Using Mobile Extensions to Support Disconnected Services*

Mike Dahlin, Bharat Chandra, Lei Gao, Amjad-Ali Khoja, Amol Nayate, Asim Razzaq, Anil Sewani
Department of Computer Sciences
The University of Texas at Austin

DRAFT April 29, 2000– please check
http://www.cs.utexas.edu/users/dahlin/papers/
for the most current version

Abstract

This paper examines the design and implementation of *mobile extensions*, a distributed operating system abstraction for supporting disconnected access to dynamic distributed services. The goal of mobile extensions is to make it as easy for service providers to deploy services that make use of caching, hoarding, asynchronous messaging, and application-level adaptation to cope with mobility, network failures, and server failures. We identify resource management as a crucial problem in this environment and develop a novel popularity-based resource management policy and demonstrate that under web service workloads it allocates resources nearly as efficiently as traditional schedulers, while under workloads with more aggressive resource users, it provides much stronger performance isolation. Overall, we find that for the four web service workloads we study, mobile extensions can reduce failures by as much as a factor of 5.9 to a factor of 16.7 for those applications able to provide tolerable service when disconnected.

1 Introduction

This paper examines the design and implementation[1] of *mobile extensions*, a distributed operating system abstraction for supporting disconnected access to dynamic distributed services. Previous work has shown how to support disconnected access to static data [2, 19, 21, 31]. However, many modern services dynamically generate large amounts of uncachable data [34]. For example, HTTP services can extend the default GET/PUT semantics to run arbitrary programs at the server in response to user requests [6]. Unfortunately, providing dynamic services using such *server extensions* inherently limits system performance, mobility, and robustness to network failures.

In a previous study we demonstrated how mobile, location-independent extensions could significantly improve performance for clients accessing dynamic services [33]. This paper focuses on using mobile extensions to address the problem of disconnected operation. Enabling clients to continue to access dynamic services during periods of disconnection is crucial both to support mobile clients, where disconnection is deliberate, and fixed clients, where failures and overloads at network and servers might cause service interruptions. Whereas highly available systems may seek to have “five nines” of availability (99.999% uptime — about 5 minutes of downtime per year), the Internet network layer provides only about two nines of host-to-host connection availability (99% uptime — about 14 minutes of unavailability per day.) For example Paxson found that “major routing pathologies” thwart IP routing between a given pair of hosts 1.5% to 3.4% of the time [24], and recent (March 13-19, 2000) measurements by keynote.com from clients in 25 cities viewing pages from 40 popular HTTP servers found a median end-to-end failure rate of 1.63% [18]. Such failure rates at the network and at servers make it difficult to deploy mission-critical dynamic services under a server-extension architecture because such architectures do not afford end-to-end strategies.

When network connections are slow or unreliable, many services can operate in a *degraded* mode by using a combination of general techniques (such as caching [15], prefetching/hoarding [19, 21], write buffering, and asynchronous messaging via persistent message queues [7, 17]) and application-specific adaptation [23]. Unfortunately, current implementations of the general techniques focus on traditional client-server relationships where a set applications are to be installed at a well defined set of satellite sites. Thus, using these techniques generally requires installing operating system patches, middleware services, or client applications. The goal of mobile extensions is to make it easy for service providers to deploy and for users to access services that support disconnected operation just as HTTP makes it easy to deploy and access server-extension-based services.

Mobile code is not new. For example, Javascript and Java Applets allow servers to ship code to browsers, Smart Clients [35] allows servers to ship code to caches, Active Caches [5] allow servers to ship code to proxies, Active Networks [30] allow network infrastructures to be pro-

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

[1] Our mobile extension framework and example applications as well as the simulators and traces used in this paper are available at http://www.cs.utexas.edu/users/dahlin/osdi-review.

grammed, and agents [12, 20, 25] allow clients and servers to inject code into a distributed infrastructure. This paper makes three contributions towards understanding how to use mobile extensions to support disconnected operation for distributed services.

First, our mobile extension system provides a novel combination of three features that make it particularly suitable for supporting disconnected access to dynamic services: (i) rather than simply support arbitrary programmability, the system retains HTTP's successful approach of providing simple default GET/PUT behavior with the ability to add extensions when and where needed; (ii) the system uses location independence to simplify software engineering, improve security, and to facilitate incremental deployment; and (iii) the system allows services to take full control of their caching, hoarding, and messaging protocols.

Second, we develop a resource management framework that (i) provides dynamic 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 decisions automatically without relying on user input or directions from untrusted extensions. To accomplish these goals, the system infers priority from "popularity" based on request patterns, and it considers popularity on different timescales for different resources according to the following rule: the more state associated with a resource, the longer the timescale across which popularity should be considered. For example, "stateful" resources such as disk must be scheduled over longer time periods than "stateless" resources such as CPU. We evaluate popularity-based resource management via a trace-based study and conclude that it provides reasonable global performance while protecting the system from aggressive extensions, and we find that averaging popularity over timescales proportional to a resource's state appears to work well.

Our third contribution is to quantify the robustness gains available to Internet services as a class and to several specific case study applications. Using trace-driven simulations, we find that for Internet services as a class, mobile extensions can improve availability by over an order of magnitude by transforming network failures into degraded-mode operations. Note that the benefits of degraded-mode operations vary across services: some require network connectivity to function and will gain no benefit, some can provide indistinguishable service regardless of the network state, and many will fall between these extremes.

We have constructed a Java-based mobile extension prototype that provides backwards compatibility with HTTP, that allows mobile extensions to run at clients, proxies, or servers, and that enforces security and resource restrictions on mobile extensions. Our initial applications include an e-commerce service that hoards catalog entries and queues orders, a prototype hospital laboratory order service that transfers requests from doctors to technicians and results back to doctors, and a set of client-specified hoarding and QRPC-based extensions for enhancing disconnected access to legacy HTTP services. Measurements of our system under synthetic workloads show that it can successfully hide the cost of downloading and installing extension code by taking advantage of extensions' location independence.

The rest of this paper proceeds as follows. In Sections 2 through 4 we discuss the design and implementation of the system: its goals, programming model, and its resource management framework. Section 5 provides our experimental evaluation. Section 6 discusses related work, and Section 7 summarizes our conclusions and discusses future directions.

2 Design goals

The effectiveness of a mobile extension architecture depends on how it meets three goals: extensibility with simple default semantics, location independence, and flexible and automatic resource management.

Extensibility with simple default semantics. Requests to services should have simple default semantics that do not require explicit definition of mobile service programs to handle them, but the infrastructure should allow users and services to specify extensions that will override some or all aspects of the default semantics for specified subsets of requests. This approach has been highly successful for deploying distributed services under HTTP. HTTP provides a basic GET interface that provides simple default behavior of reading a file; at the same time, HTTP allows servers to arbitrarily redefine the semantics of GET (and other methods) for specific subsets of requests so that GETs may be used to activate arbitrary RPC calls. In contrast with providing a raw RPC interface, this combination of widely-useful default behavior and extensibility allows complex services to be prototyped, constructed, deployed, and updated easily.

A mobile extension framework should balance extensibility and simple default semantics. Ideally, a service should be able to (i) use default semantics only, (ii) completely override the default semantics for a subset of requests and use default semantics for the others, (iii) override some aspects of default semantics for some or all requests while retaining some aspects of the default behavior, or (iv) redefine all behavior for all requests to that service.

