Operating System Support for Massive Replication

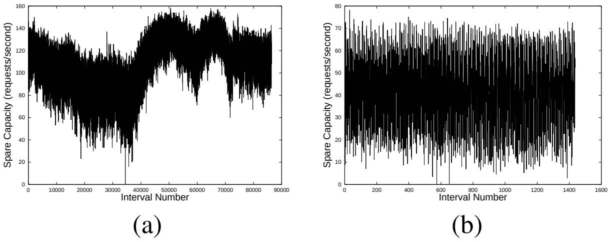
Arun Venkataramani Ravi Kokku Mike Dahlin
{*arun,rkoku,dahlin*}*@cs.utexas.edu*
Computer Sciences, The University of Texas at Austin, USA

1 Introduction

The increasing number of devices used by each user to access data and services and the increasing importance of the data and services available electronically both favor “access-anywhere” network-delivered services. Unfortunately, making such services highly available is difficult. For example, even though end servers or service hosting sites advertise an availability of “four nines” (99.99%) or “five nines” (99.999%), the end-to-end service availability (as perceived by clients) is typically limited to two nines because of poor wide area network availability [6]. Moreover, although network bandwidths are improving quickly, network latencies are much more difficult to improve in wide area networks, which limits performance for access-anywhere services if those services are delivered from a single location.

This paper first argues that operating systems should provide support for massive replication of data and services. In particular, we argue that (1) technology trends favor “wasting” surprisingly large amounts of bandwidth and storage in order to improve availability or latency and (2) system support for massive replication is needed to realize these benefits because hand-tuning by engineers will not work.

This paper then outlines areas where operating system support can facilitate massive replication. We conclude that although a number of useful building blocks exist – particularly in the area of end-host resource management – additional work is needed to develop scalable end-to-end network support for massive replication, to develop client or edge-server support for simultaneously hosting large numbers of applications with essentially unlimited resource demands, and for developing end-to-end abstractions that make programming massive replication applications simple.



(a) (b)

Figure 1. Server loads averaged over (a) 1-second and (b) 1-minute time scales for the IBM sporting event workload.

2 Case for Operating System Support

Operating system support for massive replication is motivated by two factors. First, technology trends suggest that massive replication will be an important building block for a wide range of services. Second, these same trends suggest that it will be difficult to successfully hand-tune applications that use massive replication.

Moving and storing electrons is extremely cheap, so it can make sense to “waste” many electrons to improve human-perceived availability and latency. In particular, the rapidly falling cost of network bandwidth [5, 14] and disk storage [8] – each improving at nearly 100% per year – may call for more aggressive replication than intuition might first suggest. Gray and Shenoy [10] describe a back of the envelope analysis that compares the dollar value of caching data versus the dollar cost of waiting while the data are refetched at some time in the future [10]; Chandra et. al [6] extend this model to consider prefetching and conclude that using an assumption similar to Gray and Shenoy’s estimates of network and disk costs in the year 2000, a system may be economically justified in prefetching an object even if there is only a 1% chance that that object will ever be used. In his Master’s thesis [5], Chandra argues that even more aggressive replication could be justified in the future given disk and network cost trends and given the desire to improve end-to-end availability not just performance.

This work was supported in part by Tivoli Software, IBM Software Group and the Texas Advanced Technology Program through Faculty Partnership Awards. Dahlin was also supported by an NSF CAREER award(CCR-9733842) and and Alfred P. Sloan Fellowship.

Care should be taken in interpreting these results. If everyone started prefetching this aggressively tomorrow, the Internet would likely be overwhelmed. One way of viewing this calculation is that it suggests that economic incentives may exist to grow network capacity over time to accommodate increasingly aggressive prefetching.

A second technology factor that favors massive replication is the burstiness of demand workloads. Figure 1, from Chandra's thesis, shows the request load on an IBM server hosting a major sporting event during 1998 averaged over 1-second and 1-minute intervals. Server loads are bursty at many time scales with significant differences between peak and trough loads. Similar patterns have been noted elsewhere [7]. This burstiness suggests that systems are likely to be built with considerable spare capacity in order to accommodate bursts of load; this spare capacity can often be used to support aggressive replication. Conversely, systems built with the capacity to support aggressive replication will also benefit from an increased ability to handle large bursts of load.

