Resource management for scalable disconnected access to web services *

Bharat Chandra, Mike Dahlin, Lei Gao, Amjad-Ali Khoja
Amol Nayate, Asim Razzaq, Anil Sewani

Department of Computer Sciences
The University of Texas at Austin

ABSTRACT

Disconnected operation, in which a client accesses a service without relying on network connectivity, is crucial for improving availability, supporting mobility, and providing responsive performance. Because many web services are not cachable, disconnected access to web services may require mobile service code to execute in client caches. Unfortunately, (a) this code is untrusted, (b) this code may have nearly limitless resource demands due to prefetching, and (c) a large number of competing code modules must coexist. Thus, resource management is a key problem both for preventing denial of service attacks and for providing good performance across many services. This paper addresses the feasibility of meeting the resource management needs of an environment where service code is shipped to clients, proxies, or content distribution intermediaries. It first examines the requirements of such a system and then develops a resource-management strategy to meet these requirements by (a) providing isolation across services to prevent denial of service attacks, (b) automatically providing appropriate allocations to different services to provide good global performance, and (c) requiring no hand tuning across a wide range of system configurations and workloads.

1. INTRODUCTION

This paper examines resource management issues for constructing a scalable infrastructure to support disconnected access to web services. We focus on environments that allow web services to ship service code to caches and proxies and that allow this code to use prefetching, hoarding [1, 17, 18], write buffering, persistent message queues [7, 16], and application-specific adaptation [21] to mask disconnections by satisfying requests locally. Several researchers have proposed such systems [5, 16, 30]. However, web workloads pose a key scalability and resource management challenge: systems must provide a framework to allow hundreds of different services to use these techniques without interfering with one another.

Support for disconnected operation allows clients to access web services without relying on the network connection between the client and the origin server. Supporting disconnected operation is a key problem for improving web services for three reasons.

1. Disconnected operation allows mobile clients to access services when their network connection is unavailable, expensive, or slow.
2. Disconnected operation can improve service availability. Studies consistently find that, in contrast with targets of "four nines" or "five nines" of availability (99.99% uptime or 99.999% uptime) for important services, the Internet network layer provides only about two nines of host-to-host connection availability [6, 23, 25, 32]. The resulting average of about 14 minutes per day of unavailability to a typical client hinders commercial sites such as information sites and commerce sites, and it prevents the use of the standard web infrastructure for mission-critical sites such as a hospital medical information access and order-dispatching service.
3. Disconnected operation can significantly improve performance. Traditional web caching is a simple example of this strategy, and several studies have demonstrated even more dramatic speedups when service code is shipped to clients and proxies [5, 30].

Infrastructure for gaining these benefits has been developed for file systems that use caching, hoarding, and write buffers to support disconnected file access [17]. Unfortunately, modern web services no longer treat HTTP's Get/Put interface as a simple file Read/Write interface. Instead, many HTTP requests are essentially arbitrary RPCs that are not cachable using traditional means. Wolman et al. [31] find that uncachable web accesses reduce the upper bound on cache hit rates by about a factor of two. In the context of disconnected operation, Chandra et al. [6] find that if an infrastructure supports mobile service code for disconnected operation, it can reduce average service unavailability by factors of 2.7 to 15.4, but if an infrastructure only supports caching and aggressive prefetching, average improvements are limited to 1.8 to 6.2.

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

Copyright is held by the author/owner.
WWW10, May 1-5, 2001, Hong Kong.
ACM 1-58113-348-0/01/0005.

Although Java applets and Javascript allow services to ship code to clients, they do not provide the infrastructure needed to support scalable disconnected service access where large numbers of services use mobile code to mask disconnection. On one hand, these systems are too restrictive. They prevent access to disk by untrusted code, and they can evict stored state without warning. These restrictions prevent these systems from supporting crucial building blocks for disconnected operation such as hoarding, prefetching, write buffering, and persistent message queuing. On the other hand, these systems are too permissive. Current implementations provide no limits on the memory space, network bandwidth, or CPU cycles consumed by untrusted code. As a result, systems are vulnerable to denial of service attacks, and they have no way to partition resources fairly among downloaded services.

Although several experimental systems have provided low level mechanisms for limiting resources consumed by untrusted Java code [3, 8], properties of web service workloads make it challenging to develop a scalable infrastructure for disconnected access. First, clients may access a large number of services, meaning that many untrusted services will compete for resources. Second, the resources available at client devices may vary widely. Third, prefetching and hoarding – key techniques for coping with disconnected operation – can dramatically increase the resource demands of applications: Ironically, providing mobile code with the ability to access disk to support disconnected operation worsens the resource management challenge because it gives applications an incentive to use more resources. Finally, the large number of services and the diverse user population preclude solutions that require user intervention to manage system resources.

Although this paper focuses on an aggressive point in the web-service design space – caching service code at clients, proxies, or content-distribution nodes – many of the resource management issues we examine arise in less aggressive service architectures. For example, systems such as AvantGo [2] for palm-size computers or Microsoft Internet Explorer [19] that provide hoarding for disconnected operation must divide storage space and network bandwidth across services. And efforts such as ICAP [29] to allow services to install code near clients must manage a wide range of resources that are shared across competing services.

This paper makes three contributions. First, we quantify the resource requirements of disconnected services by examining both the general requirements across services and the requirements of several case-study services. These data suggest that supporting disconnected operation for a large number of services is feasible. In particular, we argue that prefetching an order of magnitude more data than is used on demand may often be reasonable. However, we also find that careful resource management by the system and application-specific adaptation by the services is needed. Second, we develop a resource management framework that (i) provides *efficient allocation* across extensions to give important extensions more resources than less important ones, (ii) provides *performance isolation* so that aggressive extensions do not interfere with passive ones, and (iii) makes allocation automatic *self-tuning* decisions without relying on user input or directions from untrusted extensions. Third, we develop a prototype system that provides these resource management abstractions for mobile servelets.

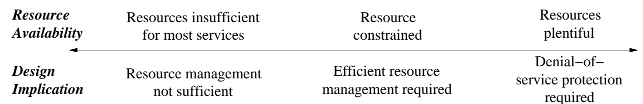


Figure 1: Design space for resource management.

The rest of this paper proceeds as follows. Section 2 studies the system requirements imposed by the workload. Section 3 describes and evaluates a simple resource management policy for this environment. Section 4 describes our prototype. Finally, Sections 5 and 6 discuss related work and our conclusions.

2. REQUIREMENTS

This section examines the impact of web workloads on the requirements for infrastructures that support disconnected service access. Systems could operate in one of three regimes illustrated by Figure 1. First, services may demand large amounts of resources in order to support disconnected operation, and the aggregate demands of services may significantly exceed the total capacity of the infrastructure. In that case, providing a general infrastructure for disconnected operation may not be feasible. At the other extreme, resources may be plentiful relative to the likely demands of services. In that case, infrastructure should focus on providing mechanisms for shipping code to clients, running that code securely, and, perhaps, limiting "resource hogs" to prevent deliberate or unintentional denial of service; beyond that, resource management is not likely to determine system performance. The middle case is more challenging: if resources are constrained but sufficient for applications to provide reasonable disconnected service, then a key task for the infrastructure is partitioning resources fairly among untrusted code modules to maximize global utility.

It is difficult to specify workload requirements definitively. First, applications vary widely. Some can easily operate in disconnected mode with few additional resources compared to their normal requirements. For example, a daily comic site could be prefetched in its entirety once per day for little more cost than fetching it on demand. Other services are not suitable for disconnected operation at all because they require live network communication (e.g., a stock ticker or phone call) or would require unreasonable amounts of state to be replicated at clients for disconnected operation (e.g., a search engine). Many services may operate between these extremes: by expending additional resources (i.e., prefetching data that may be needed in the future or buffering writes or outgoing requests) they can support disconnected operation. Examples of this class may include many shopping, news, entertainment, and corporate services.

