggesting that this optimization may become less valuable as network costs fall relative to the cost of requiring programmers to carefully analyze applications to rule out the possibility of unexpected interactions [Hill 1998].

6.3.2.3 *Availability.* We measure the replication policies' effect on availability as follows. For each of 50 runs of our simulator for a given set of parameters, we randomly choose a point in time when we assume that the origin server becomes unreachable to replicas. We simulate a failure at that moment and measure the length of time before any replica receives a request that it cannot mask due to disconnection. We refer to this duration as the mask duration. We assume that systems enforce Δ-coherence with Δ = 60 seconds before the disconnection but that disconnected replicas maximize their mask duration by stopping their processing of invalidations and updates during disconnections and extending Δ as long as they can continue to service requests. Thus, during periods of disconnectivity, our system chooses to provide stale data rather than failing to satisfy client requests. Note that given these data, the impact of enforcing shorter Δs during disconnections can be estimated as the minimum of the time reported here and the Δ limit enforced.

Figure 17 shows how the average mask duration varies with bandwidth for the TRIP, TRIP (Δ = 0), and Demand Only algorithms. Because mask duration is highly sensitive to the timing of a failure, dierent trials show high variability. We quantify this variability in more detail in an extended technical report [Nayate et al. 2003].

Note that the traditional Demand Only algorithm performs poorly. In Figure 17, the line closely follow y = 0, indicating virtually no ability to mask failures. This poor behavior arises because the first request for an object after that object is modied causes a disconnected replica to experience an unmaskable failure. On the other hand, the Push All algorithm can mask all failures due to the fact that at any point in time, the entries in a replica's cache form a sequentially consistent (though potentially stale) view of data.

The TRIP algorithm outperforms the Demand Only algorithm in the graph by maxi-

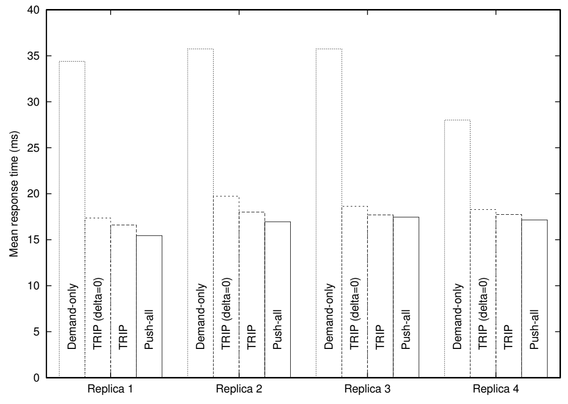


Fig. 18. Replica-perceived response times yielded by the Demand Only, FIFO, and TRIP algorithms.

mizing the amount of local valid data.We note that both TRIP variations provide average masking times of thousands of seconds for bandwidth of 1.5MB/s and above and that providing additional bandwidth allows these systems to prefetch more data and hence mask a failure for a longer duration. As noted in Section 6.2.2, systems may choose to relax their Δ-coherence time bound to some longer $\Delta_{disconnected}$ value during periods of disconnection to improve availability. These data suggest that systems may often be able to completely mask failures that last the maximum maskable duration even for relatively large $\Delta_{disconnected}$ limits.

6.3.2.4 *Prototype Measurements.* We evaluate our prototype on the Emulab testbed [White et al. 2002]. We configure the network to consist of an origin server and 4 replicas that receive 5Mb/s of bandwidth and 200ms round-trip times. We mount the local user-level file server using NFS with attribute caching disabled. For simplicity, we do not monitor object replication priorities in real time but instead pre-calculate them using each object's average read rate, write rate, and size [Venkataramani et al. 2001].

Since the goal of the prototype is to evaluate how our system performs in practice, we use a more realistic evaluation methodology from the one we use for our simulator. In particular, when evaluating our prototype we do not remove any entries from our traces and make no simplifying assumptions about the size of invalidate messages or the behavior of network links. However, due to the lack of data on which resources or objects get accessed to handle dynamic requests, our system simply treats dynamic requests as accesses to static pre-generated objects.

Figure 18 shows the response times as seen at each of the 4 replicas.We collect these data by replaying at the origin and at each replica the first hour of our update trace and web traces in real time. The response time for a given request is calculated as the difference between when the request arrives at a replica and when its reply is generated. Note that these response times do not represent the end-to-end delay experienced by clients because they do not include the network delays between clients and replicas. However, one can easily compute total end-to-end delays by adding estimated client-replica network delays to this data. As we see in the graph, the Push All algorithm yields the best response time. For example, it outperforms the Demand Only algorithm by a factor of 2 for 3 of the 4 replicas. We note that at 5Mb/s bandwidth available to the system, TRIP incurs only minor increases in response times over the more dangerous Push All: 7.5%, 6.2%, 1.4%, and 3.4% increase for each replica respectively. We also note that by delaying the application of invalidate messages, TRIP with Δ = 60s reduces response times compared to Δ = 0 by 4.4%, 8.7%, 5.0%, and 3.0% respectively.

