Draft: To appear in the USENIX Symposium on Internet Technolgies and Systems Oct 1999.
Please see updated version at http://www.cs.utexas.edu/users/dahlin/papers.html after September 1, 1999

Active Names: Flexible Location and Transport of Wide-Area Resources*

Amin Vahdat
Department of Computer Science
Duke University

Michael Dahlin
Department of Computer Science
University of Texas, Austin

Thomas Anderson Amit Aggarwal
Department of Computer Science and Engineering
University of Washington

Abstract

In this paper, we explore flexible name resolution as a way of supporting extensibility for wide-area distributed services. Our approach, called Active Names, maps names to a chain of mobile programs that can customize how a service is located and how its results are transformed and transported back to the client. Relative to proposals such as Active Networks to add extensibility at the packet level, the connection-oriented approach of Active Names offers simpler and more efficient end-to-end semantics. Relative to proposals such as Active Services to add migratable computation at the application level, Active Names offers a single framework supporting the composition of a wide variety of proposed new services, including replica management, client customization, cache management, compression, mobility, universal resource names (URNs), and global object systems. To illustrate our system, we implemented advanced prototypes of the first four of these services, demonstrating up to a five-fold performance improvement to end users. We show how these new services are developed, composed, and secured in our framework. Finally, we develop a set of algorithms to control how mobile Active Name programs are mapped onto available wide-area resources to optimize performance and availability.

1 Introduction

Historically, the Internet's Domain Naming System (DNS) was developed to support a simple one-to-one mapping from machine names to IP addresses [52]. More recently, however, Internet names (such as "cnn.com") have come to represent complex distributed services, potentially implemented by hundreds of machines spread across different continents. While simple replica management strategies have been added to DNS (e.g., pick a random replica as representative of the logical service [45]), better performance and end-to-end semantics could be provided by taking into account a myriad of more dynamic factors, such as server CPU loads, network congestion, and client reference patterns [10, 74, 11], suggesting the need for a more general architecture for mapping names to services.

More broadly, academic and industrial researchers have proposed a bewildering array of new services for mediating between clients and servers; again, we believe this argues for rethinking the architecture for mapping between names and services. In addition to automatic load balancing and failover among server replicas, examples include dynamic redistribution of replicas over the wide area [66], compression and distillation of multimedia content [28, 3], proxy cache extensions such as support for hit counting and dynamic content [14, 65], client customization of web content [18, 71], network address translators (NATs) [26], packet delivery services for mobile hosts [16], and so forth.

To address this need for rapid deployment and extensibility of Internet services, a variety of proposals have been made to support "active" (dynamically migratable) computation inside the network, leveraging Java, Safe-Tcl, and other systems supporting machine-independent executables [32, 68, 49, 41, 72]. Example "active" networking proposals include Active Networks [70], Active Caches [14], Active Services [3, 33], RentAServer [66], and others. In addition, a number of proposals have been made for single point (non-migratable) extension code, such as HTTP front ends [28], NATs, magic routers [4], and router-based load balancing [11, 17]. These approaches have varied in where in the protocol stack the computation is applied – from packet filtering to connection filtering to transparent proxies to giving up on transparency and making the application responsible for everything.

If active computation can be applied anywhere in the protocol stack, where in the protocol stack *should* it be done? The danger in the rapid pace of proposed new mediation services is that the new services are unlikely to be developed with an eye to preserving consistency with each other, or with the existing Internet. Anyone who has used a PC knows firsthand the management nightmare posed by an unbridled extension architecture [29]. Installing one application (and its extensions) often breaks other previously working applications; applications routinely defend themselves by re-

*This work was supported in part by the Defense Advanced Research Projects Agency (F30602-95-C-0014, F30602-98-1-0205), the National Science Foundation (CDA 9401156, CDA 9624082), Cisco Systems, Sun Microsystems, California MICRO, Novell, Hewlett Packard, Intel, Microsoft, and Mitsubishi. Anderson was also supported by a National Science Foundation Presidential Faculty Fellowship. Dahlin was also supported by an NSF CAREER grant (CCR 9733742).

installing their extensions on top of later software, creating a race to the bottom between extensions as viruses.

In this paper, we argue for *active naming* as a unifying principle, to efficiently support the composibility of a wide variety of new services while providing correct end-to-end behavior. Our Active Names system maps names to a chain of mobile programs responsible for finding the remote service and mediating its response. A service owning a portion of the namespace (e.g., Time-Warner in the case of cnn.com) has complete control over which protocols are used to access the service, along with control over where in the network those protocols run. Similarly, a client machine can use Active Names to customize how services are presented to the user. Thus, with Active Names, it is natural to express circumstance and client-specific name resolution, where the same logical name can be resolved to different actual representations. For example, a mobile user with an expensive wireless connection in Europe will want a different version of the "cnn.com" resource from a different replica than a user in the United States with a T1 connection.