Location independence. Extensions should be defined using a single code base, allowing the same code to run at a client, at a proxy cache, at a server proxy, or at a server. The primary advantage of this approach is that extensions can choose to run at the appropriate point in a network to meet the requirements of their particular application. For example, an extension designed to allow a mobile client to access a mail service when the client has no available network connection must run at the client to be of use, whereas an extension designed to allow doctors and lab technicians to exchange orders and results in a hospital when the hospitals external connection is down should run at a shared proxy within the hospital.

Location independence has several additional benefits: First, location independence facilitates incremental deployment because it simplifies software engineering by allowing services to be used by clients that support the framework and those that do not, while avoiding the need to maintain two code bases. Systems should use the same program for the case when code is shipped to clients and the case where the code runs at the server.

Second, location independence can improve performance. Running the same code at clients and servers allows systems to hide the start-up cost of accessing a new service: Initially a client can access the extension at the server, but once the extension has been installed at the client the client can switch to the local copy for improved robustness and performance. This reduces the incremental cost of deploying mobile-extension-based services by avoiding the need to wait several seconds in the common case of accessing a service for the first time in order to improve the uncommon case of disconnected operation.

Flexible and automatic resource management. Clients and client proxies will run large collections of heterogeneous extensions, and the system should automatically assign each an appropriate amount of resources. The mobile extension environment poses two challenges to resource management. First, techniques for supporting disconnected operation, such as hoarding, can dramatically increase a service's resource demands: it is one thing to cache the pages one has visited at a site; it may be another matter entirely to hoard all of the pages one *might* visit. Second, this environment must accommodate large numbers of untrusted extensions. Because code is untrusted, policies that reward increasing resource usage with increasing allocations (e.g., LRU or MFU cache replacement) or that explicitly ask applications what their resource needs are [22, 23, 28] are not appropriate. And, because extensions are general, there is no obvious progress metric [10, 29] that can be tracked to allocate resources by the utility yielded by each extension.

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 as the system functions and that are not easily influenced by untrusted code's actions.

3 Programming model

Our prototype implementation of mobile extensions 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 [33], which allows services to define a pipeline of programs that will interpret a request. Both "default protocols" such as HTTP and "extension" are defined in terms of these service programs. Each service program is a Java program that provides a method called *Eval()* with three arguments: an *ActiveName* that identifies the service and encodes the request to be interpreted by that service, an *InputStream* of data to that service, and a Vector of *AfterMethods*

- The ActiveName consists of two components: the URL of the code representing the extension service program and a string. In Active Names terminology, the URL identifies a Namespace program and the string represents a name to be interpreted by that Namespace program.
- AfterMethods lists services (represented as Active Names) for the request to visit after the current service. The AfterMethods list allows the system to implement a continuation-passing style of programming where each namespace can insert remaining work later on the AfterMethods list.
- The InputStream is used to transport bulk input to a service; the service, in turn, produces an InputStream that it passes to the next service to be run. For efficiency, a service that does not touch the contents of its InputStream may pass the handle of that InputStream to the next service to avoid extra data copies.

Thus, as Figure 1-a illustrates for the case of a default HTTP request, a request visits a series of extensions that form a pipeline, with each extension selecting the next extension to run, processing its input stream, and transporting the result to the next service.

The rest of this subsection describes how the system provide extensibility, location independence, and mechanisms for security and resource management.

3.1 Extensibility

The AfterMethods list provides the key abstraction for extensibility. Before passing control to the next program on the AfterMethods list, a program may modify that list by inserting, deleting, or changing elements on the list and thereby modify the pipeline of services and extensions that will handle the request.

In particular, all incoming requests are assigned a default set of services to visit including standard services such as HTTP-cache and fetch-legacy-HTTP-server- as well as a ServerCust module. This customization module provides an opportunity for the server to modify the standard AfterMethod list to override some or all of the standard processing for a request. ServerCust is a trusted module that

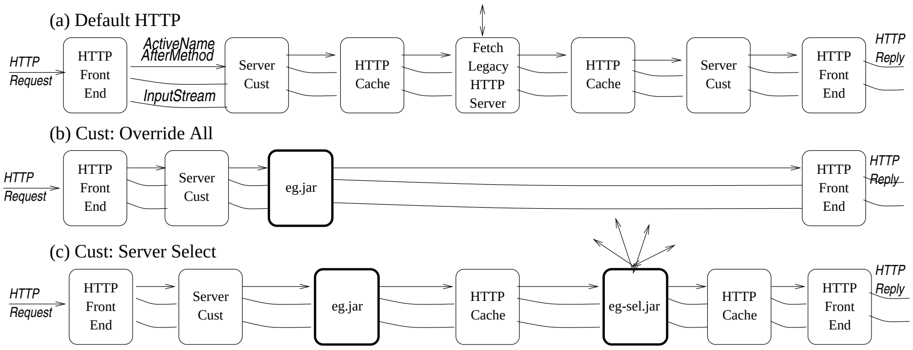


Figure 1: Example service/extension program pipeline of (a) standard HTTP protocol, (b) completely overridden protocol, (c) partially-overridden protocol.

maintains a *delegation table* of mappings from URL prefixes to programs that should be executed to customize requests to those URLs, and it provides an interface that allows each service to update its mapping (but, obviously, not the mappings of other services) by piggy-backing delegation directives on HTTP reply headers. For example, the HTTP service www.exmpl.com might specify that the program http://www.exmpl.com/eg.jar be inserted into the pipeline for all requests to www.exmpl.com/*. As Figure 1-b and 1-c illustrate, incoming requests to www.exmpl.com would visit HTTP Front End and ServerCust, which would next send the request to the program eg.jar. When the eg.jar program runs it could, for instance, completely override the standard HTTP protocol with its own caching, hoarding, and network fetch protocol (Figure 1-b) or it could partially modify the AfterMethod list to replace the standard HTTP-network-fetch module with a module that does automatic fail-over across several exmpl.com mirror servers [35] (Figure 1-c).

3.2 Location independence

The extension programs are location independent and can run on any node that provides the virtual machine interface. In this paper we focus on two configurations. In the first, origin servers and end-clients support mobile extensions. In the second, origin servers, end-clients, and shared client-proxies support mobile extensions.

By default, each program executes the next program in the pipeline on the local machine (which may be the client, proxy, or server), but each program is free to explicitly invoke the same program or a different one on a different node. The choice of when to execute locally and when to jump to a different machine is extension-specific. More sophisticated topologies such as replicated servers or third-party hosting services such as Akamai are possible, but rather than try to provide a general topology-discovery mechanism, we allow each service to provide whatever topology-discovery mechanism is appropriate for that service as an extension program. Although solving the general topology discovery problem is difficult, for the system configurations discussed in this paper, topology discovery simply consists of examining the AfterMethod stack to place computations on the local client machine or at the remote client proxy.

4 Virtual machine and resource management

Service programs run on a virtual machine that provides security, resource management, and local/remote method invocation among service programs. We use the Java-2 security system to associate each downloaded set of code with a separate codebase and use the codebase associated with code both to restrict what memory and disk state it may access and to identify the resource principal for disk and network requests, CPU scheduling, and memory allocation.

Our security model is oriented towards isolating untrusted namespace programs from one another and from the underlying machine, both for security and to limit resource consumption. When untrusted code in the form of remote namespaces runs on the Active name system, we need to dynamically give it permissions to access its fair share of resources. In Java2, there is a central java.Security.Policy object that dictates the set of permissions for every codebase. But, it requires that all such permissions be provided prior to execution. ActiveNames has enhanced the current Java security architecture by giving permissions to classes dynamically when they are loaded from an untrusted remote site. We achieve this by overloading the central policy object into an ActiveName Policy object, which assigns to every namespace a unique permission to identify itself. When accessing any resources or security-sensitive information, a namespace has to identify itself with its Active Name. The virtual machine then uses standard Java-2 stack inspection verifies that the caller has permission to use the offered name before allowing the request to proceed.