Given these technology and workload trends, it seems likely that large numbers of applications will seek to make use of aggressive replication to improve availability or performance or both. For example, content distribution networks may wish to send updates to replicas before the new versions of objects are requested by clients [20]; multi-replica file systems may wish to immediately propagate all updates to maximize availability and consistency [21]; peer-to-peer systems may wish to replicate directory information [17] or data [20] to multiple replicas; systems supporting mobile clients may wish to replicate portions of file systems or databases to many client machines for disconnected access [12, 16]; and WAN enterprise servers or third-party file service providers [2] may wish to replicate the contents of file systems to geographically-distributed installations to provide the availability and responsiveness of local file systems while providing the global consistency of a WAN file system.

Although the opportunity for aggressive replication exists, it will be difficult for applications to take advantage of massive replication without end-to-end system support. A well-known problem with prefetching is that it consumes more resources than demand replication because some prefetched data is not used. Both *self-interference* – a prefetching application should ensure that its prefetch requests do not interfere with its demand requests – and *cross-interference* – prefetch requests should not interfere with other applications' demand requests – should be minimized.

Most often, this problem is addressed by hand tuning. For example, many prefetching algorithms estimate the probability that an object will be used, and then prefetch objects whose probability of use exceeds a threshold [9, 20].

Unfortunately, hand tuning seems unlikely to work for WAN massive replication.

- First, as noted above, intuition may be a poor guide for balancing the benefits and costs of prefetching. For example, Duchamp [9] selects a prefetch threshold of 25% as a reasonable balance between wasted bandwidth and latency reduction; the analysis discussed above suggests that this threshold may be far too conservative in many environments.
- Second, because of network cost and disk storage cost technology trends, the break-even point for prefetching will change significantly from year to year.
- Third, because of bursty workloads, the spare capacity available at a given point in time will change from second to second or minute to minute.
- Fourth, the complexity of hand tuning, and the lack of end-to-end support to ensure that prefetching requests do not interfere with demand requests discourages deployment of aggressive replication applications by (a) making the implementation of such systems more complex or fragile or both and (b) forcing conscientious application writers to be extremely cautious in their designs.

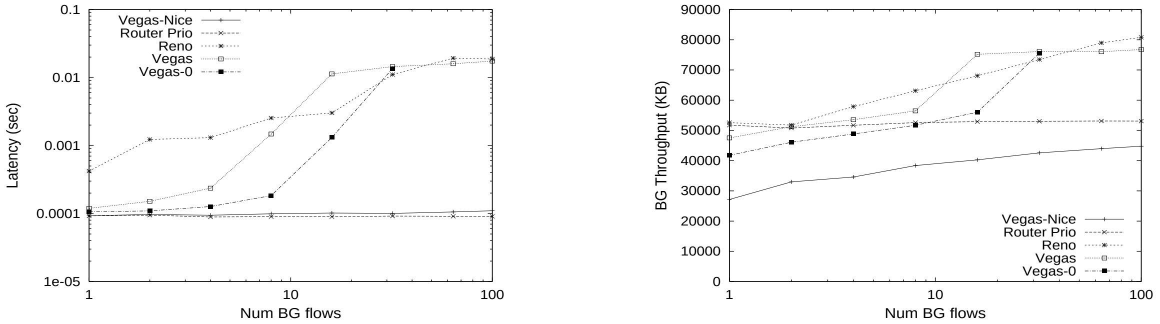
We believe that system support for massive replication should encourage development and deployment of applications that use massive replication to improve availability and performance. In particular, such support would allow simple applications that just state what they wish to replicate; the underlying system should be "self-tuning" and replicate as much as can be done without interfering with demand requests.