To illustrate these points, we have constructed a prototype Active Naming system and have used it to implement a set of complex distributed services including (i) replica selection that adapts to client location as well as network and server load, (ii) image retrieval protocols that migrate distillation code dynamically around the network to optimize client response time, and (iii) a cache enhanced to support both client customization and caching of active content. Our results show Active Names extensions offer a significant advantage relative to existing static approaches.

Using our experience with the Active Name prototype, this paper makes a number of contributions in the area of extensibility and migratable code in the protocol stack. We develop a distributed continuation-passing mechanism to support the efficient composibility of services. We devise a flexible bootstrapping mechanism using our naming system to load code on remote machines. We extend the namespace delegation paradigm used in DNS and file systems to the composition of network services. We examine the demands that a set of key services place on active computation frameworks. Finally, we define a set of algorithms for determining the placement of mobile code based on dynamic network conditions.

Programmable naming is not new (neither is extensibility), and we have borrowed many aspects of our approach from earlier work. For example, object-oriented systems such as Smalltalk have long provided application control over the binding between caller and callee. The Mercury RPC system did the same in the mid-80's for a distributed client-server environment. Likewise, database transaction processing (TP) monitors manage server replica selection and consistency on behalf of clients. Traditionally, TP monitors run only on the client and are not customizable by application code; recently, however, TP monitors have begun to support mobile code, although not to the extent proposed here [34].

The remainder of this paper covers these issues in more detail. We first discuss the strengths and weaknesses of Active Names by contrasting it to two popular alternatives. We next outline the the architecture of our system; we then describe several applications we have built on top of our system. We conclude with related work and a summary of our contributions.

2 Motivation

To motivate our decision to provide extensibility via naming, we compare our approach to two popular alternatives, Active Networks and Active Services, that comprise extreme endpoints in this design space. At one extreme, by applying arbitrary computation on packets as they flow through routers, Active Networks can be completely general and transparent to end hosts; however, this transparency comes at a cost of both efficiency and in making it more difficult to express end-to-end semantics. At the other extreme, Active Services provides a framework for applications to execute arbitrary computation in the network; however, each application is free (even encouraged) to link with a different framework customized to its needs, making it more difficult to share extensions across applications. Active Names attempt to combine the best of both worlds; of course, our approach has its own set of limitations.

A principal advantage of our approach is that naming is at the top of the network protocol stack; by hijacking name resolution, we can gain control over (and therefore can extend) any network access. At the other end of the spectrum, hijacking packet processing inside routers likewise offers the ability to extend any network access. For example, consider how a client selects among a set of server replicas. Naively, the client can choose randomly among a set of replicas, but the choice can also be made on a number of other grounds, such as the network characteristics between the client and server replicas, relative server loads, and client reference locality. In our system, a new policy for server selection can be implemented by mapping the service name to a program that selects the replica. Equivalently, the same policy could be implemented at the packet level inside programmable routers by mapping the name to a group address, and then dynamically routing packets to the desired server.

However, extending names offers simpler end to end failure semantics than is possible when extensibility is applied further down the protocol stack. Today, it is possible to build highly available services inside of a machine room [60, 5]. However, the end-to-end availability of a wide-area service further depends on external factors such as power outages and whether packets can be routed from a client to the machine room. For example, routing pathologies can make a service appear unavailable to some clients even though the service is otherwise "up". To cope with these external factors, the service needs to be replicated at multiple geographic locations, each of which may fail independently. In the case of many read-only replicas, it is straightforward to redirect requests to a failed replica to any member of the group at multiple points in the protocol stack. However, for replicas that are writeable, maintain state, or require authentication, the recovery protocol can require the coordinated activity of both the client and other replicas; for example, Bayou guarantees session consistency by restricting clients to bind only to replicas that have seen the client's updates [63]. Implementing session consistency correctly via transparent packet processing requires the network to maintain hard state about the behavior of the client; worse, this state largely duplicates what the client already has stored in its cache. Cheriton and Skeen [15] have cataloged numerous examples where failure recovery cannot be correctly implemented inside a transparent transport layer protocol. In our model, replica failures can be either reflected back to the client or handled transparently, under control of the program providing the binding between the name and the server.