Given the challenges discussed in Section 2, our goal is to construct a dynamic allocation framework that can make reasonable allocation decisions based on information about users or services that can readily be observed by the system 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.

Our implementation consists of four main components: an observation module that tracks service popularity, a scaling module that translates raw popularity counts into per-resource per-service allocations, a manual override module that allows users to override the algorithm's decisions, and per-resource schedulers. These pieces are described below.

4.1 Observation module

The observation module tracks system activity to infer the priority that users give to different services. This popularity tracking is implemented by attaching a "coin" (implemented as a protected class) to each HTTP request that arrives at the RawHTTP module from a client authorized to make requests to that proxy. As the request visits different services within the system, the observation module credits a fraction of the coin to each service visited. The system uses heuristics to ensure that all services visited by a request receive approximately equal fractions of the request's coin, and the system ensures that the sum of the fractions allocated to services is less than or equal to 1.0. In addition, the system allows only trusted modules to create new "coins;" untrusted extensions can only pass coins to one another or split coins.

A limitation of the prototype is that our interface to legacy HTTP clients makes it 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 problem could be addressed by having browsers tag each outgoing request with the number of requests issued by a page or its code since the last user interaction with the page; our system would then assign smaller coins to later requests.

A second limitation of our prototype is that our strategy of providing one coin 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.

4.2 Scaling module

Whereas the observation module produces "raw" counts of popularity (how many requests visit each service and which services each request visits), the scaling module converts these raw counts into per-resource, per-service allocations. As noted above, our intuition is that we can infer priority from popularity. However, the appropriate definition of "popularity" varies across resources because different resources must be scheduled on different time scales. "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 resources 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 should 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 perhaps the last several hours or days. Other resources — such as network bandwidth, disk bandwidth, and memory space — may fall between these extremes.

The scaling module provides a general interface to assess each service's popularity on the different time scales appropriate to different resources. Each resource registers with the scaling module by specifying an *epochLength* and *scalingFraction*. For each resource, the scaling layer maintains per-service resource containers [3], and the fraction of resource R that the scheduling layer should give to service S is *frac[R,S]* = $\frac{container[R,S]}{containerTot[R]}$, where $containerTot[R]$ equals the sum of all services' containers for a resource (i.e., $containerTot[R] = \sum container[R,*]$.)

Whenever the popularity layer credits a service with a fraction of a coin, the resource containers for that service at each resource are increased by the specified amount, as are the $containerTot$ values for each resource.

Conceptually, the array of per-service containers for each resource is multiplied by *scalingRate* every *epochLength* interval. For efficiency, we store the last update time with each container and rescale the value if necessary when it is read or written. If the *scalingFraction* and *epochLength* are powers of two, this operation can be accomplished with a few addition, subtraction, and shift operations per read or update.

For each resource, we choose an *epochLength* proportional to the state associated with the resource or the typical occupancy time in the resource for a demand request. For example, for disks, we count the number of bytes delivered to HTTP Front End and increment the disk epoch number once per $diskSize$ bytes seen. For networks, we use *epochLength = 2 seconds* to represent a generous network round trip time.

4.3 Override module

Although we do not rely on manual resource allocation, there are cases where human direction is desirable. For example, a user may wish to tell her proxy to give high priority to requests to her online trading service even though she uses it only occasionally. Also, shared replication services such as Rent-A-Server [32], Akamai, or Sandpiper may allocate resources across services according to contractual agreements rather than the popularity of the services.

For such cases, our system provides an override module that allows resource allocations to be manually set. Note that none of the experiments discussed in this paper use the override module.

4.4 Resource schedulers

Our virtual machine provides proportional-share resource schedulers for each critical resource. The mechanisms used to track and restrict system resource utilization in our Java-based system are similar to those used in JRes [9]. Our current implementation provides a Start-time Fair Queuing (SFQ) scheduler for network bandwidth [13] and a proportional-share disk space allocator (described below). We are in the process of implementing a proportional-share CPU scheduler and memory allocator. To support application-level adaptation, the system's resource scheduler decides what fraction of each resource to give to each service, while leaving it to the services how best to make use of their allocations. To facilitate adaptation, the resource manager signals extensions when their allocations cross specified thresholds.

The proportional-share schedulers for stateless resources, such as CPU and network, enforce resource limits by scheduling requests and threads using SFQ.

The proportional share scheduler partitions disk space across extensions according to their scaled popularities. If some extensions do not used their full allotment, the remaining space is divided proportionally among the other extensions. The disk space scheduler accomplishes this by maintaining a per-service *price*, which is the scaled popularity of the service divided by the disk space held by the service. When a service requests more space, the system selects the service with the lowest current page price as a victim and signals that extension to release disk space. For efficiency, these actions are decoupled via a reserve buffer from which new pages can be allocated immediately and victims selected lazily as allocation demands and priorities change. When an extension's allocation shrinks below a warning threshold relative to the extension's allocated space, the scheduler warns the extension to reduce its usage with a signal. If the extension fails to reduce its usage before its allocation falls to the point where the extension exceeds its maximum allocation, the system kills the extension. To maintain the abstraction of persistent storage in such cases, the disk system provides an interface for extensions to specify an ”forwarding address” and to mark on-disk objects that should be forwarded. In the event of the extension's demise, the system promises to forward this marked state, although it may discard state after asking for the user's permission if the forwarding address is persistently unreachable. This forwarding procedure may sometimes prevent the system from reclaiming space when it would like to do so, but we feel that the benefits of true persistent state are worth this limitation.

4.5 Utility libraries

Using the low-level resources provided by the resource managers, extension services implement higher level abstractions such as caching, hoarding, write buffering, or asynchronous messaging by making use of common libraries the system provides or by implementing custom versions of these abstractions [11]. We examine several example applications in the next session. A systematic discussion of the techniques that applications may use to cope with disconnection is beyond the scope of this paper.

5 Evaluation

We first investigate some basic properties of our implementation. We then examine properties of our resource management and robustness policies and implementations.

5.1 Location independence

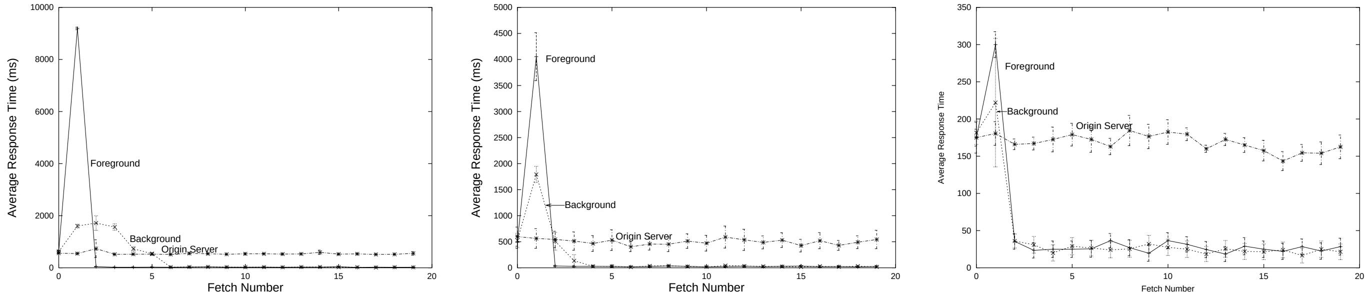
A disadvantage of implementing services as mobile code is that startup time may increase: this approach may hurt performance in the common case of first accessing a service to help in the uncommon case of network failures. Unfortunately, users confronted with a long “applet loading” message when they first access a service may go elsewhere before they have a chance to benefit from the downloaded code.