Note that the application-specific adaptation afforded by mobile code often may allow services to provide degraded, though still useful, service when disconnected. For example, although a stock trading service probably would not accept orders when disconnected, the company providing the service may desire to operate in "degraded" disconnected mode by turning off the order service but providing other services: a portfolio summary, news bulletins related to the user's holdings, a history of past orders, and so on. In this example, even though the "primary" function for a service is inoperable when disconnected, the service may gain significant benefit from mobile code that allows the user to access a subset of the services when disconnected.

A second challenge to precisely specifying workload requirements is that the potential demands of an individual

service may span a wide range. In particular, prefetching is a common technique to cope with failures. Often, the more data that are prefetched the larger fraction of client requests that can be handled during disconnection, and the better service that can be provided. Thus, over some range of demands, using incrementally more resources can yield incrementally better service when disconnected. For example, a news service might prefetch headlines and abstracts in a resource constrained environment, full text of articles from a few major sections (e.g., headlines, international, sport, finance) in a less constrained environment, and so on up to full text, pictures, and video from all articles in an unconstrained environment.

Given the methodological challenges posed by the wide range of web service behaviors, we take the following approach. First, we examine the average demands of current web workloads in order to assess approximately how many additional resources may be available for supporting disconnected operation. This provides a rough guide to the constraints of the system. Second, we examine several case study workloads to determine their resource requirements to provide different levels of service.

We focus primarily on the bandwidth and space requirements of hoarding and related techniques such as prefetching. Although other techniques – write-buffering message queues, and application-specific adaptation – are also important for coping with disconnection, the resource demands of services using these techniques may not be significantly higher than the normal demands of the services. In contrast, aggressive hoarding may dramatically increase the network bandwidth and disk space demands of applications and therefore presents the most direct challenge to scalability.

2.1 Workload characteristics

The operating regime of the system with respect to resources is largely determined by the workload. We study several client traces of web service workloads to determine how many services are in a client's working set and how much data those services access. From this, we derive an estimate of the amount of spare capacity machines are likely to have to support disconnected operation and argue that it may be feasible for services to prefetch 10 or more times as much data as they access on demand.

We analyze two traces: Squid [26], which contains 7 days (3/28/00 – 4/03/00) of accesses to the Squid regional cache at NCAR in Boulder, Colorado that serves requests that miss in lower-level Squid caches, and the first seven days from UC Berkeley Home-IP HTTP traces [13]. The simulator uses information in the traces to identify cachable and non-cachable pages as well as stale pages that require reloads. In this analysis, we study the resource demands from each service, where a "service" is defined by the set of URLs from the same DNS domain (for the Squid trace) or from the same IP address (for the UCB trace, which identifies origin servers using a 1-way hash of the IP address).

We study two cache configurations. In the first, we simulate a separate cache at each client IP address in the trace. Since the UCB trace tracks individual client machines, this corresponds to the environment that might be seen by code that seeks to support mobile clients as well as to improve client performance and availability. In the second, we simulate a proxy cache shared by all clients in the trace. This configuration does not support client mobility, but it may improve service availability or performance. Note that the Squid traces remap client IDs each day, so we only examine the first day of the Squid workload in our per-client cache analysis. We refer to this workload as Squid-1-day for clarity. Also note that for the Squid trace, some clients may correspond to lower-level Squid proxy caches that aggregate requests from collections of users.

Figure 2 summarizes the number of services accessed by each cache over different time scales to provide a rough guide to the "working set" size of the number of services a cache might have to host. Each graph summarizes data for a different trace/cache configuration. Each graph shows the minimum, 25th percentile, median, 75th percentile, and maximum number of services accessed by a cache over intervals of the specified length. Only UCB statistics shown due to space constraints.

Figure 3 summarizes the distribution of the per-client maximum number of services accessed at different interval sizes for UCB. For example, if a client accesses 3 services during the first hour, 7 during the second, and 2 during the third, that client's maximum working set size for 1 hour is 7 services. The plot shows the range of maximums at different clients.

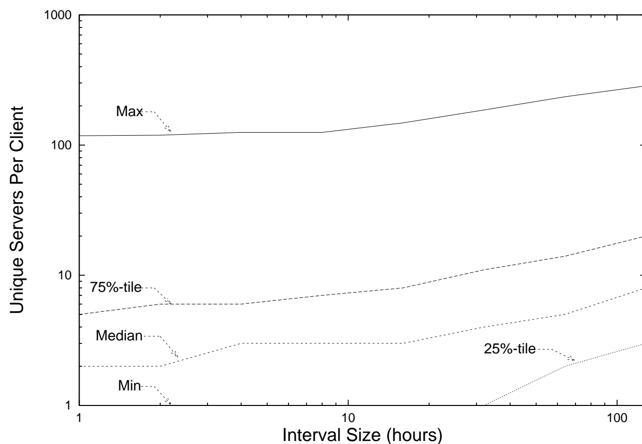
Two features of these distributions stand out. First, the working sets of a cache can be large over even modest time scales. For caches at individual clients, 25% of 16-hour-long intervals contain accesses to more than 10 services in the UCB trace and 200 services in the Squid-1-day trace(not shown). For proxy caches, 25% of 16-hour-long intervals contain accesses to more than 8,000 services in the UCB trace and 18,000 services in the Squid trace.

Second, these working sets vary widely from client to client. For example, in the UCB trace 25% of clients never use more than 3 services in a 16-hour period and in the squid trace 25% use at least 148 services during at least one 16-hour period.

These features have several implications on the system design. First, they suggest that resource management could be a significant challenge for some caches where many services compete for service. They also suggest that the framework must be self-tuning over a wide range of situations both because the range of demands is large and because the number of services under consideration is often too large for convenient hand-tuning.

Figures 4 and 5 show the disk space consumption of individual services and of the collection of services hosted by a cache, respectively. In Figure 4, the x-axis is the approximate service working set size (the amount of data the service accesses during the trace) and the y-axis is the fraction of services with working sets of the specified size or smaller. In Figure 5, we plot the total disk size consumed by all services at each client on the x-axis with the fraction of clients with disk consumption below the specified amount on the y-axis.

The graphs indicate that per-service demand fetched data typically have modest footprints in caches; for the Squid and UCB traces, 90% and 80% of services consume less than 100KB. A few large services consume large amounts of disk space. Overall, for the UCB per-client caches, the total data footprint of all services accessed by a cache is below 10MB for all but a few percent of clients. The Squid data footprints are significantly larger, but recall that each "client" in the Squid trace may correspond to a lower-level Squid proxy



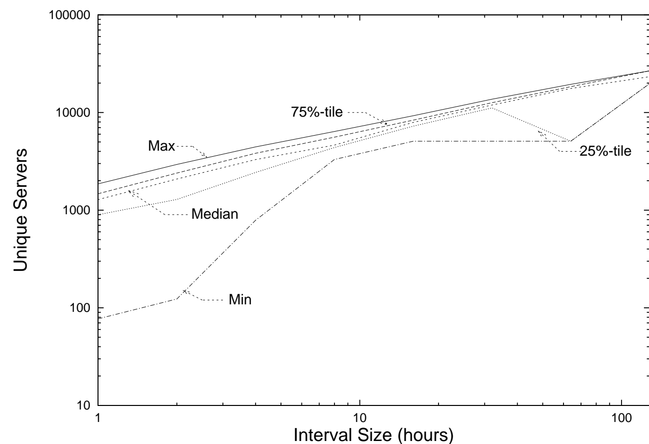


Figure 2: The number of services accessed by (a) a per-client cache or (b) an active shared proxy over different time scales for UCB trace file.