6.3.2.5 *Summary.* In summary, we conclude that the SSR approach is practical for large-scale distributed systems with consistency constraints. The TRIP data dissemination system case study shows that, despite the CAP dilemma, SR can provide significant improvements in availability and response time while satisfying strong consistency constraints. Although we report on experience only with sequential and Δ consistency in a single-writer environment in this article, we believe that the SSR approach can be used to build more complex distributed systems with general consistency constraints [Belaramani et al. 2006].

7. RELATED WORK

In this paper, we presented a general architecture for speculative replication with two novel contributions: (i) self-tuning resource management and (ii) integration of SR with consistency constraints to alleviate the CAP dilemma. In this work, we draw on three of our previous studies that define TCP-Nice [Venkataramani et al. 2002b] and implement systems for speculative replication of web [Kokku et al. 2003] and dissemination [Nayate et al. 2004] data. This article builds on these ideas to synthesize a new, general architecture for speculative replication. Our case studies here are limited to client-server systems in wide use today such as the Web and data dissemination applications. Subsequent to the work described here, we have taken SSR's principles and applied them to serverless distributed systems with multiple writers and general consistency semantics. This approach, known as PRACTI replication [Belaramani et al. 2006], implements a log-exchange mechanism on top of SSR to propagate invalidates and additionally uses summarization techniques to simulataneously provide *partial replication, arbitrary consistency, and topology independence*.

A number of studies have pointed out the benefits of speculative replication for large-scale distributed systems such as the Web, wide-area file systems, distributed databases, edge service architectures etc.

Proposed models for Web prefetching include Web server-assisted pushing of popular content [Padmanabhan and Mogul 1996a; Gwertzman and Seltzer 1995; Duchamp 1999], Web proxy-assisted prefetching [Fan et al. 1999; Bestavros 1996b], client-side only prefetching [Wcol ; Klemm 1999; Fireclick]. The general consensus appears to favor a high level architecture in which a Web server or an edge server sends hints to a client or proxy for what to prefetch and the latter issues requests for the same [Padmanabhan and Mogul 1996a; Chen and Zhang ; Duchamp 1999; Mozilla]. Proposed schemes include prefetching hyperlinks [Duchamp 1999; Wcol], prefetching the "top N" most popular objects [Markatos and Chronaki 1998], history-based Markov models [Padmanabhan and Mogul 1996a; Palpanas 1998], utility-based decision-theoretic models [Horvitz 1998], data mining based approaches [Pitkow and Pirolli 1999; Chen et al. 1998; Su et al. 2000] etc. A survey of various prediction algorithms may be found in [Davison 2002].

Unfortunately, none of these schemes address the problem of resource management amidst interference in a systematic manner. Several schemes attempt to balance cost and benefit of prefetching using threshold-based approaches that are validated using simplistic trace-based simulation experiments. For example, Duchamp [Duchamp 1999] proposes a threshold of 0.25 on the access probability to limit prefetching bandwidth; Klemm [Klemm 1999] proposes limiting prefetching to objects whose expected round-trip time is at most 75% of the average of all accesses; Padmanabhan and Mogul suggest tuning the threshold on access probability to limit bandwidth; Jiang and Kleinrock [Jiang and Kleinrock 1997] propose an adapting a threshold to limit bandwidth cost by modeling the server and network as an M/G/1 round-robin processor sharing system; Wcol [Wcol] limits bandwidth by placing a bound on the number of hyperlinks prefetched; Firefox [Mozilla] is provisioned to prefetch during the client's idle periods but ignores interference with other applications and users. Some schemes propose coarse-grained scheduling schemes to reduce interference such as prefetching during hours where there is little demand traffic [Dykes and Robbins 2001; Maltzahn et al. 1999].

Davison et. al [Davison and Liberatore 2000] propose using a connectionless transport protocol and using low priority datagrams (infrastructural support for which is assumed) to reduce network interference due to Web prefetching. Modified servers speculatively push documents chunked into datagrams of equal size and (modified) clients use range requests as defined in HTTP/1.1 for missing portions of the document. Servers maintain state information for prefetching clients and use coarse-grained estimates of per-client bandwidth to limit the rate at which data is pushed to the client. Their simulation experiments do not explicitly quantify interference and use lightly loaded servers in which only a small fraction of clients are prefetching.

All of the above schemes focus on limiting the bandwidth consumed by prefetching. Though bandwidth is an expensive resource in a WAN system, it does not represent the true overall cost of prefetching. SSR adopts a wholistic view of resource management for speculative replication and attempts to eliminate interference at every resource. The NPS Web prefetching system demonstrates that it is possible to do so in a deployable manner without modifying existing infrastructure.