As noted above, location independence can hide the cost of installing new services at clients. In particular, when our ServerCust module delegates a service to an extension, by default it downloads and installs the extension program in the background while continuing to send requests to the original server. Once the extension has been installed, the ServerCust module sends requests to it instead. The delegation interface allows servers to specify foreground loading if needed.

Figure 2 shows the impact of background loading. In this experiment, a client issues 20 requests to a service with a 1 second delay between each request. Because we are interested in the cost of loading the service and not the service itself, we examine a simple service that dynamically generates a small (100 byte) page. The Java jar file containing this program is 1790 bytes, but we expand it to 22031 bytes by adding some unnecessary functions.

Our client machine has a 366 MHz Pentium-II processor and 128 MB of memory, and it runs JDK-1.2.2 under Microsoft Windows 98. The graphs show three cases for network connectivity – the client is connected to the network via a modem that reports a 26.4 Kbit/s connection, a 128 Kbit/s ISDN, and a 10 Mbit/s Ethernet. We repeat each experiment at least 10 times and show the 90% confidence intervals in the figure.

In each graph, the x-axis shows the request number and the y-axis shows the response time of that request. The lines show three cases: *origin server* where all requests are sent to the origin server, *foreground* where the reply



(a) Modem

(b) ISDN

(c) Ethernet

Figure 2: Foreground v. background loading of extensions.

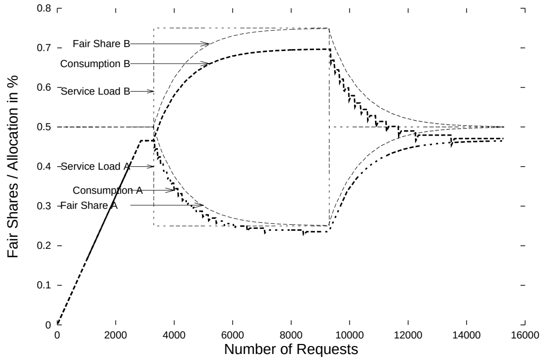


Figure 3: Service load, disk fair share, and disk consumption.

to the first request delegates future requests to a mobile extension at the client and where the second request cannot complete until the extension is installed, and *background* where requests after the first request but before the service replica has been downloaded are serviced by the remote replica and where subsequent requests are serviced locally. As the data in the figure show, background loading improves worst case performance by factors of 5.3, 2.3, and 1.4 compared to foreground loading, although it does not completely eliminate, the installation cost of mobile extensions.

The single code base and location independent code models used by mobile extensions are useful for supporting this optimization. The same code runs at the server and client, which makes switching from one replica to the other simple. It also avoids the need to write the service twice and maintain two versions of the code.

5.2 Resource management mechanism

Figure 3 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 splitting 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.

5.3 Popularity-based policy

This simulation experiment tests two hypotheses about the performance of per-service popularity-based resource allocation policies relative to that of traditional allocation algorithms that optimize for global performance without considering performance isolation. First, we hypothesize that under benign workloads — where services use only the resources needed to satisfy on-demand requests from users — per-service popularity-based resource allocation can provide performance competitive with traditional allocation. Second, we hypothesize that under workloads where some services aggressively use resources, per-service popularity-based resource allocation prevents aggressive extensions from hurting global performance.

We study this problem in the context of cache replacement by examining three algorithms: (1) traditional *LRU* 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 in Section 4. We assume that the system tracks the number of bytes delivered by each service to end users and that it rescales popularity for the disk resource by multiplying each service's accumulated value by 0.5 when the total number of bytes delivered to users since the last rescale exceeds (1.0 * the size of the disk). This approach relates the period of time over which to scale popularity to the amount of system state and thus allows us to avoid changing "magic numbers" for different disk sizes.

Our simulator uses as input two traces: Squid [1], 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 [14]. 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 proxy cache shared by all clients in the trace.

Figure 4 shows the hit rate of these algorithms as total cache size varies. As hypothesized, the Service Popularity algorithm is competitive with the Global LRU algorithm across the range of cache sizes studied: Service Popularity's performance is slightly worse for UCB and slightly better for squid. 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.

Next, we randomly select 20% of the sites and introduce artificial prefetch requests from them. In particular, each time such a site references an object, we generate 10 requests for additional objects whose total size is $prefetch\_agressiveness$ times the size of the original request. Thus, when $prefetch\_aggressivenes = 10$, prefetching services prefetch 10 times as much data as they use. Note that because prefetches are not in response to user requests and their results are not delivered to users, prefetches do not increase a service's popularity or priority under the Service Popularity algorithm. Figure 5 shows the performance of the *non*-prefetching services as we vary the aggressiveness of the prefetching services. 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, while their performance under Popularity-based replacement is mostly unaffected.

5.4 Internet service robustness

This experiment examines potential effectiveness of using downloaded code to improve robustness of Internet services by transforming *failed sessions* that are interrupted by network disconnections into *degraded sessions* that are served by downloaded mobile extensions. Clearly, the relative advantage of degraded sessions over failed sessions will vary from service to service: some services can provide full service while disconnected, others can provide tolerable service across short disconnections, and still others require continuous on-line communication with a remote site to be effective. This experiment does not attempt to quantify the benefit of degraded service over failed service; instead it seeks to quantify how often services that do support mobile extensions can expect to improve their robustness to network disconnections.

In theory, allowing services to ship code to clients should allow services to significantly improve their robustness. In practice, three factors may limit this effect. First, we assume that clients begin to download extension code the first time they access a service in the trace, so sessions soon after the first access may be unprotected and suffer *capacity misses.* Second, systems may have to evict state or code from one service to provide resources for others causing later *compulsory misses* for that service. Third, due to *installation time*, if a failure occurs, soon after an earlier request in which the client started to load an extension, the client may not have time to install the service's extension or the extension may not have time to download the state needed to mask failures.

Workload	Date	NClients	NServers	Sessions
Squid-P	3/28/00 – 4/03/00	1	131193	1557875
Squid-C	3/28/00	107	52526	403235
BU-P	1/17/95 – 5/17/95	1	4614	56789
BU-C	1/17/95 – 5/17/95	33	4614	68949

Table 1: Web access trace parameters.

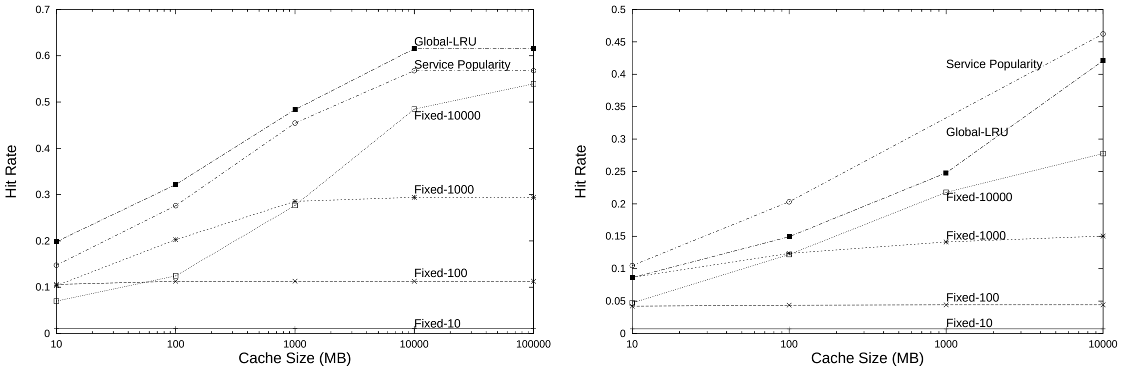
Workload. Our simulation study uses two types of traces: a set of web service access logs that we use directly as reference traces and Internet connectivity failure measurements from which we a generate a synthetic failure workload.