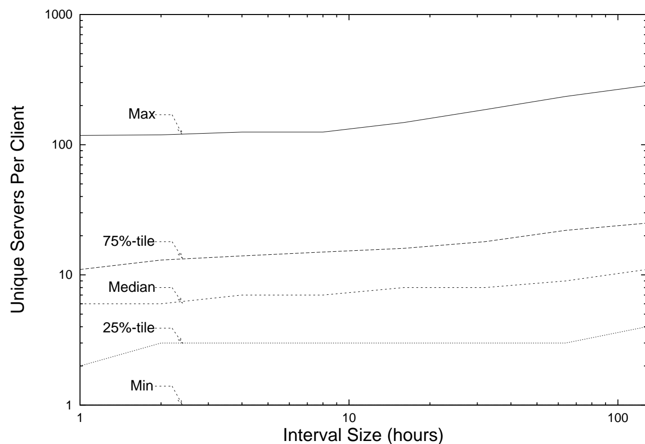


Figure 3: The range of the maximum number of services accessed by different clients for the UCB trace.

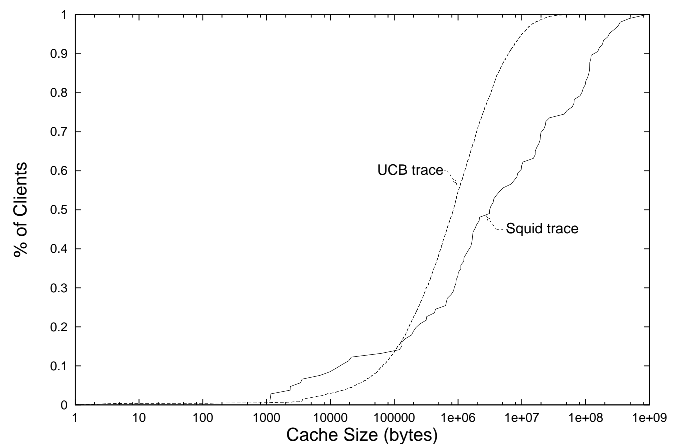


Figure 5: Per-client cache size demands.

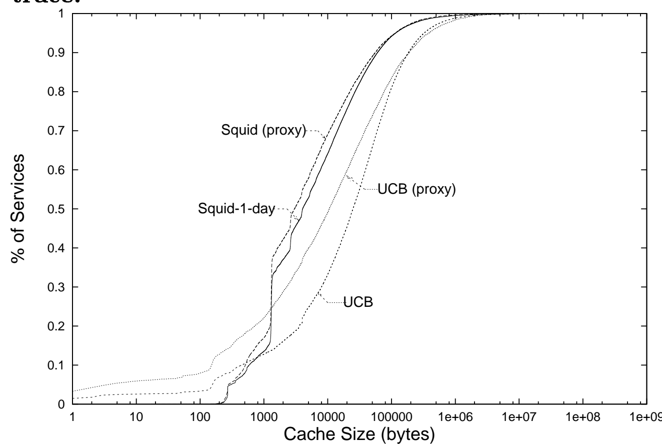


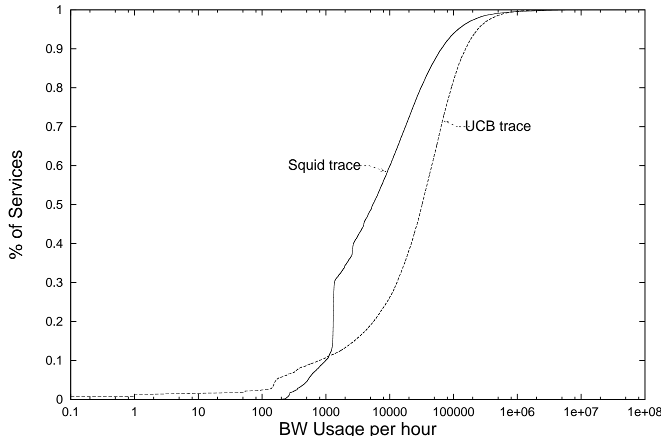
Figure 4: Per-service cache size demands. Shows the cumulative histogram of the faction of services that occupy the specified size at the end of the trace.

serving many clients.

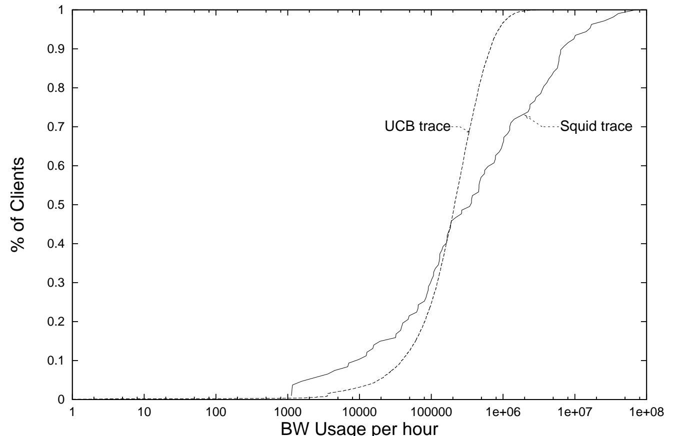
These data have several implications with respect to scalable resource management. First, the wide range of per-service and per-cache demands suggests the need for a flexible approach. For example, allocating 100KB to each service would waste large amounts of space for many services and be far to little space for others. Second, for the desktop clients that presumably make up most of the UCB trace, the amount of disk space consumed for caching demand-fetched objects is relatively small compared to the total disk capacity of such machines. This suggests that disk space considerations may allow significant prefetching. We discuss this issue in more detail below.

Another key resource for supporting disconnected operation is network bandwidth. Figure 6 summarizes the network bandwidth consumption by the trace workloads. As indicated in Figure 6(a), most services have low average bandwidth requirements of a few tens of KB per hour or less. This suggests that caches with modest bandwidth can support relatively large number of prefetching services. Figure 6(b) shows the hourly bandwidth usage at each client. 90% of clients demand less than 2 MB/hour – 8% of a 56Kbit/s modem and 0.4% of a 1Mbit/s DSL. This suggests that considerable spare bandwidth may be available.

Similar analysis for maximum bandwidth usage shows the distribution across clients and services of the maximum bandwidth demand during any hour for that client (graph not shown). 90% of clients demand less than 10MB per hour – 40% of a 56Kbit/s modem and 2% of 1Mbit/s DSL connection – during their worst hour. Most services need only a few hundreds of KB in their worst hour. This suggests that considerable spare bandwidth may be available even during



(a) Per-service distribution



(b) Per-client distribution

Figure 6: Distribution of average bandwidth usage.

periods of high demand.

Discussion. The above data suggest that in terms of raw capacity, client disks and networks may be able to provide considerable resources to support disconnected operation. For example, in the UCB workload, 95% of client systems could allow each service to prefetch approximately 10 bytes for every byte served on demand and still consume less than 1% of a 10GB disk. Similarly, a client on a 1MBit/s DSL connection could prefetch 10 bytes for every byte served on demand and consume less than 4% of the raw connection bandwidth for most clients. Given this spare capacity, one might ask: is it reasonable for a service to fetch and store 10 or even 100 times as much data as is accessed on demand?

In classic operating systems terms, fetching 10 or 100 bytes per byte accessed might seem excessive. However, the long latencies and significant failure rates of wide-area networks may make doing so a reasonable strategy. Furthermore, whereas the costs of prefetching (network bandwidth, server processing, and disk space) fall rapidly over time, the value of human time remains approximately constant. Thus, prefetching a large number of bytes to save a small amount of latency or risk of disconnection becomes a more attractive strategy over time.