3 OS Support for Massive Replication

In this section we introduce a technique that allows us to allocate network resources in a manner such that a flow can be transmitted across the network without interfering with the flows already present. Thus, if there is spare capacity in the network, such flows can be efficiently transmitted; if not, the network will automatically ensure minimal self and cross-intereference with other flows. We then discuss resource management issues at the end-stations to minimize interference between competing applications.

3.1 Network

Aggressively replicating and maintaining copies of data across the Internet can potentially consume virtually unbounded amounts of network bandwidth and interfere with existing applications. Since the network is a shared resource, it is essential to ensure minimal interference. However, a key question is whether network resource management be done in the network or at the end? One one hand



(a) Foreground latency vs. number of background flows (b) Background throughput vs. number of background flows

Figure 2. Performance comparison of Reno, Router Prioritization, Nice and Vegas

simple prioritization schemes implemented at the routers such as the ones proposed for DiffServ [1] can easily prevent low priority flows from interfering with high priority flows. Unfortunately, there are practical hurdles to the immediate widespread deployability of such schemes.

We examine the other approach to develop an end-to-end transport protocol that approximates router prioritization without actually modifying the routers. We have developed a new congestion control algorithm, namely TCP *Nice* [19] as an end-to-end solution to the problem of minimizing interference. Nice is a simple extension to the TCP Vegas [4] congestion control protocol. Vegas uses round trip time (RTT) to limit the number of packets enqueued by a flow in the bottleneck router, so it provides a reasonable basis for conservative end-to-end congestion control. Unfortunately, Vegas was designed to compete fairly with TCP-Reno, so using it does not prevent background flows from interfering with foreground flows. The Nice extension makes the protocol much more responsive to competing traffic by adding an additional RTT-based congestion detection rule and by backing off multiplicatively when this rule detects congestion.

It is simple to make a protocol that behaves less aggressively than Vegas. The challenge is to meet two conflicting goals. First, background flows should not interfere with (e.g., increase the latency of) foreground flows. Second, demanding background flows should be able to consume a large fraction of the bandwidth not consumed by foreground flows. Our preliminary simulation based analysis with Nice suggests that it meets these goals effectively.

The set of graphs in Figure 2 show results of our simulation experiments over a dumbbell-shaped network topology with one bottleneck link connecting 20 clients and a server. The workload used for the simulations is a 15 minute trace of HTTP requests logged by Squid at UC Berkeley and a set of permanently backlogged background flows.

Figure 2(a) plots the latency of foreground requests as a function of the number of background flows for a network utilization of 50%, when the background flows use Reno, RouterPrio, Nice and Vegas respectively. It can be seen that while Reno causes the latency to blow up by an order of magnitude because of interference, Nice causes little increase in latency compared to router prioritization and increases gracefully with the number of background flows. Figure 2(b) shows that the throughput attained by the background flows gets reduced (almost halved when there is just one background flow), but is comparable with RouterPrio as the number of flows increases. However, the lower background throughput is a reasonable trade-off for the agility that Nice provides in avoiding interference. Reno and Vegas on the other hand steal bandwidth from the foreground requests and hence delay them considerably.

In order for the application to be able to specify whether the data being sent is background or foreground traffic, we propose to have the transport layer at the sender expose a suitable API. This functionality can be provided by a lightweight *interference manager* just above the transport layer. The interference manager decides whether to transmit the data as regular foreground traffic or use Nice with an appropriately tuned *niceness* level. The interface manager may also aggregate appropriate sets of background requests into a single flow so as to create constantly backlogged flows.

3.2 End-stations

Two issues complicate dealing with interference at end-stations – i) end systems should deal with efficient allocation of multiple resources like CPU, disk and memory. ii) deployment of large number of services together with aggressive replication by each of them effectively implies near infinite demand for resources. For example, CDNs have to deal with thousands of demanding services and mobile clients that are capable of disconnected services [6], have to deal with tens or hundreds of services all contending for the system resources.

With respect to the first problem, many existing techniques can be adopted for efficiently allocating multiple resources such that interference is minimized. Resource containers [3] allow us to define resource principals appropriate to the application. Systems such as Qlinux [15] enable the provision of QoS guarantees with repect to CPU, disk and network bandwidth.