Active Names is also more efficient than packet level filtering for those services that can be provided within our model. By interposing on connection setup, the overhead of programmability is typically paid once per connection, instead of once per packet. The result of an Active Name resolution is usually just a reliable byte stream for transmitting requests to the server and replies back to the client. We have the flexibility to return a filtered connection, however, with a chain of mobile programs applied to every byte on the connection. For example, we can install monitoring code on the connection to provide correct failover behavior; we can install per-byte filters to perform compression/decompression on the path between the client and server. Nevertheless, there are some services which cannot be provided above the packet layer and thus are not supported by our system; these include packet-level resource allocation in routers and multi-host transparent packet filtering, such as firewalls.

At the other extreme, some researchers have proposed customized application-level frameworks as a way of supporting application-specific computation inside the network. For example, the Active Services framework implements dynamically relocatable multimedia gateways; they suggest that different frameworks would be needed to support different applications [3]. Our approach differs from Active Services in that we are trying to provide a single, general-purpose framework capable of supporting the composition of a wide range of new services. Our goal is to allow Active Name modules to be developed and reused in a variety of contexts [42]; for instance, Active Services supports customizable filtering at the media gateways, but does not support customizable protocols for locating, managing, or communicating with the gateways, nor does the framework support dynamic installation of new implementations of client software. Except for the code to transform the multimedia stream to fit a limited link capacity, the client-gateway and gateway-server protocols in Active Services are fixed and non-extensible. By contrast, Active Names allows all aspects of service location and transport to be customized by the namespace owner; code is referenced by name, allowing us to use Active Names to locate and download new implementations of extension code whenever the namespace binding is changed.

An additional advantage of our approach is that human-readable names are a naturally extensible global virtual name space, analogous to virtual memory addresses in computer architecture. IP addresses form a physical name space, analogous to physical memory addresses. We believe there is tremendous leverage to be gained from making the mapping from virtual to physical addresses programmable, just as there was in computer architecture with software TLB's [21]. Instead of supporting mobile computers by translating IP addresses transparently inside the network, Active Names can support mobility directly by translating a name into a connection that reconfigures itself when the endpoint moves. Likewise, network address translators (NATs) have been proposed to cope with limitations in the IPv4 address space, but they can cause problems by not being completely transparent. In case a NAT fails, its bindings must be replicated as hard state on every alternate path the packets may take around the failure; it can be difficult to know when the hard state can be safely reclaimed. With Active Names, IP addresses can be allocated dynamically when the connection is created and deallocated when the connection terminates, in much the same way that virtual memory paging backs virtual addresses with physical addresses when they are in use.

One might argue that a significant drawback to Active Names is that we modify the existing naming abstraction. Traditionally, Internet name resolution returns an IP address; the application uses the IP address to establish a socket connection to the end host. With Active Names, name resolution and retrieval of the specified resource is a single step. In effect, we are combining *location* of a resource and its *transport* back to the client into a single step. For many applications, this can be hidden inside an application-specific proxy; for example, web browsers can be configured to connect to a proxy which mediates the browser's interaction with the network.

A similar debate on the structure of naming has occurred in file systems and databases; the conclusion has been that it is dangerous to separate naming from use [58]. Superficially, it can seem simpler to have the naming system return an ID (IP address, inode number, or physical memory location) which is then used by the system to access the named resource. However, if the ID is visible to applications, the ID becomes hard state, something that must continue to be supported by the system even after the binding has changed or the reference has become invalid.

3 Active Name Architecture

To support the functionality discussed above, four goals drive our architectural decisions. The architecture must (i) support customization and extensibility of each namespace, (ii) support composibility of different namespace customizations, (iii) support the efficient use of network resources, and (iv) support location-independent execution of namespace resolution.

Our core system is simple, and we will describe it by examining four key concepts of the design. First, to support extensibility, the architecture follows a microkernel philosophy of providing a basic set of building blocks and allowing services and clients maximum freedom to customize the system for their needs. Second, to support efficient use of network resources, location-independent Active Name programs represent services and handle name resolution for them. Third, to support composibility, namespaces may delegate resolution to other namespaces. Fourth, to support composibility and network efficiency, our programming model is to construct a chain of unidirectional filters from the source of a request, through intermediate services, to the reply sink. Because one of the major motivations of Active Names is to improve the performance of clients accessing services, our design is careful to allow room for several performance optimizations that complicate an otherwise straightforward design.

The core of the system is fully functional, and we have constructed a number of useful applications (described in the next section). The system is complete and stable enough for our own internal use. As detailed below, some aspects of security have not been fully integrated into the prototype.

3.1 Microkernel Approach

The core system provides low-level basic services and, whenever possible, allows services and clients the freedom to customize the name system to their own needs. Programs assume the availability of an execution environment that provides a common set of local resources. At a minimum, each node must provide for the safe execution of untrusted code, local soft state, and a means for communicating with and invoking programs on remote nodes.