Speculative update propagation has been proposed for improving performance and availability of wide-area file systems. Examples include hoarding in Coda [Kistler and Satyanarayanan 1992] during periods of connectivity, anti-entropy in Bayou [Terry et al. 1995], aggressive replica creation and update propagation in Pangaea [Saito et al. 2002], lazy replication [Ladin et al. 1992], AvantGo [AvantGo 2001] etc. Edge service architectures [akamai ; Gao et al. 2003; Sivasibramanian et al. 2005] perform speculative replication in a loose sense where content is moved close to a client before access. However, these systems operate in resource-rich environments where it is feasible to replicate and update the entire content base or an earmarked subset thereof at each location. Often, as in the case of CDNs, the resources used for such replication are provisioned separately from those used by demand load. Thus, the above systems do not face the problem of managing intereference across different wide-area administrative domains. One reason for lack of deployment of large-scale speculative replication systems today is that they must wait until system capacity becomes high enough to employ simplistic *push-all* or *prefetch-all* strategies.

7.1 Background Transport

Our goal of preventing network interference between speculative and demand load in a deployable manner led us to develop TCP Nice that provides low priority transport in a network dominated by TCP flows.

TCP congestion control has seen an enormous body of work since Jacobson's seminal paper on the topic [Jacobson 1988]. 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 TCP Nice's 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. Nice 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 [Yang and Lam 2000] and binomial [Bansal and Balakrishnan 2001] 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. Router prioritization queues such as those proposed for DiffServ [RFC 2475 1999] 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 our work on TCP Nice 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.

Rate limiting techniques can provide a coarse form of prioritization. Spring et. al [Spring et al. 2000] discuss prioritizing flows by controlling the receive window sizes of clients. Crovella et. al [Crovella and Barford 1998] 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.

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 [Andersen et al. 2000] 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) [Microsoft] 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.

TCP-LP [Kuzmanovic and Knightly 2003], proposed roughly simultaneously with TCP Nice, provides low-priority transfers using a similar round-trip time based algorithm. Key et al [Key et al. 2004] subsequently developed an application-level receiver-side scheme for low priority transport that uses receiver window sizes to control the sending rate. They use a fluid model and an optimization-based framework to correlate the change in the goodput of a background flow to interference with foreground flows. In comparison to these schemes, our work on TCP Nice, in addition to serving as a stand alone background transfer protocol, presents it in the broader context of building large-scale replication systems.

7.2 Consistency in SR systems

Most proposed Internet-scale data replication systems focus on ensuring various levels of coherence or staleness or both [Cohen et al. 1998; Sivasibramanian et al. 2005; Krishnamurthy and Wills 1998; Li and Chariton 1999; Mikhailov and Wills 2003; Worrell 1994; Yin et al. 1999; Yin et al. 2002a], but few provide explicit consistency guarantees. Unfortunately, Frigo notes that even strong coherence is considerably weaker than sequential consistency [Frigo 1998]. Bradley and Bestavros [Bradley and Bestavros 2003] argue that increasingly complex Internet-scale services will demand sequential consistency and propose a vector-clock-based algorithm for achieving it. In the light of the scarcity of Internet-scale data replication systems providing strong consistency guarantees, it is not surprising that speculative replication for such systems has been a largely unexplored domain

The IBM Sporting and Event CDN system uses a push-all replication strategy and enforces delta coherence via invalidations [Challenger et al. 1999]. Akamai's EdgeSuite [akamai 2001] primarily relies on demand reads and enforces delta coherence via polling with stronger consistency available via object renaming. Burns et al. [Burns et al. 2000] discuss a *publish consistency* model of consistency that is useful for web workloads and show that consistency implemented by file systems has inefficiencies that prevents easily scaling them to many clients. Most of these systems use demand reads, but several strategies for mixing updates and invalidates have been explored for multicast networks [Fei 2001; Rodriguez and Sibal 2000; Li and Chariton 1999]. These multicast-based proposals all use static thresholds to control prefetching and provide best-effort consistency, coherence, and timeliness semantics by sending and applying all messages eagerly. In contrast, TRIP, based on the SSR architecture, provides self-tuning support for prefetching and maintains sequential consistency. A potential avenue for future work is to develop a way for TRIP to make use of multicast or hierarchies to scale to larger numbers of replicas. SSR's separation of data and metadata paths should make such extensions straightforward.

Our choice of sequential consistency for the TRIP case study is similar in spirit to Hill's position that multiprocessors should support simple memory consistency models like sequential consistency rather than weaker models [Hill 1998]. Hill argues that speculative execution reduces the performance benefit that weaker models provide to the point that their additional complexity is not worth it. We similarly argue that for dissemination workloads, as technology trends reduce the cost of bandwidth, prefetching can reduce the cost of sequential consistency so that little additional benefit is gained by using a weaker model and exposing more complexity to the programmer. In the context of distributed systems, Gray [J. Gray and I.L. Traiger and C.A. Galtaire and B.G. Lindsay 1982] and Birman [K. Birman 2005] have similarly argued for saving (expensive) human effort and attention by developers and users at the possible cost of (inexpensive) additional processing and protocol messages.