Gray and Shenoy [12] suggest a methodology for estimating whether caching a data object is economically justified by comparing the cost of storage against the cost of network bandwidth and human waiting time. They estimate that downloading a 10KB object across an Internet/LAN network costs about $NWCost_{LAN}$ = \$.0001, across a Modem costs about $NWCost_{Modem}$ = \$.0002, across a wireless modem about $NWCost_{Wireless}$ = \$.01, and they estimate that storing a 10KB object costs about $StorageCost$ = $\$8 \cdot 10^{-6}$/month. Assuming that human time is worth about \$20/hour, the “waiting cost” due to cache miss latency is about $WaitCost_{LAN}$ =\$.02 when connected via an LAN/Internet, $WaitCost_{Modem}$ =\$.03 when connected via a modem, and $WaitCost_{Wireless}$ = \$.07 when connected via a wireless modem. Based on these estimates, they conclude, for example, that once fetched, an object should be kept in cache even if the expected time to reaccess it is on the order of decades.

That methodology can be extended to estimate whether

	Network used for demand fetch		
Prefetch NW ⇓	LAN/Internet	Modem	Wireless
LAN/Internet	0.0054	0.0036	0.0014
Modem		0.0069	0.0026
Wireless			0.1251

Table 1: Estimate of the P_B break-even probability that a prefetched object is used that justifies prefetching it to reduce human waiting time to demand fetch it (based on Gray and Shenoy's year 2000 estimates of cost data).

prefetching an object is justified. Suppose that P_{used} is the probability that a prefetched object is used before it is updated or evicted from the cache and that unreferenced prefetched objects are evicted after one month. Then the break-even probability P_B can be defined as the lowest P_{used} for an object that justifies prefetching it:

$$P_B = \frac{NWCost_{prefetchNW} + StorageCost}{WaitCost_{demandNW} + NWCost_{demandNW}}$$

Table 1 summarizes our estimates of P_B. The rows correspond to different networks used for prefetching ($prefetchNW$), and the columns to different networks used for demand fetch ($demandNW$). The diagonal corresponds to prefetching on the same network as the later demand fetch. For a LAN/Internet environment, for example, it can make sense to prefetch an object if the user has a 0.5% chance of accessing it over the next month.

This economic argument for aggressive prefetching is particularly attractive for supporting heterogeneous networks or disconnected operation. As the table indicates, prefetching when network bandwidth is cheap (e.g., on a LAN connected to the Internet) to avoid network traffic when it is expensive (e.g., on a wireless modem) can be advantageous even if there is a 1/500 chance that an object will be used. Furthermore, in the case of disconnected operation, the $WaitCost$ term would be replaced by a (typically higher) *DenialOfService* term that represents the cost of not delivering the service to the user at all.

One factor ignored in these calculations is server costs. One concern about aggressive prefetching is that it could swamp servers. Our system controls this by putting prefetching under the control of programs supplied by servers. This allows servers to avoid prefetching when it

would interfere with demand traffic. More generally, the above methodology can be extended to include server processing costs.

Another argument against aggressive prefetching is that it may overwhelm the Internet infrastructure. Certainly, if everyone started aggressively prefetching tomorrow, capacity would be stretched. One way of viewing this calculation is that it suggests that the economic incentives to grow network capacity over time to accommodate prefetching may exist. Odlyzko [22], for example, suggests that in rapidly-growing communications systems, capacity and demand increases seldom occur at more than 100% per year, partially due to the time needed to diffuse new applications with new demands.

Another factor ignored in these calculations is the presence of heterogeneous devices. For example, a palmtop machine may have significantly less storage space than a desktop machine. This can be modeled by increasing the $StorageCost$ term [12].

Technology trends may favor more aggressive replication in the future. First, network and disk costs appear likely to fall much more quickly than human waiting time costs. Second, the deployment of commercial content-distribution networks (CDNs) may significantly reduce the network costs of prefetching by allowing clients to prefetch from a nearby CDN node rather than the origin server.

Overall, the raw capacity of disks and networks as well as the back of the envelope economic calculations all suggest that in many current environments, it may be reasonable for services to use significant amounts of spare capacity to support disconnected operation. We conclude that for web service workloads, capacity to support prefetching of ten times more data than is used on demand may often be available, though support to prefetch 100 times more data than is used may be excessive for today's configurations. This factor, however, may increase in the future.

2.2 Case study application

The above section provides a sense of how much spare capacity a service might have available to it for prefetching or hoarding. This section examines a case study workload to illustrate the range of demands such services can require. We focus on prefetching for the www.cnn.com news service because it is typically referenced by a significant number of clients in the daily NLANR Squid access logs. This service is, of course, not representative of the broad range of web services, but it serves to illustrate various techniques that services may use. In previous work [9, 30], we illustrate several additional approaches for supporting disconnected operation including advertisement rotation and logging, a "disconnected catalog" that hoards catalog contents and buffers orders, and a hospital order-transmission system. Here, we focus on prefetching behavior.

As discussed above, services can choose from an almost limitless number of different algorithms for trading prefetch bandwidth for performance. Our goal is not to identify the "best" algorithm for prefetching services in general or this service in particular. Instead, we seek to understand the extent of the design space.

We simulate five algorithms:

1. Demand. Clients maintain an infinite cache but do not prefetch.
2. Oracle. An oracle prefetches the current version of any uncached or stale object immediately before a client references it.
3. Push-updates. The system pushes updates of objects that the client is caching. A parameter is the *time-granularity*: how often are updates pushed to clients. We examine two values: 15 minutes (which is the frequency at which we take snapshots of server state) and 1 hour.
4. Push-new. When a new object is created, the server immediately pushes it to all subscribed clients. After that, push updates of such objects according to the push update protocol.
5. Headlines. This is a CNN-specific algorithm. Push objects attached to the first N links under each major section (e.g., "Sports", "Finance", "International") to each client.

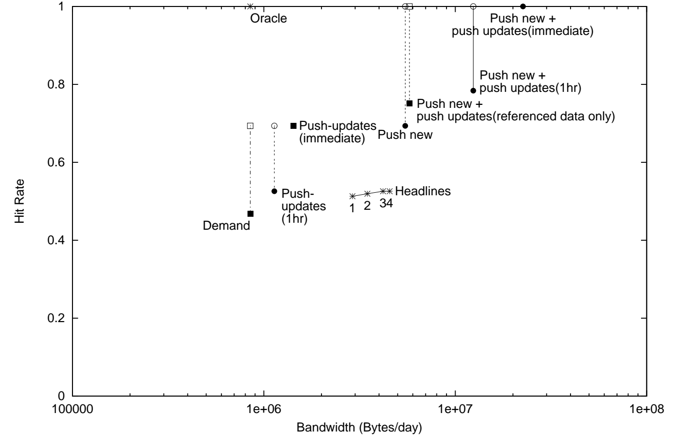


Figure 7: Range of trade-offs for CNN prefetch algorithms. Filled points show hit rates and unfilled points show hit rates including hits to stale objects.

To study these algorithms, we periodically scan http://www.cnn.com/index.html for all files (including HTML and images, but excluding streaming media) within the cnn.com domain referenced directly by that page. We gather a simultaneous trace of requests to this service from the daily NLANR Squid cache logs. We focus on the subset of requests to the "CNN-top" pseudo-service represented by the pages reached by our scan.

In our simulations, we assume an infinite client cache, and we use delta-encoding (via diff+gzip) of non-image files to transfer updates.

In Figure 7 we plot the network cost of each algorithm as the x-axis and use hit rate – the fraction of requests that can be satisfied locally – as a figure of merit. For each algorithm we show the hit rate considering both the hit rate to fresh objects and the total hit rate including hits to "stale" objects. In the graph, the fresh hit rates are shown with filled symbols and the total (fresh + stale) hit rates are shown with outline symbols.