Table 1 summarizes key parameters for our access pattern traces. We examine both the squid trace described earlier and a four-month trace taken at clients at Boston University [8]. This trace is old, but it includes client cache hits, and the client ID mappings are not changed over the trace period. We examine both traces from the point of view of a proxy shared by all clients in the trace (Squid-P and BU-P) and from the point of view of individual client machines (Squid-C and BU-C) with no shared proxy. Because the squid traces change the client-anonymization mapping daily, we only look at the first day of the Squid-C trace.

For our simulations, we post-process the traces to group individual accesses into *sessions.* We define a session as a set of accesses from a client (-C traces) or proxy (-P traces) to a single server in which the maximum gap between successive requests is 60 seconds.

Table 2 summarizes key parameters of our failure model based on Paxson's measurements of connectivity among a collection of 37 sites in 1994 and 1995 [24]. Failures in this dataset are of two types: *temporary outages* that resolve themselves during the course of a 450-second traceroute session and *persistent failures* that continue beyond the end of a traceroute session.

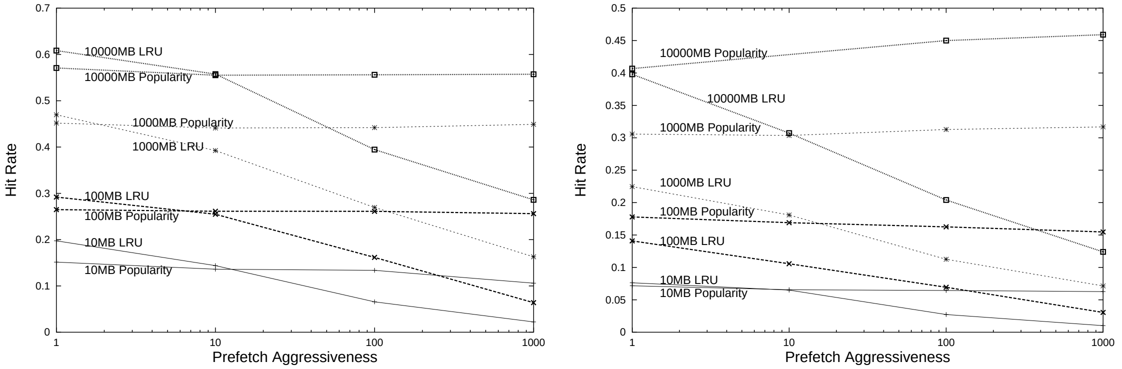
Our simulation models the connection between a client and server with three states: *Up* (the network is up), *Temp* (the network is encountering a *temporary outage*), and *Perst* (the network is encountering a *persistent failure*). For each client-server connection, we choose an initial state according to the steady-state probabilities — P_{Up}, P_{Temp}, and P_{Perst} — which we set based on Paxson's values for temporary failures and our analysis of the trace's persistent failures. Note that Paxson remarks that the tracing method-



(a) UCB

(b) Squid

Figure 4: Cache replacement policy: Cache hit rate v. cache size (*simulation results*).



(a) UCB

(b) Squid

Figure 5: Cache performance with 20% of sites prefetching (*simulation results*).

		Temporary Failures			Persistent Failures		
Model	P_{Up}	P_{Temp}	Duration	$P_{Up \rightarrow Temp}$	P_{Perst}	Duration	$P_{Up \rightarrow Perst}$
Fail-S	.982	.015	30 + exponential(40)	2.1×10^{-4}	.003	500 with probability 0.95 30000 with probability 0.05	1.5×10^{-6}
Fail-L	.982	.015	30 + exponential(40)	2.1×10^{-4}	.003	3600 with probability 0.95 111000 with probability 0.05	3.3×10^{-7}

Table 2: All times in seconds from Paxson [24] and rounded to 2 significant digits.

ology may under-sample during times of network failures, so our model may underestimate failure rates.

Paxson's measurements allow him to characterize the expected duration of temporary outages with precision: for temporary outages not ascribed to router loops, durations in one trace were modeled by a constant 30 seconds plus an exponentially distributed random variable with mean of about 40 seconds in one trace. For simplicity, we assume that temporary failures arrive at exponentially-distributed intervals with mean $1/P_{Up \rightarrow Temp}$ calculated to yield the specified P_{Temp} for the given failure duration.

Unfortunately, Paxson's data provide less precision for modeling persistent failures. Due to the relatively low sampling rate used in the study failures lasting several minutes to several hours are generally only detected in a single traceroute session. Thus, for most persistent failures observed, the data provide loose lower and upper bounds on average failure duration of 450 seconds to several hours, respectively. Our analysis of the raw trace data indicates that 94% of the persistent failure events (excluding end-host failures) spanned only one traceroute sample; Of the persistent failures that last long enough to be seen in multiple traceroute sessions, our analysis yields lower and upper bounds on average failure duration of 8.6 hours and 30.9 hours if only network failures are considered. Because we cannot characterize persistent failures precisely, we consider two bounding models for the duration of persistent failures: Fail-S, in which we assume the shortest persistent failures consistent with the bounds above, and Fail-P, in which we assume the longest. For simplicity, we assume that persistent failures arrive at exponentially-distributed intervals with mean $1/P_{Up \rightarrow Perst}$ calculated to yield the specified P_{Perst} for a given mean failure duration.

For these experiments, we conduct four trials with different random seeds for the network failure model and graph the mean and standard deviation of results.

Methodology. Our simulator assumes that each server identified in the trace is implemented as a location-independent mobile extension that can be accessed either at the server, proxy, or client and that clients and proxies begin to download mobile extensions when they first access service. The amount of time to download, install, and get useful service from a mobile extension is a configurable parameter with default value $install\_time = 100$ seconds. During the $install\_time$, clients and proxies access the service from the origin server. In addition to mobile extensions, the simulator assumes that clients and proxies implement infinite-size traditional web object caches that store each cachable object referenced by the client or proxy.

If the network remains up during an entire session, the simulator classifies the session as *No Failure*. For sessions in which the network fails, the simulator examines the objects referenced in the session and classifies the session as follows: (1) *Cache Hit* if all requests are for fresh cached web objects; (2) *Stale Hit* if all requests are for cached web objects and if the trace indicates that some of those objects require updates from the server; (3) *Hoard-able Degraded* if the mobile extension is installed at the time of the network failure all requests are for cachable objects but some miss; (4) *Dynamic Degraded* if the mobile extension is installed at the time of the failure but not all session data are cachable; and (5) *Fail* if the mobile extension is not installed at the time of the failure and either some data are not cachable or some data are cache misses. Note that due to limitations of the trace and of the HTTP protocol, the traces may overstate the cachability of data and may underestimate the rate of change of data. Thus our results may understate the Stale Hit and Dynamic Degraded rates and overstate the Cache Hit, Stale Hit, and Hoardable Degraded rates.

Results. Our goals are to first quantify the improvements that mobile extensions can provide, then to determine what factors cause these improvements, and finally to determine the sensitivity of the approach to service installation time, network failure rates, and the number of simultaneous extensions proxies and clients maintain.