The problem of isolating speculative and demand load has been previously considered in the context of stand-alone systems. For hardware prefetching, Lin et. al [Lin et al. 2001] propose issuing prefetch requests only when bus channels are idle and giving them low replacement priorities so as to not degrade the performance of regular memory accesses and avoid cache pollution. Several algorithms for balancing prefetch and demand use of memory and storage system have been proposed [Cao et al. 1995b; Chandra et al. 2001;

Kimbrel et al. 1996; Patterson et al. 1995b]. Applying any of these schemes in the context of large-scale SR systems would require significant infrastructural changes. The SSR approach, supported by our case studies of NPS and TRIP, shows how to isolate speculative and demand load in simple and deployable manner for WAN systems.

8. DISCUSSION AND CONCLUSIONS

In this article, we argue that in order to be useful in practice, speculative replication should have three key features: it should be self-tuning, it should integrate consistency, and it should be deployable. We propose the Safe Speculative Replication (SSR) architecture to meet these goals.

The biggest surprise in conducting this work was the importance and feasibility of an end-to-end approach to resource management. For example, we initially planned to leverage the vast literature of processor and disk schedulers to avoid server interference in our NPS system. But, when the time came to implement the server, we had trouble getting our hands on the code for the disk scheduler we had planned to use, which forced us to reconsider our approach and use the simpler end-to-end monitor approach. Although low level schedulers may sometimes be appropriate when instantiating the SSR architecture, we now appreciate that the deployability advantages of the end-to-end approach will frequently outweigh the performance advantages of low-level schedulers.

One key piece of future work is generalizing the SSR resource schedulers to account for costs like energy or per-byte bandwidth charges that impose a charge on prefetching even when preetching does not interfere with demand requests. We speculate that SSR schedulers that consider such costs may choose to drop prefetch requests whose expected value falls below some threshold, but evaluating this idea remains future work.

REFERENCES

ACHARYA, A. AND SALTZ, J. 1996. A study of internet round-trip delay. Tech. Rep. CS-TR-3736, University of Maryland.

ADVE, S. AND GHARACHORLOO, K. 1996. Shared memory consistency models: A tutorial. *IEEE Computer*, 66–76.

AGGARWAL, A., SAVAGE, S., AND ANDERSON, T. 2000. Understanding the performance of TCP pacing. In *Infocom (3)*.

akamai. Akamai, Inc. Home Page. http://www.akamai.com.

akamai 2001. Turbo-charging dynamic web data with akamai edgesuite. Akamai White Paper.

ANDERSEN, D., BANSAL, D., CURTIS, D., SESHAN, S., AND BALAKRISHNAN, H. 2000. System support for bandwidth management and content adaptation in internet applications.

ANDERSON, T., PETERSON, L., SHENKER, S., AND TURNER, J. 2005. Overcoming the Internet impasse through virtualization. *Computer 38*, 4, 34–41.

APACHE HTTP SERVER PROJECT. http://httpd.apache.org.

AvantGo 2001. http://www.avantgo.com.

AWADALLAH, A. AND ROSENBLUM, M. 2002. The vMatrix: A network of virtual machine monitors for dynamic content distribution. In *"Proceedings of the Web Caching and Content Distribution Workshop*.

BANGA, G., DRUSCHEL, P., AND MOGUL, J. 1999. Resource Containers: A New Facility for Resource Management in Server Systems. In *OSDI99*.

BANSAL, D. AND BALAKRISHNAN, H. 2001. Binomial Congestion Control Algorithms. In *INFOCOM 2001*. 631–640.

BELARAMANI, N., DAHLIN, M., GAO, L., NAYATE, A., VENKATARAMANI, A., YALAGANDULA, P., AND ZHENG, J. 2006. PRACTI replication. In *Proceedings of the Third USENIX Symposium on Networked Systems Design and Implementation*.

BESTAVROS, A. 1996a. Speculative Data Dissemination and Service to Reduce Server Load, Network Traffic, and Service Time in Distributed Information Systems. In *International Conference on Data Engineering*.

BESTAVROS, A. 1996b. Speculative data dissemination and service to reduce server load, network traffic and service time in distributed information systems. In *ICDE '96: Proceedings of the Twelfth International Conference on Data Engineering*. IEEE Computer Society, Washington, DC, USA, 180–187.

BONALD, T. 1998. Comparision of tcp reno and tcp vegas via fluid approximation. In *INRIA, Research Report*.

BRADLEY, A. AND BESTAVROS, A. 2003. Basis token consistency: Supporting strong web cache consistency. In *GLOBECOMM 2003*.

BRAKMO, L., O'MALLEY, S., AND PETERSON, L. 1994. Tcp vegas: New techniques for congestion detection and avoidance. In *Proc SIGCOMM*. 24–35.

BREWER, E. 2001. Lessons from giant-scale services. *IEEE Internet Computing*.