Note that the presence of application-specific adaptation makes "hit rate" an imperfect figure of merit. In particular, when disconnected, the mobile code that implements our CNN service shows users only the subset of locally stored stories. Thus, rather than allowing users to click a link and see a failure, the system hides useless links so that the user will be less aware of missing articles. We speculate that by using application-specific prioritization of articles such as the Headlines algorithm above, and by controlling

presentation, this service would provide high utility to users with a modest fraction of its total set of articles.

This set of algorithms illustrates the a broad range of trade-offs between network bandwidth consumption and hit rate available to many web services. Push updates provides significant improvements to fresh hit rates for modest bandwidth costs. Push-new with different push-updates variations allows the system to achieve total hit rates of 100% and fresh hit rates ranging from 52% to 100% for bandwidths from 1.1 MB/day to 22.6 MB/day. The Headlines algorithm uses bandwidth from 2.9 MB/day to 4.5 MB/day but it achieves modest hit rate improvements; we speculate that the improvement in utility to the service may be higher than indicated by the hit rates. Finally note that the range of bandwidth demands extends further: in this simulation we have not prefetched streaming media files and we have only prefetched the "CNN-top" subset of the service.

3. POLICY

In this section we first outline the requirements for resource management in this environment. We then examine a simple policy that appears to meet these requirements.

1. **Isolation.** The policy should prevent denial of service attacks by bounding the resources consumed by any service, and it should prevent aggressive services from interfering with other services. This goal is motivated by the fact that the target workload comprises large numbers of untrusted modules competing for resources.
2. **Efficiency.** The policy should divide resources among services so as to maximize overall utility. In contrast with the first goal, which might be achieved by placing a loose upper bound on worst-case resource demands, this goal implies careful resource allocation may be necessary. This goal is motivated by our expectation that systems are likely to have sufficient resources to be useful for disconnected operation, but that they probably will not have sufficient resources to prefetch everything that applications might want.
3. **Self-tuning.** The policy should not require user intervention or hand tuning. This goal is motivated by the large number of services that a client may host as well as the wide range of service demands and system configurations likely to be encountered.

A potential problem with standard resource management polices – such as LFU or LRU for cache replacement or FIFO or round-robin for CPU or network scheduling – is that these policies reward increasing resource demands with increasing allocations: as a program references more data, it is given more memory; as it spawns more threads or sends more network packets, it gains a larger fraction of those resources. Such approaches provide global allocation of resources that can meet the goal of efficiency (assuming that each application's requests have similar utility.) Such an approach also meets the goal of self-tuning. However, this approach is vulnerable to denial of service attacks.

A second simple approach is to give each service an equal share of resources. But such an approach faces a dilemma: making that fixed amount too large risks denial of service attacks while making it too small thwarts construction of useful services. For example, browser cookies represent requests from untrusted sources to store data on client machines, but limitations on allowed cookie size and number of cookies prevent construction of, say, a "disconnected Hotmail." On the other hand, if cookies large enough to contain a respectable inbox and outbox were made available to all services, a user's disk might fill quickly.

3.1 Popularity-based resource policy

Given these constraints, a resource management system for mobile extensions should attempt to forge a compromise between static allocations that require no knowledge about users or services and dynamic approaches that require unrealistic amounts of knowledge about users or services. Our goal is to construct a dynamic allocation framework that can make reasonable, albeit not perfect, allocation decisions based on information about users or services that can readily be observed by the system and that are not easily manipulated by the extensions. We use service "popularity" as a crude indication of service priority, and allocate resources to services in proportion to their popularities. This approach is based on the intuition that services that users often access are generally more valuable to them than those the they seldom use. It also has the attribute of providing users with better service for those services that they often access.

Our approach is simple: for each resource and each service, the system maintains an exponentially decaying time-average of the number of requests by the user agent to the service. The resource manager for each resource allocates the resource to each service in proportion to the service's time average popularity as a fraction of the popularity of other services. Our resource schedulers are work conserving: if one service uses less than its full share of a resource, the excess is divided among the remaining services in proportion to their scaled popularities.

A key idea in the system is that separate scaled per-service popularities are maintained for each resource, and each resource uses a different timescale for computing its time average popularity. This is because the appropriate definition of "popularity" varies across resources because different resources must be scheduled at different granularities. In particular, "stateless" resources such as CPU can be scheduled on a moment-to-moment basis to satisfy current requests. Conversely, "stateful" resources such as disk not only take longer to move allocations from one service to another but also typically use their state to carry information across time, so disk space may be more valuable if allocations are reasonably stable over time. Thus, the CPU might be scheduled across services according to the momentary popularity of each service, while disk space should be allocated according to the popularity of the service over the last several hours, days, or longer. Other resources — such as network bandwidth, disk bandwidth, and memory space — fall between these extremes.

Although having different time scales for different resources might appear to introduce the need for extensive hand-tuning, we avoid this by choosing each resource's time scale to be proportional to the state associated with the resource or typical occupancy time in the resource for a demand request. For example, for disks, we count the number of bytes delivered by the system to the HTTP user agent and rescale the per-service running popularity averages by multiplying them by $\frac{1}{2}$ each time $diskSize$ bytes are delivered. As our results below indicate, this approach works well across several workloads and several orders of magnitude of disk sizes without changing any parameters.

For network and CPU scheduling, we use the weighted sum of two popularities with each averaged over a different time-scale. The first represents the share of "demand" resources that should be allocated to allow a service to respond to user requests. This time scale should be on the order of the time needed to respond to a user request. We use 10 seconds. The second term represents the share of background CPU and network resources that should be allocated to allow a service to, for example, prefetch and write back data. Since these background actions primarily support disk usage, we use the disk's timescale here so that services are granted "background" network and CPU resources in proportion to the disk space they control. Since we wish to favor demand requests over background requests, we weight the first term much more heavily than the second in computing the total CPU and network resource fractions for each service. In particular, suppose that the scaling interval for the demand term is t_1, that the scaling interval for the background term is t_2, and that we scale the running average by $\frac{1}{2}$ at the end of each interval. If requests arrive at some rate r, then the total raw weight for the demand term is about $t_1 r + \frac{1}{2} t_1 r + \frac{1}{4} t_1 r \ldots \simeq 2 t_1 r$. Similarly, the total raw weight for the background term is about $2 t_2 r$. Therefore, to allow demand requests to dominate background requests during the first seconds after a demand request, we weight the demand term by a factor of $100 \frac{t_2}{t_1}$. During periods of idleness, the second term becomes dominant in roughly 100 seconds.

Limitations. One focus of our evaluation is to determine whether the readily observable metric of popularity provides sufficient indication of user priority to serve as a basis for resource management. To make the analysis tractable, our analysis abstracts some important details.

In particular, our strategy of providing one credit per incoming HTTP request represents a simplistic measure of popularity. For example, one might also track the size of the data fetched or the amount of screen real estate the user is devoting to a page representing a service. Other means will also be required for streaming media.

In addition to these simplifications in these simulations, the algorithm itself has several significant limitations.

First, even if our popularity measures perfectly captured user priority, our resource management algorithm emphasizes simplicity over optimality. It could be enhanced in several ways. For example, one might implement a more complete economic model that gives services coins in proportion to their popularity and that allows "the market" to determine the prices of different resources over different time scales. Applications that have a surplus of one resource could then trade rights to that resource for a different one; or applications could follow application-specific strategies to optimize their resource usage (e.g., "my content is not time critical, so wait until 2AM when BW is cheap to prefetch it.") Developing flexible economies and strategies for competing in them is an open research problem.

Second, our use of the requests from legacy HTTP user agents as a measure of raw popularity makes the system vulnerable to attacks in which legacy client-extension code running at clients (e.g., Java Applets or Javascript) issues requests to the mobile extension proxy in the client's name, thus inflating the apparent popularity of a service. This particular problem could be addressed by having browsers tag each outgoing request with the number of requests issued by a page or its embedded code since the last user interaction with the page; our system would then assign smaller coins to later requests. But this problem illustrates a more fundamental issue: any system that tries to infer priority from user activity provides the opportunity for applications to "game" the system by encouraging activities that will increase resource allocation. We must therefore compromise between simplicity on one hand and precision on the other.