The y-axis of Figure 6 shows the fraction of sessions classified in the categories listed above. The x-axis shows the *install_time* for each service using a logarithmic scale, and each graph shows these results for a different workload. When installation times are short, caching plus mobile extensions can improve availability by at most factors of 16.7, 16.5, 16.3, and 5.96 for the four workloads compared to the failure rate that would be encountered if each request were sent to the origin server. The improvements available from caching alone appear small (reductions in failure rates of 1.12, 1.13, 1.38, and 1.30) although in the Squid workloads lower-level caches may hide sessions that only reference cached data, causing us to understate the benefits of caching alone. Conversely, aggressive hoarding plus caching may be able to achieve significant improvements; the simulations indicate upper bounds of 2.88, 2.91, 5.03, 3.78 for this combination.

The available benefits fall gradually as installation time increases. At a 10,000 second installation time the upper bound on availability improvements are 10.71, 10.24, 8.18, and 2.91 for the four workloads.

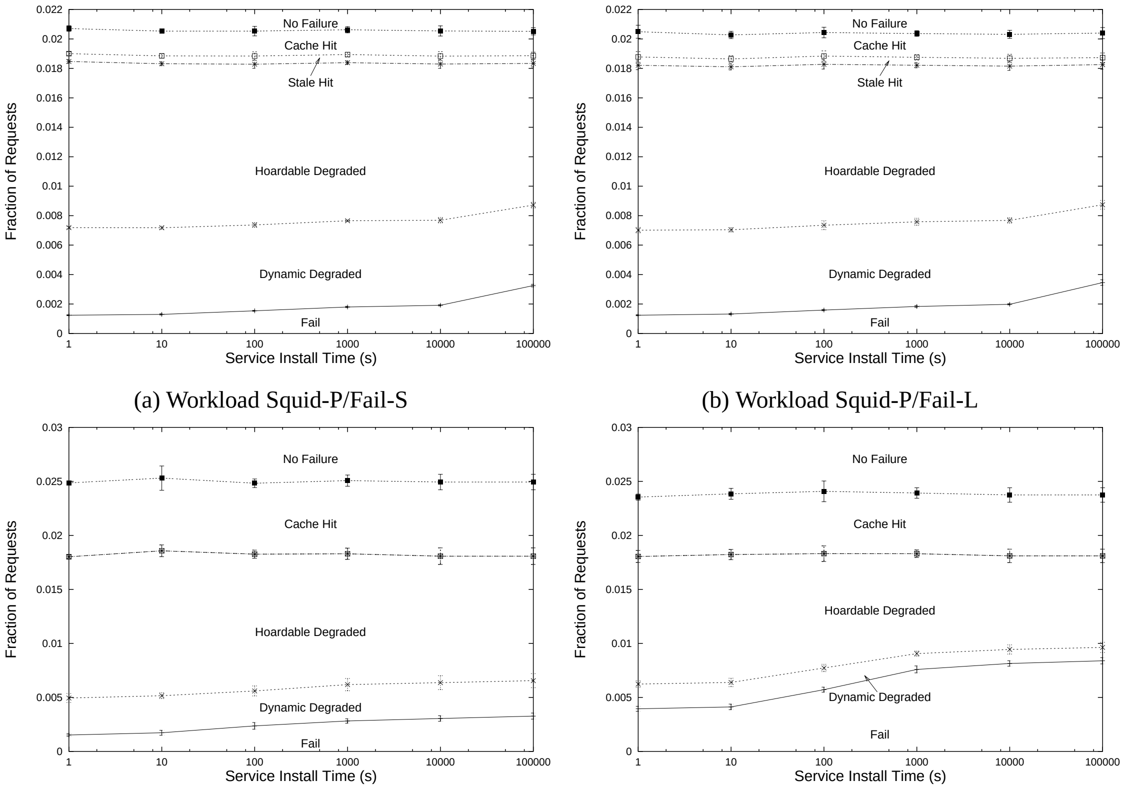
Figure 7 shows session results as we vary (a) persistent and (b) temporary failure rates . These data and the data in Figure 6-(a) and (b) suggest the improvement in session failure rate provided by mobile extensions are relatively insensitive to the underlying network failure patterns.

Figure 8 shows session results for caches and proxies that maintain only a finite number of local copies of extensions and evict the rest using an MFU policy (results for LRU replacement and exponentially decaying average MFU are similar but not shown.) Shown are the Squid-P/Fail-S and BU-C/Fail-S workloads; BU-P/Fail-S and Squid-C/Fail-S are similar. take advantage of mobile extensions, client and proxy virtual machines must be scalable to handle hundreds or thousands of simultaneously downloaded extensions. This paper does not explicitly address the issue of the scalability of Java virtual machines to such large numbers of extensions, and this appears to be an important subject for future work. These data also suggest that fair and efficient resource management will be important since large numbers of extensions may need to share a client or proxy's resources.

5.5 Disconnected catalog application

Consider an e-commerce application in which users *read* several pages of information, *add* an element to a shopping cart, and *commit* a purchase transaction. In a traditional server extension architecture, a network or server failure at any point in the interaction can cause the transaction to fail. Furthermore, the "store" is not available to mobile clients that are deliberately disconnected. To improve robustness of this application and to allow purchases to be made from mobile, disconnected devices, we have constructed a "Disconnected catalog" application using mobile extensions. This application hoards the contents of the catalog, and logs additions to the shopping cart to local disk. Upon commit, the client's system writes the order to an asynchronous message queue to be sent to the server and waits for up to two seconds. If, during those two seconds, the resulting origin-server response appears in the client's message queue, the client returns that response to the user immediately. If the two seconds elapse before the server response appears, the client extension proxy generates a reply to the client browser that indicates that the network connection to the server is slow or down and that the system will continue to retry the request, along with a "receipt" URL that the client may use to check on the status of the order.

We test this system in two configurations: *Origin Server* in which all requests go through the origin server and *ME-*



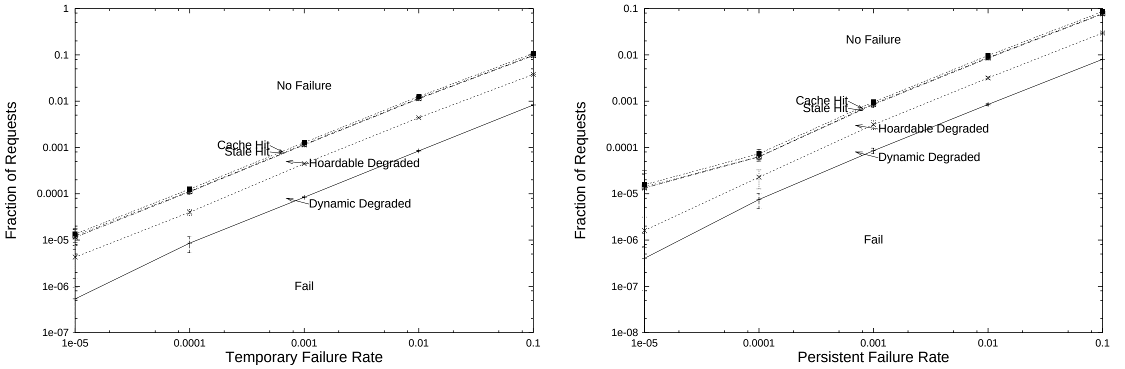
(a) Workload Squid-P/Fail-S

(b) Workload Squid-P/Fail-L

(c) Workload BU-P/Fail-S

(d) Workload BU-C/Fail-S

Figure 6: Session result v. server-extension installation time (*Simulation results.*)