BURNS, R., REES, R., AND LONG, D. 2000. Consistency and locking for distributing updates to web servers using a file system. In *Workshop on Performance and Architecture of Web Servers*.

CAO, P., FELTEN, E., KARLIN, A., AND LI, K. 1995a. Implementation and Performance of Integrated Application-Controlled Caching, Prefetching, and Disk Scheduling. In *Proceedings of the SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. 188–197.

CAO, P., FELTEN, E. W., KARLIN, A. R., AND LI, K. 1995b. A study of integrated prefetching and caching strategies. In *SIGMETRICS*.

CAO, P., ZHANG, J., AND BEACH, K. 1998. Active Cache: Caching Dynamic Contents on the Web. In *Proceedings of Middleware 98*.

CHALLENGER, J., IYENGAR, A., AND DANTZIG, P. 1999. A scalable system for consistently caching dynamic web data. In *IEEE INFOCOM*.

CHANDRA, B. 2001. Web workloads influencing disconnected services access. M.S. thesis, University of Texas at Austin.

CHANDRA, B., DAHLIN, M., GAO, L., KHOJA, A., RAZZAQ, A., AND SEWANI, A. 2001. Resource management for scalable disconnected access to web services. In *WWW10*.

CHANDRA, B., DAHLIN, M., GAO, L., AND NAYATE, A. 2001. End-to-end WAN Service Availability. In *Proceedings of the Third USENIX Symposium on Internet Technologies and Systems*.

CHEN, M.-S., PARK, J. S., AND YU, P. S. 1998. Efficient data mining for path traversal patterns. *IEEE Transactions on Knowledge and Data Engineering 10,* 2, 209–221.

CHEN, X., CHEN, Y., AND RAO, F. 2003. An efficient spatial publish/subscribe system for intelligent location-based services. In *DEBS '03: Proceedings of the 2nd international workshop on Distributed event-based systems*. ACM Press, New York, NY, USA, 1–6.

CHEN, X. AND ZHANG, X. Coordinated data prefetching by utilizing reference information at both proxy and web servers.

CHEN, X. AND ZHANG, X. 2001. Coordinated Data Prefetching by Utilizing Reference Information at Both Proxy and Web Servers.

CHIU AND JAIN. 1989. "analysis of increase and decrease algorithms for congestion avoidance in computer networks". *Journal of Computer networks and ISDN 17,* 1 (June), 1–14.

CHO, J. AND GARCIA-MOLINA, H. 2000. Synchronizing a database to improve freshness. In *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. ACM Press, New York, NY, USA, 117–128.

COHEN, E., KRISHNAMURTHY, B., AND REXFORD, J. 1998. Improving End-to-End Performance of the Web Using Server Volumes and Proxy Filters. In *Proceedings of the ACM SIGCOMM '98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*.

CORPORATION, I. 1995. Mqseries: An introduction to messaging and queueing. Tech. Rep. GC33-0805-01, IBM Corporation. July. `ftp://ftp.software.ibm.com/software/mqseries/pdf/horaa101.pdf`.

CROVELLA, M. AND BARFORD, P. 1998. The network effects of prefetching. In *Proceedings of IEEE Infocom*.

CROVELLA, M. AND BESTAVROS, A. 1996. Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes. In *SIGMETRICS*.

CUREWITZ, M., KRISHNAN, P., AND VITTER, J. 1993. Practical prefetching via data compression. In *SIGMOD Intl. Conference on Management of Data*.

DAHLIN, M., CHANDRA, B., GAO, L., AND NAYATE, A. 2003. End-to-end WAN service availability. *ACM/IEEE Transactions on Networking 11,* 2 (Apr.).

DAVID L. MILLS. 1992. Network time protocol (version 3) specification, implementation, and analysis. Tech. Rep. RFC-1305, IETF.

DAVISON, B. D. 2002. The design and evaluation of web prefetching and caching techniques. Ph.D. thesis, Department of Computer Science, Rutgers University.

DAVISON, B. D. AND LIBERATORE, V. 2000. Pushing politely: Improving Web responsiveness one packet at a time (extended abstract). *Performance Evaluation Review 28,* 2 (Sept.), 43–49.

DUCHAMP, D. 1999. Prefetching Hyperlinks. In *Proceedings of the Second USENIX Symposium on Internet Technologies and Systems*.

DYKES, S. AND ROBBINS, K. A. 2001. ”a viability analysis of cooperative proxy caching”. In *INFOCOM 2001*.

FAN, L., CAO, P., LIN, W., AND JACOBSON, Q. 1999. Web prefetching between low-bandwidth clients and proxies: Potential and performance.

FEI, Z. 2001. A novel approach to managing consistency in content distribution networks. In *Internat. Workshop on Web Caching and Content Distribution*.

FIELDING, R., GETTYS, J., MOGUL, J., FRYSTYK, H., MISINTER, L., LEACH, P., AND BERNERS-LEE, T. 1999. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, Internet Engineering Task Force. June.

FIRECLICK. Fireclick blueflame.
http://www.fireclick.com.