Third, a user's value of a service may not correspond to frequency that the user accesses that service. For example, a user might consider her stock trading service to be her most important service even though she only accesses it once per day to read one page. Although popularity will clearly not capture precise priority, we believe that the heuristic that services a user often uses are likely to be more important than services she seldom uses is a reasonable compromise between simplicity on one hand and precision on the other. Our prototype system provides an "override module" to allow manual adjustment of service priorities if users desire to do so.

3.2 Evaluation

The simulation experiments in this subsection test whether a simple popularity-based policy can meet the three goals – isolation, efficiency, and self-tuning – outlined above.

Our simulator uses as input two traces: Squid [26], which contains 7 days (3/28/00 – 4/03/00) of accesses to the Squid regional cache at NCAR in Boulder, Colorado that serves requests that miss in lower-level Squid caches, and the first seven days from UC Berkeley Home-IP HTTP traces [13]. The simulator uses information in the traces to identify cachable and non-cachable pages as well as stale pages that require reloads. We simulate a separate resource principal for each service (as defined in Section 2) in the trace. We simulate two cache configurations. In the first, we simulate a separate cache at each client in the trace. This corresponds to the environment that might be seen by code that seeks to support mobile clients as well as to improve client performance and availability. In the second, we simulate a proxy cache shared by all clients in the trace. This configuration does not support client mobility, but it may improve service availability or performance.

We first study the resource management algorithms in the context of disk space management by examining three algorithms: (1) traditional *LRU* cache replacement that emphasizes global performance, (2) *Fixed-N*, which supports performance isolation by dividing the cache into N equal parts and allowing each of the the N most recently accessed services to use one part, and (3) *Service Popularity*, which allocates disk spaces in proportion to each service's time-scaled popularity as described above.

A key challenge in studying web services is that as indicated in Section 2, services' prefetching demands, prefetching strategy, and prefetching effectiveness vary widely. It is not practical to simulate application-specific prefetching and adaptation for each of the thousands of services that appear in our trace. The key observation that makes our analysis tractable is that for the purposes of evaluating resource management algorithms, it is not necessary to determine the impact of prefetching on the service that issues the prefetching; one may assume that a service benefits from its own prefetching. What is more relevant is the impact that

one service's prefetching has on *other* services.

So, rather than simulating what benefit a particular service gains from prefetching, we focus instead on the impact that services' resource demands have on other services' performance. We simulate prefetching by a service by fetching sets of dummy data that occupy space but that provide no benefit.

Figure 8 shows the hit rate of the LRU, Fixed-N, and Service Popularity algorithms as we vary per-client cache size (figure (a)) or total cache size (figures (b)) for UCB trace. In this experiment no services prefetch. This experiment thus tests whether the algorithms allocate resources fairly and efficiently when all services are equally aggressive relative to their demand consumption. In such environments, LRU works well because it optimizes global hit rate. Conversely, Fixed-N's performance suffers because it allocates the same amount of space to all services and because the parameter N must be chosen carefully to match the size of the cache. The Service Popularity algorithm is competitive with LRU across a wide range of cache sizes for both workloads and for both the per-client and proxy cache configurations. The results of the Squid traces (not shown) are qualitatively similar. These results suggest two things. First, they indicate that the service popularity algorithm is reasonably efficient: it partitions resources nearly as well as the global LRU algorithm. Second, they provide evidence that the use of time averages proportional to the "natural frequency" of disk residence time supports our goal of developing a self-tuning algorithm.

In Figure 9 we examine what happens when prefetching aggressiveness varies across services. We randomly select 20% of the services and introduce artificial prefetch requests from them. For each demand request, a prefetching service fetches ten objects whose total size is the x-axis value times the size of the demand object. The remainder of the services do not prefetch. The figure plots the performance of the services that do not prefetch. If a system that provides good isolation, the performance of less aggressive services should not by hurt by more aggressive services. In this experiment, when prefetching is restrained, the Popularity and LRU algorithms are competitive. However, as prefetching becomes more aggressive, the performance of non-prefetching sites suffers under LRU, whereas their performance under Popularity-based replacement remain largely unaffected.

Figure 10 evaluates the resource management approach for network bandwidth. We consider three network schedulers: (1) FCFS which services requests in FIFO order, (2) Equal-Fair, which splits bandwidth equally across all services that request bandwidth using start-time fair queuing (SFQ) [11], and (3) Popularity-Fair, which also uses a SFQ scheduler, but which divides bandwidth according to the Popularity-based algorithm described above.

In this simulation, we assume that the bottleneck in the network is the shared link. Note that our base SFQ scheduler is a work-conserving scheduler: if a service is not able to use its full allocation due to a different bottleneck, the algorithm divides the excess among the remaining services in proportion to their priorities.

To introduce prefetching load, we randomly select 20% of the services and introduce artificial prefetch requests from them at the rate specified on the x-axis. For each demand request, a prefetching service fetches ten objects whose total size is the x-axis value times the size of the demand object.

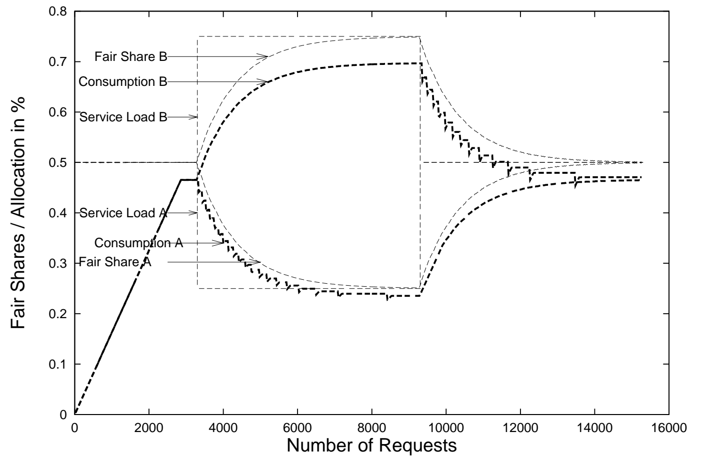


Figure 11: Service load, disk fair share, and disk consumption.

The remainder of the services do not prefetch. The figure plots the performance of the services that do not prefetch. As for the disk space case, we do not assess the effectiveness of prefetching for the services that issue prefetching. Instead, we focus on how excess demand from one service affects other services.

