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
April 29, 2000

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

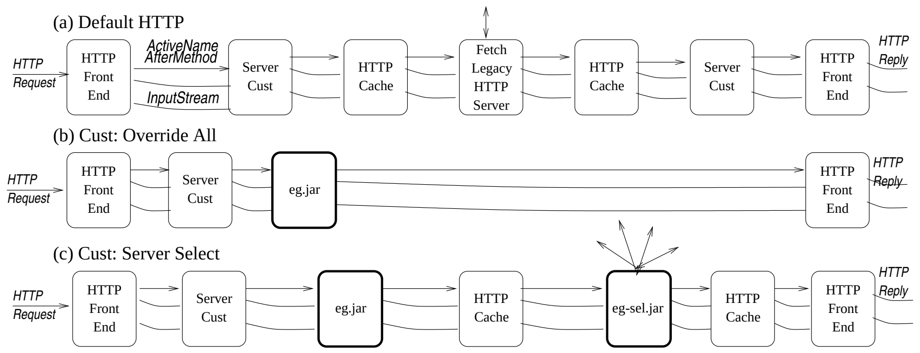

Figure 1: Example service/extension program pipeline of (a) standard HTTP protocol, (b) completely overridden protocol, (c) partially-overridden protocol.

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