FOSTER, I., KESSELMAN, C., NICK, J., AND TUECKE, S. 2002. The physiology of the grid: An open grid services architecture for distributed systems integration. In *Global Grid Forum*.

FRIGO, M. 1998. The weakest reasonable memory. M.S. thesis, MIT.

FRIGO, M. AND LUCHANGCO, V. 1998. Computation-Centric Memory Models. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*.

GAO, L., DAHLIN, M., NAYATE, A., ZHENG, J., AND IYENGAR, A. 2003. Application specific data replication for edge services. In *International World Wide Web Conference*.

GENI. Global Environment for Network Innovations (GENI). http://www.geni.net.

GILBERT, S. AND LYNCH, N. 2002. Brewer’s conjecture and the feasibility of Consistent, Available, Partition-tolerant web services. In *ACM SIGACT News, 33(2)*.

GRAY, J. 2003. Distributed computing economics. Tech. Rep. MSR-TR-2003-24, Microsoft Research.

GRAY, J. AND SHENOY, P. 2000. Rules of Thumb in Data Engineering. In *”Proc. 16th Internat. Conference on Data Engineering”*. 3–12.

GRIBBLE, S. D., BREWER, E. A., HELLERSTEIN, J. M., AND CULLER, D. 2002. Scalable distributed data structures for internet service construction. In *OSDI*.

GRIFFIOEN, J. AND APPLETON, R. 1993. Automatic Prefetching in a WAN. In *IEEE Workshop on Advances in Parallel and Distributed Systems*.

GRIFFIOEN, J. AND APPLETON, R. 1994a. Reducing File System Latency Using A Predictive Approach. In *Proceedings of the Summer 1994 USENIX Conference*.

GRIFFIOEN, J. AND APPLETON, R. 1994b. Reducing file system latency using a predictive approach. In *USENIX Summer*. 197–207.

GWERTZMAN, J. AND SELTZER, M. 1995. The case for geographical pushcaching. In *HOTOS95*. 51–55.

HILL, M. 1998. Multiprocessors should support simple memory consistency models,. In *IEEE Computer*.

HORVITZ, E. 1998. Continual computation policies for utility-directed prefetching. In *CIKM ’98: Proceedings of the seventh international conference on Information and knowledge management*. ACM Press, New York, NY, USA, 175–184.

HOWARD, J., KAZAR, M., MENEES, S., NICHOLS, D., SATYANARAYANAN, M., SIDEBOTHAM, R., AND WEST, M. 1988. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems 6,* 1 (Feb.), 51–81.

IBM. Websphere. http://www.ibm.com/websphere.

INTEL. N-tier architecture improves scalability and ease of integration. http://www.intel.com/eBusiness/pdf /busstrat/industry/wp012302.pdf.

J. Gray and I.L. Traiger and C.A. Galtaire and B.G. Lindsay. 1982. Transactions and Consistency in Distributed Database Systems. *ACM TODS 7.3*, 323–342.

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

Jiang, Z. and Kleinrock, L. 1997. Prefetching links on the WWW. In *ICC (1)*. 483–489.

K. Birman. 2005. *Reliable Distributed Systems, Technologies, Web Services, and Applications*. Springer Verlag.

Karger, D., Lehman, E., Leighton, T., Levine, M., Lewin, D., and Panigrahy, R. 1997. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of the Twenty-ninth ACM Symposium on Theory of Computing*.

Key, P., Massouli&#233;, L., and Wang, B. 2004. Emulating low-priority transport at the application layer: a background transfer service. In *SIGMETRICS 2004/PERFORMANCE 2004: Proceedings of the joint international conference on Measurement and modeling of computer systems*. ACM Press, New York, NY, USA, 118–129.

Kimbrel, T., Tomkins, A., Patterson, R. H., Bershad, B., Cao, P., Felten, E., Gibson, G., Karlin, A. R., and Li, K. 1996. A trace-driven comparison of algorithms for parallel prefetching and caching. In *OSDI*. 19–34.

Kistler, J. and Satyanarayanan, M. 1992. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems 10,* 1 (Feb.), 3–25.

Klemm, R. P. 1999. Webcompanion: A friendly client-side web prefetching agent. In *IEEE Transactions on Knowledge and Data Engineering*. Vol. 11(4):577–594.

Kokku, R., Yalagandula, P., Venkataramani, A., and Dahlin, M. 2003. Nps: A non-interfering deployable web prefetching system. In *Proceedings of the Fourth USENIX Symposium on Internet Technologies and Systems*.

Kostic, D., Rodriguez, A., Albrecht, J., and Vahdat, A. 2003. Bullet: High bandwidth data dissemination using an overlay mesh. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*.

Kotz, D. and Ellis, C. 1991. Practical Prefetching Techniques for Parallel File Systems. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*. 182–189.

Krishnamurthy, B. and Wills, C. 1998. Piggyback Server Invalidation for Proxy Cache Coherency. In *Proceedings of the Seventh International World Wide Web Conference*.