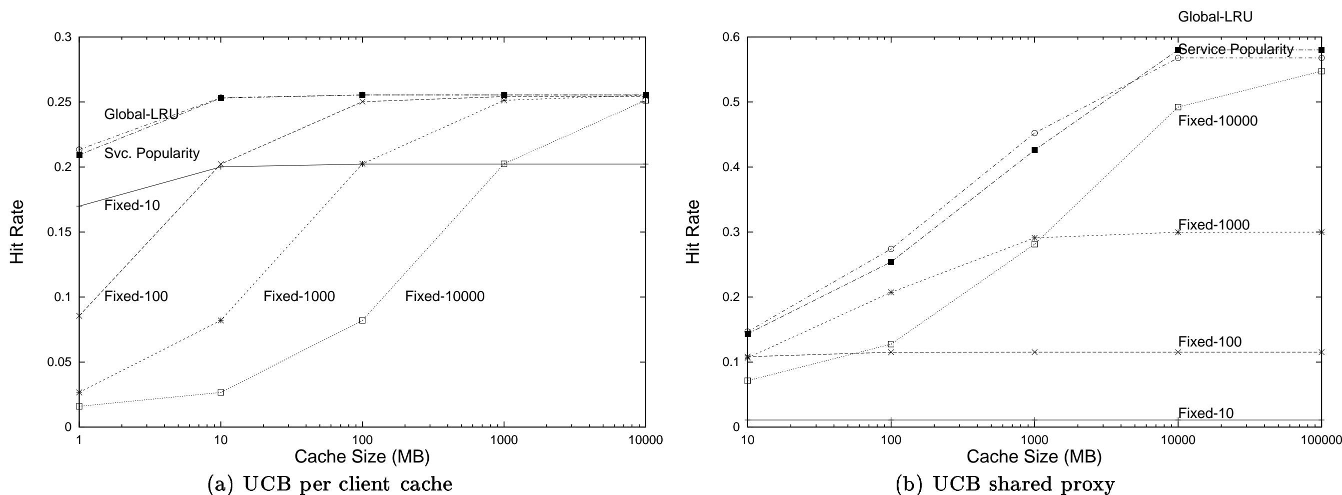
Under FCFS scheduling, prefetching services slow down demand-only services by a factor of 10 and a factor of 2 in the Squid and UCB traces for prefetching rates of 10. In contrast, Equal-Fair is not sensitive to the aggressiveness of the prefetching services. Even though this algorithm does not favor recently accessed services over prefetching services, the fact that only 20% of our services are prefetching and that they prefetch soon after their demand requests finish limits the amount of damage that prefetching inflicts on other services in this experiment. When there is no prefetching, Popularity-Fair is competitive with the FCFS scheduler. When prefetching by aggressive services increases, however, this increase has almost no affect on the less aggressive services.

4. PROTOTYPE SYSTEM

Our prototype implementation is constructed as an HTTP proxy that accepts legacy HTTP requests and by default forwards these requests to legacy HTTP servers. We constructed it using the Java-based Active Names framework [30], which allows services to define a pipeline of programs that will interpret a request. The system provides a *delegation* interface to allow an HTTP reply from a service to specify a service-extension program to handle future requests to that service. The mobile service programs are Java programs that implement an interface fundamentally similar to Java Servelets. Although we use Active Names for our prototype, the resource management approach we describe would also apply to other prefetching or distributed service execution systems.

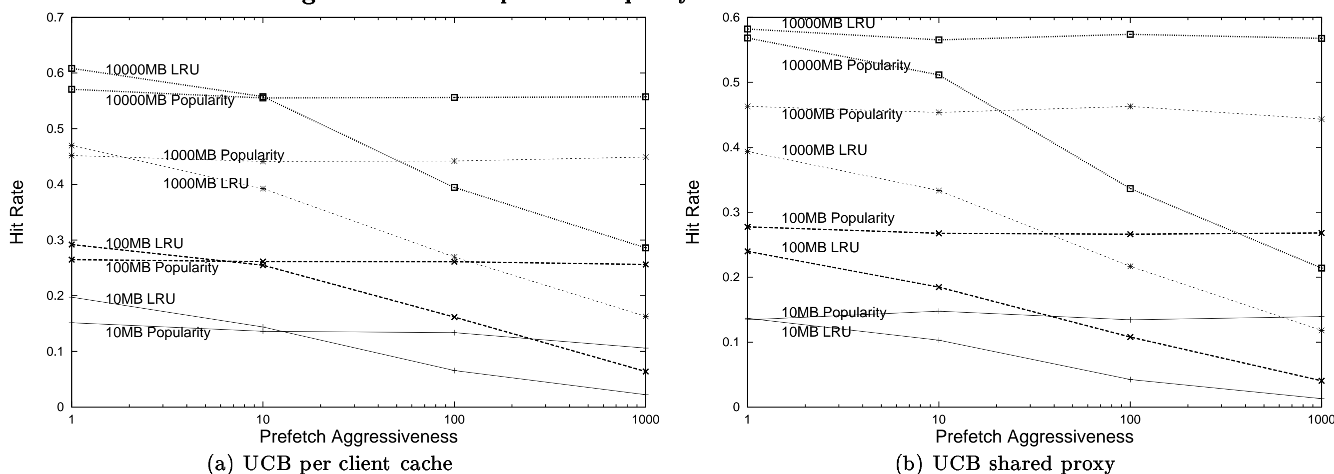
Currently, we have implemented resource management for disk space and network bandwidth, and we are in the process of implementing resource management for CPU cycles and memory space.

Figure 11 shows the popularity-based resource management algorithm in action. We construct two simple extension programs each of which repeatedly writes as much data as it can to disk. We activate an artificial workload that sends two requests per second to the services, initially split-



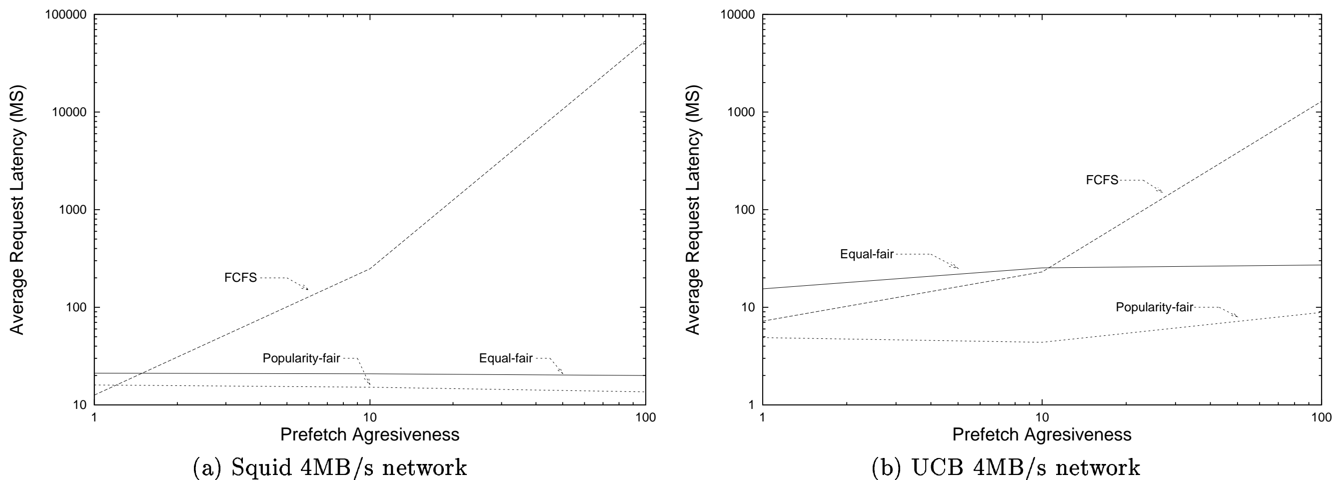
(a) UCB per client cache (b) UCB shared proxy

Figure 8: Cache replacement policy: Cache hit rate v. cache size.



(a) UCB per client cache (b) UCB shared proxy

Figure 9: Cache performance with 20% of sites prefetching.



(a) Squid 4MB/s network (b) UCB 4MB/s network

Figure 10: Network response time v. prefetching aggressiveness.

ting requests evenly between them. As the two services fill up the small (3 MB) disk partition under test, their allocations are equal. Then, when the request distribution changes so that the first service receives three times as many requests as the second, the first's allocation grows at the expense of the second's until their disk allocations are in the ratio of 3:1. Finally, the workload returns to even request rates to the two services, and, over time, so do the disk allocations. Note that the fair share and consumption lag the load because disk scales popularity over time. Also note that the extensions' schedulers keep consumption at about 95% of fair share, yielding a small gap between the two lines.

5. RELATED WORK

File caching [14], replication [28], hoarding [17, 18], message queues [7, 16], and write buffering are standard techniques for coping with disconnection for static file services. These systems have primarily been examined in environments where a small number of programs deliberately installed by a user share resources.