(a) Workload: Squid-S, $P_{Perst} = 0$

(b) Workload: Squid-S, $P_{Temp} = 0$

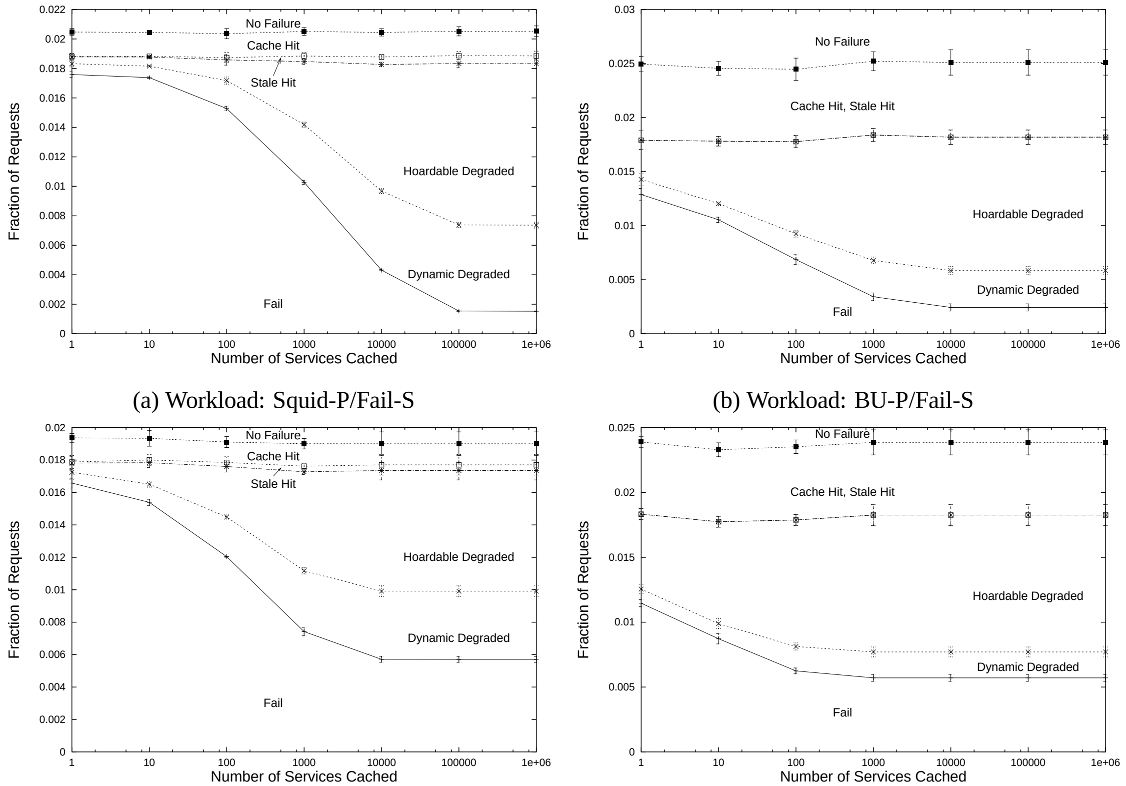
Figure 7: Session results as (a) persistent and (b) temporary network failure rates vary. (*Simulation results.*)

Local in which the client uses mobile extensions to download a local copy of the service as described above.

We test the system with a 233 MHz Pentium II running Windows NT acting as the client and a 166 MHz UltraSPARC running Solaris acting as the server. The client is connected to the Internet via a commercial ISP using a cable-modem. The server is connected via the University of Texas's connection.

We use a synthetic client that reads three pages with 20 seconds of think time between reads, then adds an item to the cart, then checks out. A client repeats this process 45 seconds after the previous session completes. We introduce synthetic failures of duration 30 + Exponential(40) seconds each. The failures arrive randomly according to an exponential distribution with mean 3500 seconds to yield an average failure rate of 2%. This failure pattern is intended to stress the system rather than to model any particular observed workload.

Figure 9 shows failure rates and performance for this application. The bars represent failure rates for *Read*, *Add*, and *Commit* requests sent to the *Origin Server* or requests sent to the *Mobile Extension*. The white bars represent failure rates where a client's request does not receive a reply from the service. The gray bar on the right shows cases of *degraded service* under mobile extensions where the client's commit request takes more than two seconds to



(a) Workload: Squid-P/Fail-S

(b) Workload: BU-P/Fail-S

(c) Workload: Squid-C/Fail-S

(d) Workload: BU-C/Fail-S

Figure 8: Service failure rate v. number of cached service extensions (*simulation results*.)

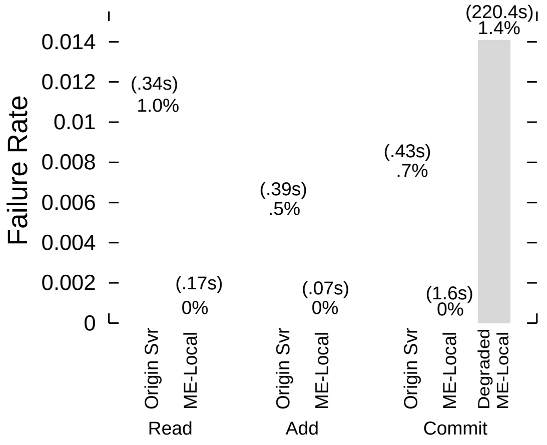


Figure 9: Failure rate and response time for catalog application.

be sent to and returned from the origin server, so the system gives the client a "receipt" and the client checks the URL later for updates. In parenthesis above each bar is the average time for the successful requests of the specified type (except in the case of the "degraded" bar, which shows elapsed time to complete the degraded requests.)

This experiment illustrate how mobile extensions can improve robustness by decoupling the service from the network connection. At the same time, it illustrates that this approach transforms but does not eliminate the problems caused by connectivity failures: situations that cause transaction failures in the Origin Server configuration cause delays in the mobile extension configuration. Finally, these data illustrate that an additional benefit of the approach is significantly improved responsiveness for most requests from moving the bulk of the service near the client.

5.6 "Hospital" application

In this experiment, we consider the following scenario: doctors in a hospital use web terminals to order laboratory tests for their patients, technicians receive these orders, and when the tests are complete, a technician enters the results into a terminal, after which the doctor may view the result. It is easy to build and deploy a *prototype* of such an application under HTTP's server-extension architecture with a centrally-located server or server cluster. However, such an implementation would be vulnerable to Internet failures. Given that most network outages are short in duration [24], this application may be able to provide acceptable, but degraded service across network failures using asynchronous messaging with persistent message queues to transmit orders among participants.

We constructed a prototype of this system that consists of three separate extensions, each built around a persistent message queue. The doctor-client extension buffers the doctor's order submissions and asynchronously receives results that are pushed to it. The lab-client asynchronously receives pushed orders and buffers results as they are for-

	Rem/Rem	Loc/Rem	Loc/Prxy
Success($<$1 min)	82.94%	78.59%	99.90%
1 - 10 min	0.67%	3.26%	0.00%
$>$ 10 min	7.88%	17.86%	0.015%
Failure	8.58%	0.00%	0.085%

Table 3: Network- and infrastructure-induced failures and delays. Success indicates the percentage of requests that were satisfied within 1 minute of the best case completion time of 10 minutes. The next two lines indicate degradation of service in which requests and replies are delayed. Finally, the last column includes cases where the system refused to accept an order request.

warded to the rest of the system; both the doctor-client and lab-client are accessed via standard HTTP. Finally, the server application routes orders from a doctor to the right lab and routes results the other way.

All extensions are mobile and the location that they run will affect robustness. In the Local/Proxy case, the client programs install themselves on the doctor and lab terminals and server program installs itself at the hospital proxy machine. In the Local/Remote case, the client programs install themselves at the lab terminals, but the server runs at a central site. Finally, in the Remote/Remote case, the terminals do not support mobile extensions, so all three programs must run at the central site.