Kuzmanovic, A. and Knightly, E. 2003. Tcp-lp: A distributed algorithm for low priority data transfer.

Ladin, R., Liskov, B., Shrira, L., and Ghemawat, S. 1992. Providing high availability using lazy replication. *ACM Transactions on Computer Systems 10,* 4, 360–391.

Lamport, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21,* 7 (July).

Lamport, L. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers C-28,* 9 (Sept.), 690–691.

LeFebvre, W. 2001. Cnn.com: Facing a world crisis.

Li, D. and Chariton, D. 1999. Scalable Web Caching of Frequently Updated Objects Using Reliable Multicast. In *Proceedings of the Second USENIX Symposium on Internet Technologies and Systems*. 1–12.

Lin, W.-F., Reinhardt, S., and Burger, D. 2001. Designing a modern memory hierarchy with hardware prefetching. In *IEEE Transactions on Computers special issue on computer systems*. Vol. Vol.50 NO.11.

Lipton, R. and Sandberg, J. 1988. PRAM: A scalable shared memory. Tech. Rep. CS-TR-180-88, Princeton.

Liu, C. and Cao, P. 1997. Maintaining Strong Cache Consistency in the World-Wide Web. In *Proceedings of the Seventeenth International Conference on Distributed Computing Systems*.

Maltzahn, C., Richardson, K., Grunwald, D., and Martin, J. 1999. On Bandwidth Smoothing. In *Proceedings of the 4th International Web Caching Workshop, San Diego, CA*.

Markatos, E. and Chronaki, C. 1998. A Top-10 Approach to Prefetching on the Web. In *INET 1998*.

Mazires, D. 2001. A toolkit for user-level file systems. In *USENIX Technical Conf.* 261–274.

Microsoft. Background intelligent transfer service in windows server 2003. http://http://www.microsoft.com/windowsserver2003/techinfo/overview/bits.mspx.

Mikhailov, M. and Wills, C. 2003. Evaluating a new approach to strong web cache consistency with snapshots of collected content. In *International World Wide Web Conference*.

MOGUL, J. C. 1995. Network Behavior of a Busy Web Server and its Clients. Tech. Rep. WRL 95/5, DEC Western Research Laboratory, Palo Alto, California.

MORRIS, R. 1997. ”tcp behavior with many flows”. In *ICNP 1997*.

MOSBERGER, D. AND JIN, T. 1998. httperf: A tool for measuring web server performance. In *First Workshop on Internet Server Performance*. ACM, 59—67.

MOZILLA. Firefox browser.
http://www.mozilla.org/projects/netlib/Link_Prefetching_FAQ.html.

NAYATE, A., DAHLIN, M., AND IYENGAR, A. 2003. Integrating prefetching and invalidation for transparent replication of dissemination services. Tech. Rep. TR-03-44, Dept. of Computer Sciences, UT Austin. Nov.

NAYATE, A., DAHLIN, M., AND IYENGAR, A. 2004. Transparent information dissemination. In *Proceedings of the ACM/IFIP/USENIX 5th International Middleware Conference*.

PADMANABHAN, V. AND MOGUL, J. 1996a. Using Predictive Prefetching to Improve World Wide Web Latency. In *Proceedings of the ACM SIGCOMM ’96 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. 22–36.

PADMANABHAN, V. N. AND MOGUL, J. C. 1996b. Using predictive prefetching to improve World-Wide Web latency. In *Proceedings of the SIGCOMM*.

PALPANAS, T. 1998. Web prefetching using partial match prediction.

PATTERSON, R., GIBSON, G., GINTING, E., STODOLSKY, D., AND ZELENKA, J. 1995a. Informed Prefetching and Caching. In *Proceedings of the Fifteenth ACMSymposium on Operating Systems Principles*. 79–95.

PATTERSON, R. H., GIBSON, G. A., GINTING, E., STODOLSKY, D., AND ZELENKA, J. 1995b. Informed prefetching and caching. In *SOSP*.

PAXSON, V. 1996. 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*.

PITKOW, J. AND PIROLLI, P. 1999. Mining Longest Repeating Subsequences to Predict World Wide Web Surfing. In *Proceedings of the Second USENIX Symposium on Internet Technologies and Systems*. 139–150.

RESONATE INC. http://www.resonate.com.

RFC 2475. 1999. An architecture for differentiated services. Tech. Rep. RFC-2475, IETF. June.

ROBERT BLUMOFE, C. A. 2002. The challenges of delivering content and applications on the internet.

RODRIGUEZ, P. AND SIBAL, S. 2000. SPREAD: Scalable platform for reliable and efficient automated distribution. In *Proceedings of the 9th International WWW Conference*.

SAITO, Y., KARAMANOLIS, C., KARLSSON, M., AND MAHALINGAM, M. 2002. Taming aggressive replication in the pangaea wide-area file system. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*.