In the context of the WWW, Active channels [1] provide an interface for server-directed hoarding. In addition to being limited to static web pages, active channels require user intervention to enable each service. We speculate that the primary reason for this limitation is the need to give the user control over which servers may use client resources. Similarly, Microsoft Internet Explorer [19] lets users identify groups of pages to hoard, but users must manually select sites and indicate prefetch frequency and hoard depth. AvantGo [2] provides a similar interface where users are responsible for resource management; in this case, each channel reveals a maximum space allocation that the user can consider when subscribing.

These manual approaches have a number of limitations. First, although they may work for a few dozen services, they appear unlikely to scale to the hundreds of services a user may access in a day or a week. Second, the resource limits are device-specific, so users must make different resource management allocations for each device they use to access the services. And, as devices' resources change (e.g., due to changes in network bandwidth), there is no easy way for a user to reprioritize her decisions. Although this paper focuses on caching mobile code, the resource management strategies we describe could also be used to reduce the need for user management of these hoarding interfaces.

A range of mobile code systems have been constructed to improve performance or support mobility. These systems typically have focused on environments where the need for resource management is less severe than for the applications and environment we target. In particular Rover [16] and Odyssey [21] provide mobile code for mobility, but published studies of these systems have focused on environments with small numbers of installed services rather than the emerging WWW service access model in which users access of different services per day.

Commercial content distribution networks are beginning to offer clients the ability to execute code at content distribution nodes using interfaces such as ICAP [29]. Resource management is simplified in this case by the existence of a contractual relationship between code supplier and resource supplier, which both limits the range of behavior that the "untrusted" code may exhibit and which allows the administrator of the system to explicitly specify what fraction of resources should be guaranteed to each service. We speculate that as ICAP scales to large numbers of locations and services, the need for system support for resource management will increase.

Java applets and javascript are similar to the systems we target in that any service may ship code to a client or proxy; however, the code is prevented from accessing disk. This, in turn, reduces the incentive for these applications to use other resources to, for example, aggressively prefetch. Java applets are therefore able to get by without resource management because aggressive resource usage is an uncommon case typically resulting from buggy or malicious code.

Our work shares many goals with market-based resource management system investigated in the D'Agents project [4]. Both systems seek to develop scalable infrastructure to support large collections of untrusted code. Whereas our policy uses an economics analogy to form the basis of a simple policy, the D'Agents project is developing a more flexible resource market approach. Also, we focus on understanding and meeting the requirements of web service workloads rather than supporting a more general agent-based computational infrastructure.

A number of economics-based strategies have been proposed for distributing resources among competing applications in large distributed systems [4, 24]. These systems target more general network topologies than ours and they use secure electronic currency to ration resources.

Adaptive research scheduling is an active research area. However, most proposed approaches are designed for benign environments where applications can be trusted to inform the system of their needs [15, 20] or can be monitored for progress [10, 27]. We treat applications as untrustworthy black boxes and allocate resources based on inferred value from the user rather than stated demand from the applications. The former approach can be more precise and can get better performance in benign environments, but the latter provides safety in environments with aggressive extensions. Noble et. al [21] emphasize *agility*, the speed at which applications and allocations respond to changing resource conditions, as a metric of dynamic resource schedulers. We argue that for stateful resources such as memory and disk, agility should be restrained to match the rate at which the resource may usefully be transferred between applications.

6. CONCLUSIONS

In this paper, we first examine the resource demands for environments that wish to support mobile code to enable disconnected operation. We find that scalability is a key challenge: dozens, hundreds, or thousands of extensions may compete for client or proxy resources. Furthermore, these services may be more aggressive than current services because they use resources in the common case to guard against the uncommon case of disconnection. Although these demands can be large, the falling costs of resources as well as the flexible demands that mobile code may exhibit suggest that – if carefully managed – clients may have sufficient resources to support significant disconnected operation. In order to avoid manual tuning, we examine a simple algorithm for resource management that appears to provide both good isolation and good efficiency.

7. REFERENCES

[1] Active channel technology overview. `http://msdn.microsoft.com/workshop/delivery/channel/overview/overview.a%sp`, 1999.

[2] http://www.avantgo.com, February 2001.

[3] G. Back, W. H. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, Oct 2000.

[4] J. Bredin, D. Kotz, and D. Rus. Market-based Resource Control for Mobile Agents. In *Autonomous Agents*, May 1998.

[5] P. Cao, J. Zhang, and Kevin Beach. Active Cache: Caching Dynamic Contents on the Web. In *Proceedings of Middleware 98*, 1998.

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

[7] IBM Corporation. Mqseries: An introduction to messaging and queueing. Technical Report GC33-0805-01, IBM Corporation, July 1995. `ftp://ftp.software.ibm.com/software/mqseries/pdf/horaa101.pdf`.

[8] G. Czajkowski and T. von Eicken. JRes: A Resource Accounting Interface for Java. In *Proceedings of 1998 ACM OOPSLA Conference*, October 1998.

[9] M. Dahlin, B. Chandra, L. Gao, A. Khoja, A. Nayate, A. Razzaq, and A. Sewani. Using Mobile Extensions to Support Disconnected Services. Technical Report TR-2000-20, University of Texas at Austin Department of Computer Sciences, June 2000.

[10] J. Douceur and W. Bolosky. Progress-based Regulation of Low-importance Processes. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, pages 247–258, December 1999.

[11] P. Goyal, H. Vin, and H. Cheng. Start-time Fair Queuing: A Scheduling Algorithm for Integrated Services Packet Switching Networks. In *Proceedings of the ACM SIGCOMM '96 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 157–168, August 1996.

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

[13] S. Gribble and E. Brewer. System Design Issues for Internet Middleware Services: Deductions from a Large Client Trace. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997.

[14] 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.

[15] M. Jones, D. Rosu, and M. Rosu. CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, October 1997.

[16] A. Joseph, A. deLespinasse, J. Tauber, D. Gifford, and M. Kaashoek. Rover: A Toolkit for Mobile Information Access. In *Proceedings of the Fifteenth ACMSymposium on Operating Systems Principles*, December 1995.

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

[18] G. Kuenning and G. Popek. Automated Hoarding for Mobile Computers. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 264–275, October 1997.

[19] Microsoft internet explorer 5. http://www.microsoft.com/windows/ie/default.htm, 2000.

[20] J. Nieh and M. Lam. The Design, Implementation, and Evaluation of SMART: A Scheduler for Multimedia Applications. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, October 1997.

[21] B. Noble, M. Satyanarayanan, D. Narayanan, J. Tilton, J. Flinn, and K. Walker. Agile Application-Aware Adaptation for Mobility. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, October 1997.

[22] A. Odlyzko. The history of communications and its implications for the interne. `http://www.research.att.com/~amo/`, June 2000.

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

[24] Z. Qin, W. Wang, F. Wu, T. Lo, and P. Aoki. Mariposa: A Wide-Area Distributed Database System. *VLDB Journal*, 5(1):48–63, January 1996.

[25] S. Savage, A. Collins, E. Hoffman, J. Snell, and T. Anderson. The End-to-end Effects of Internet Path Selection. In *Proceedings of the ACM SIGCOMM '99 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 289–299, September 1999.

[26] Squid sanitized access logs. ftp://ftp.ircache.net/Traces/, April 2000.

[27] D. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole. A Feedback-driven Proportional Allocator for Real-Rate Scheduling. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, January 1999.

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

[29] The ICAP Protocol Group. Icap the internet content adaptation protocol. Technical Report draft-opes-icap-00.txt, IETF, December 2000.

[30] A. Vahdat, M. Dahlin, T. Anderson, and A. Aggarwal. Active Naming: Flexible Location and Transport of Wide-Area Resources. In *Proceedings of the Second USENIX Symposium on Internet Technologies and Systems*, October 1999.

[31] A. Wolman, G. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. Levy. On the scale and performance of cooperative web proxy caching. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, December 1999.

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