We deployed mobile extension virtual machines to four sites distributed across the Internet – three at academic institutions and one at a DSL client of a commercial ISP. Due to machine limitations, we approximate the Local/Proxy case with both clients and the server running on the same machine. We simulate a workload in which doctors submit orders at times distributed exponentially with mean 10 minutes. Lab technicians' browsers poll for orders every 10 seconds. When labs receive an order, they process it in exactly 10 minutes and then send the result. We introduce faults by killing the central server machine for uniform random intervals from 1 to 10 minutes and separate these crashes by uniform random intervals from 10 to 60 minutes. These failure rates are higher than we expect in practice, but they allow us to observe the behavior of our systems under stress.

Table 3 summarizes the results of this experiment. In this scenario under the Remote/Remote case, which corresponds to traditional HTTP, a large number of requests fail (because the doctor can not contact the server to submit an order) and a similar number of requests are delayed by more than a minute when a network failure prevents a submitted order from progressing. Column three ("Loc/Rem") shows that moving the client programs and their associated write buffers to the clients eliminates submit failures, but can transform those failures into multi-minute delays; in practice, this system would also have to warn users when requests are delayed for more than a few minutes. Finally, by taking full advantage of client and proxy programmability to move the server to the hospital as well eliminates nearly all long delays and failures.

Note that we did encounter one failed request in the Local/Proxy case where the machine hosting one of our experiments crashed in a way that left a client process running just enough longer than the proxy process that it was able to observe and log a failure to connect. This illustrates a true limitation of the system: the system provides tools for eliminating one (we believe significant) source of service disruption, but as one factor is reduced, others, in this case host failures, become significant.

6 Related work

This work is closely related to the ideas of the Rover toolkit for building mobile applications [17], which uses mobile code, caching, and QRPC to allow applications to work when disconnected. The differences between the approaches stem from our focus not only on mobility but also disconnections due to network and server failures and our goal of providing an infrastructure for large collections of services that are installed without user intervention or even knowledge. As a result, we focus more of our attention on resource management, provide location independence so that extensions can run at clients, at proxies, and at servers, and we quantify the end-to-end availability advantages such an approach can provide.

As noted in the Introduction, a large number of mobile code systems have been proposed or built. The strategies we discuss in this paper focus on providing disconnected operation for robustness and mobility and on the resource management problems that arise in such an environment.

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 [16, 22] or can be monitored for progress [29, 10]. 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 [23] 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 must be restrained to match the rate at which the resource may usefully be transferred between applications.

A number of economics-based strategies have been proposed for distributing resources among competing applications in large distributed systems [27, 4]. These systems target more general network topologies than ours and they use secure electronic currency to ration resources.

File caching [15], replication [31], hoarding [19, 21], and write buffering are standard techniques for coping with disconnection for static file services. Active channels [2] 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, presumably to control servers' use of client resources. Similarly, Internet Explorer lets users identify groups of pages to hoard, but users must manually select sites and indicate refetch frequency and hoard depth.

7 Conclusions and future work

HTTP's server extension strategy for deploying distributed services is successful in part because it facilitates incremental deployment and extensibility with simple default semantics. Unfortunately, this approach is fundamentally limited with respect to end-to-end availability and supporting mobility. Mobile extensions seek to fix those limits while retaining those advantages.

Our evaluation of our prototype system and our simulation studies suggest that mobile extensions hold promise. However, future work is needed to fully understand this approach. For example, we have evidence that aggressive hoarding could significantly improve service availability, but we have not quantified the network, disk, or server-load cost of pursuing that strategy. Also, it appears that for a large number of services to make use of mobile extensions, clients and proxies must be prepared to host hundreds or thousands of extensions. Future work is needed to evaluate and improve the scalability of our virtual machine. Finally, we have explored one class of mobile middleware extensions for improving service robustness in which servers send clients extensions that can act autonomously on their behalf. It may be interesting to consider other techniques enabled by a mobile extension framework such as server selection/fail-over [35] or having clients ship code into the network to act on their behalf during periods of disconnection [26].

Acknowledgements

We thank Anand Aggarwal, Tom Anderson, and Amin Vahdat who helped us build the Active Names framework on which the mobile extensions system was constructed. We thank Jairam Ranganathan who helped to build an earlier version of the robust catalog application. We thank the National Laboratory for Applied Network Research (NLANR) for making the Squid access logs available under National Science Foundation grants NCR-9616602 and NCR-9521745, the Oceans research group for making the BU traces available, and Steve Gribble for making the UCB traces available.

References

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

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

[3] G. Banga, P. Druschel, and J. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *OSDI99*, February 1999.

[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] cgi@ncsa.uiuc.edu. The Common Gateway Interface. http://hoohoo.ncsa.uiuc.edu/ cgi/, 1996.

[7] IBM Corporation. Mqseries: An introduction to messaging and queueing. Technical Report GC33-0805-01, IBM Corporation, July 1995.

[8] C. Cunha, A. Bestavros, and M. Crovella. Characteristics of WWW Traces. Technical Report TR-95-010, Boston University Department of Computer Science, April 1995.

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

[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] D. Engler, M. Kaashoek, and J. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the Fifteenth ACMSymposium on Operating Systems Principles*, pages 251–266, December 1995.

[12] General Magic. Telescript Technology: Mobile Agents. `http://www.genmagic.com/Telescript/Whitepapers/wp4/whitepaper-4.html`, 1996.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] Keynote.com. Web Performance Index. *Internet World*, page 30, April 2000.

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

[20] K. Kotay and D. Kotz. Transportable Agents. In *Proceedings of the CIKM Workshop on Intelligent Information Agents*, Dec 1994.

[21] 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.

[22] 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.

[23] 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.

[24] V. Paxson. *Measurements and Analysis of End-to-End Internet Dynamics*. PhD thesis, University of California, Berkeley, April 1997.

[25] H. Peine and T. Stolpmann. The Architecture of the Ara Platform for Mobile Agents. In *Proc. of the First International Workshop on Mobile Agents*, April 1997. http://www.uni-kl.de/AG-Nehmer/Projekte/Ara/Doc/architecture.ps.gz.

[26] S. Phatak, V. Esakki, B. Badrinath, and L. Iftode. Web&: An architecture for non-interactive web. In *Proceedings of the Nineth International World Wide Web Conference*, 1999.

[27] 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.

[28] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource Kernels: A Resource-centric Approach to Real-Time and Multimedia Systems. In *SPIE 3310*, pages 150–164, January 1998.

[29] 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.

[30] D. Tennenhouse and D. Wetherall. Towards an Active Network Architecture. In *Computer Communication Review*, 1996.

[31] 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.

[32] A. Vahdat, T. Anderson, M. Dahlin, E. Belani, D. Culler, P. Eastham, , and C. Yoshikawa. WebOS: Operating System Services for Wide Area Applications. In *Proceedings of the Seventh IEEE International Symposium on Higher Performance Distributed Computing*, July 1998.

[33] 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.

[34] A. Wolman, G. Voelker, N. Sharma, N. Cardwell, M. Brown, T. Landray, D. Pinnel, A. Karlin, and H. Levy. Organization-Based Analysis of Web-Object Sharing and Caching. In *Proceedings of the Second USENIX Symposium on Internet Technologies and Systems*, October 1999.

[35] C. Yoshikawa, B. Chun, P. Eastham, A. Vahdat, T. Anderson, and D. Culler. Using Smart Clients to Build Scalable Services. In *Proceedings of the 1997 USENIX Technical Conference*, January 1997.