SANGHI, D., AGRAWALA, A. K., GUDMUNDSSON, O., AND JAIN, B. N. 1993. Experimental assessment of end-to-end behavior on internet. In *INFOCOM (2)*. 867–874.

SIEGEL, A. 1992. Performance in flexible distributed file systems. Ph.D. thesis, Cornell.

SINGLA, A., RAMACHANDRAN, U., AND HODGINS, J. 1997. Temporal notions of synchronization and consistency in Beehive. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*.

SIVASIBRAMANIAN, S., ALONSO, G., PIERRE, G., AND VAN STEEN, M. 2005. GlobeDB: Autonomic data replication for web applications. In *Proc. of the 14th International World-Wide Web Conference*. Chiba, Japan. `http://www.globule.org/publi/GADRWA_www2005.html`.

SMITH, A. J. 1978. Sequentiality and prefetching in database systems. *ACM Trans. Database Syst. 3,* 3, 223–247.

SPRING, N. T., CHESIRE, M., BERRYMAN, M., SAHASRANAMAN, V., ANDERSON, T., AND BERSHAD, B. N. 2000. Receiver based management of low bandwidth access links. In *INFOCOM*. 245–254.

SU, Z., YANG, Q., LU, Y., AND ZHANG, H. 2000. Whatnext: A prediction system for web requests using n-gram sequence models. In *WISE ’00: Proceedings of the First International Conference on Web Information Systems Engineering (WISE’00)-Volume 1*. IEEE Computer Society, Washington, DC, USA, 214.

TERRY, B., DEMERS, A., PETERSEN, K., SPREITZER, M., THEIMER, M., AND WELCH, B. 1994. Session guarantees for weakly consistent replicated data. In *Internat. Conf. on Parallel and Distributed Information Systems*. 140–149.

TERRY, D., THEIMER, M., PETERSEN, K., DEMERS, A., SPREITZER, M., AND HAUSER, C. 1995. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the Fifteenth ACMSymposium on Operating Systems Principles*.

VENKATARAMANI, A., KOKKU, R., AND DAHLIN, M. 2002a. System support for background replication. Tech. Rep. TR-02-30, University of Texas at Austin Department of Computer Sciences. May.

VENKATARAMANI, A., KOKKU, R., AND DAHLIN, M. 2002b. TCP-Nice: A mechanism for background transfers. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*.

VENKATARAMANI, A., KOKKU, R., AND DAHLIN, M. 2002c. TCP-Nice: A Mechanism for Background Transfers. In *OSDI*.

VENKATARAMANI, A., KOKKU, R., AND DAHLIN, M. 2002d. TCP-Nice: A mechanism for background transfers (extended). Tech. Rep. TR-02-51, Department of Computer Sciences, University of Texas at Austin.

VENKATARAMANI, A., YALAGANDULA, P., KOKKU, R., SHARIF, S., AND DAHLIN, M. 2001. Potential costs and benefits of long-term prefetching for content-distribution. In *Proceedings of the 2001 Web Caching and Content Distribution Workshop*.

VENKATARAMANI, A., YALAGANDULA, P., KOKKU, R., SHARIF, S., AND DAHLIN, M. 2002. Potential costs and benefits of long-term prefetching for content-distribution. *Elsevier Computer Communications 25,* 4 (Mar.), 367–375.

WCOL. The Prefetching Proxy Server for WWW. http://shika.aist-nara.ac.jp/products/wcol/wcol.html.

WHITE, B., LEPREAU, J., STOLLER, L., RICCI, R., GURUPRASAD, S., NEWBOLD, M., HIBLER, M., BARB, C., AND JOGLEKAR, A. 2002. An integrated experimental environment for distributed systems and networks. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*. 255–270.

WORRELL, K. 1994. Invalidation in Large Scale Network Object Caches. M.S. thesis, University of Colorado, Boulder.

YANG, Y. AND LAM, S. 2000. General AIMD Congestion Control. In *ICNP*.

YIN, J., ALVISI, L., DAHLIN, M., AND IYENGAR, A. 2002a. Engineering scalable consistency. *ACM Transactions on Internet Technologies 2,* 3 (Aug.).

YIN, J., ALVISI, L., DAHLIN, M., AND IYENGAR, A. 2002b. Engineering web cache consistency. *ACM Transactions on Internet Technologies 2,* 3.

YIN, J., ALVISI, L., DAHLIN, M., AND LIN, C. 1999. Volume Leases to Support Consistency in Large-Scale Systems. *IEEE Transactions on Knowledge and Data Engineering*.

YOSHIKAWA, C., CHUN, B., EASTHAM, P., VAHDAT, A., ANDERSON, T., AND CULLER, D. 1997. Using Smart Clients to Build Scalable Services. In *Proceedings of the 1997 USENIX Technical Conference*.

YU, H. AND VAHDAT, A. 2001. The costs and limits of availability for replicated services. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*.

ZEUS TECHNOLOGY. http://www.zeus.com.