The second problem, resource management across multiple services involved in massive replication, is more challenging. These challenges will affect the design of operating systems for hosting systems and operating systems for light-weight clients. The Active Names system has been adapted to provide fair self-tuning division of resources across downloaded extension code modules [6]. These techniques could be adapted to partition resources for downloaded code at clients, proxies, and hosting centers. Hosting systems like Denali [11] focus on providing services to safely execute many services on a single physical machine. However, in order to support massive replication, Denali should also provide scalable multi-resource management that prevents demanding applications from interfering with each other.

4 Policy

Designing massive replication systems also throws up questions of policy, namely *object selection* - what to replicate and *placement* - where to replicate. Our past research shows that by prefetching objects based on their popularity and lifetimes [20], significant improvements in hit rates may be obtained. In [13] and [18], we present algorithms to place objects in a distributed caching system so as to maximize hit rates in scenarios constrained by space and bandwidth respectively.

In future, we intend to work on developing algorithms to place objects in more dynamic scenarios where both space and bandwidth are simultaneous constraints. These algorithms require gathering good statistics of object usage patterns. Statistics should be gathered both on global scale (across various services to reflect the effects of services on each other) and on local scale (within a service to measure the service access patterns). Gathering good statistics forces us to strike a tradeoff between the precision of measurements and the scalability of the gathering mechanism. We intend to quantify this tradeoff and study its effects.

5 Conclusion

In this paper we have presented the challenges involved in providing system support for massive replication and an overview of their effect on operating system design at the network, server and edge nodes. We have also presented TCP Nice, an end-to-end congestion control algorithm optimized to support background transfers. We argue that the design of applications relying on replication can be made simpler if the underlying operating system provides self-tuning support for resource management.

References

[1] http://www.ietf.org/html.charters/diffserv-charter.html.

[2] Akamai. Fast internet content delivery with freeflow. In *White Paper*, Nov 1999.

[3] G. Banga, P. Druschel, and J.C. Mogul. Resource containers: A new facility for resource management in server systems. In *OSDI*, 1999.

[4] L. Brakmo, S. O'Malley, and L. Peterson. Tcp vegas: New techniques for congestion detection and avoidance. In *Proceedings of SIGCOMM'94 Conference*, 1994.

[5] B. Chandra. Web workloads influencing disconnected services access. Master's thesis, UT Austin, 2001.

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

[7] J. Chase, D. Anderson, P. Thakar, A. Vahdat, and R. Doyle. Managing Energy and Server Resources in Hosting Centres. In *SOSP 2001*.

[8] M. Dahlin. Historical disk storage costs, mar 2002. http://cs.utexas.edu/∼ dahlin/techTrends/data/diskPrices/data.

[9] D. Duchamp. Prefetching Hyperlinks. In *Proceedings of the USITS*, October 1999.

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

[11] S. Gribble, A. Whittaker, and M. Shaw. Denali: Lightweight virtual machines for distributed and networked applications. Technical Report 02-02-01, Univ. of Washington, 2002.

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

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

[14] A. Odlyzko. Internet growth: Myth and reality, use and abuse. *Journal of COmputer Resource Management*, 2001.

[15] V. Sundaram, A. Chandra, P. Goyal, P.J. Shenoy, J. Sahni, and H.M. Vin. Application performance in the QLinux multimedia operating system. In *ACM Multimedia*, 2000.

[16] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of SOSP 1995*, pages 172–183, December 1995.

[17] R. Tewari, M. Dahlin, H. Vin, and J. Kay. Design Considerations for Distributed Caching on the Internet. In *Proceedings of ICDCS 1999*, May 1999.

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

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

[20] A. Venkataramani, P. Yalagandula, R. Kokku, S. Sharif, and M. Dahlin. Potential costs and benefits of long-term prefetching for content-distribution. In *WCW*, June 2001.

[21] H. Yu and Amin Vahdat. The Costs and Limits of Availability for Replicated Services. In *Proceedings of SOSP 2001*, 2001.