For safe execution, our prototype relies on the Java-2 security system [69], but we could have just as easily chosen another mechanism such as hardware protection domains or software fault isolation [68]. Without such a mechanism, it would be impossible for one Active Name program to delegate to another program with restricted or amplified rights. On top of this basic security mechanism, individual programs define policies for delegating namespaces they control and for accepting requests from other namespaces.

Nodes also provide local soft state to allow services to cache and buffer data temporarily inside the network, and they provide access to the network to allow programs to be invoked on other machines. As we will describe in more detail below, each service is entirely responsible for its own local state: for propagating updates to remote reliable storage, for managing the space as a scarce resource, and for providing consistency if needed between the local state on multiple machines. A key feature is that a service's local state is referenced by name; the state can only be accessed directly by the service that created it. In a production system, nodes would enforce resource limitations using technology such as JRES [19]; such functionality is not implemented in our prototype.

3.2 Active Name Programs

The goal of a user in typing a name is to invoke some service: for example, to fetch some piece of data, to buy a book, to send a piece of email, etc. The traditional model for Internet naming is that the user-provided name is translated into the IP address for the machine hosting the service; this IP address is used to communicate with the server using some well-known protocol (e.g., HTTP or sendmail). As noted earlier, this approach makes end-to-end availability difficult and it makes extension and customization difficult.

We believe a more natural model is to allow the service to bind to a location-independent Active Name program to be run near the client when the service is invoked (e.g., by name). This program is the service's representative, completely free to use whatever protocols the service desires to communicate with its servers. For example, an intermediate cache inside the network becomes part of the service, rather than the client's agent; the service can determine its own cache replacement, prefetching, and consistency policies. By updating the binding to a new program, a service can upgrade its protocols and algorithms – for example, from HTTP to persistent-HTTP [56] to WebTP [39], or to different cache and replica consistency policies. Although new policies could be built into the protocols by which the service is contacted, doing so today requires manual upgrades to all hosts that contact the service. Even if the protocols could be upgraded once, the resulting system would be inflexible to further algorithmic improvements. In contrast, by making Active Name programs first class abstractions in our system, backward compatibility with old protocols needs to be preserved only for the length of time it would take for the binding to be propagated, and individual services can adopt improved protocols at their discretion.

3.3 Hierarchical Name Space

Active Names are organized hierarchically into *namespaces*, analogous to domains in the Domain Naming System (DNS) and directories in a UNIX file system [61]. Names within a namespace can be, in turn, namespaces (subdomains in DNS or subdirectories in UNIX); they can also be terminal leaves in the naming tree (machines in DNS, files in UNIX). Each namespace has a program associated with it that is responsible for interpreting that portion of the namespace; this program is free to interpret the names within the namespace in any fashion it wants. A root namespace is responsible for initial interpretation of all names. Each namespace has an owner with the right to determine the program bound to the namespace. The client, by default, is the owner of the root namespace. This allows the client to install a program to mediate how its names will be translated. For example, a PDA could install a root program to take whatever is returned by lower level name spaces, and compress any images to fit the screen size [28].

To illustrate how delegation works, consider how our system implements the WWW namespace to support per-service naming and transport. Traditionally, users type web requests that specify a specific transport protocol (e.g., "http") along with the service name (e.g., "cnn.com"). But the transport used to communicate with a service should not be the concern of the end-user. In our framework, users simply name the service they wish to contact, and services specify the transport for the names they control via the hierarchical namespace delegation mechanism. In particular, the root namespace sends web requests to the WWW-root Active Name program which implements the WWW-root namespace. For bootstrapping and by default, WWW-root delegates incoming HTTP requests to a series of Active Name programs that implement the HTTP-default namespace. But under the Active Names paradigm, rather than delegate resolution of all names to HTTP-default, the WWW-root namespace has the right to delegate portions of the WWW namespace to other Active Name programs according to any policy it chooses. The WWW-root's policy is to set these mappings as follows: the response for a request to a URL may include in its MIME header a directive specifying an Active Name program to be invoked for subsequent requests for which that URL is a prefix. For example, the reply to a request to www.cs.utexas.edu/home/smith can delegate the www.cs.utexas.edu/home/smith/active/* namespace, but it can not delegate the www.cs.utexas.edu/home/jones/* namespace, the www.cs.utexas.edu/* namespace or the www.cs.duke.edu/home/smith/* namespace. This technique is similar to the delegation strategy used in Active Caching [14], but in keeping with the namespace paradigm, it allows parent directories to control the delegation of entire subdirectories rather than requiring delegation on an object-by-object basis.

This approach of associating a program with each level of a hierarchical namespace is not new. The HP Brevix and MIT Exokernel research file systems, for example, allow users to define application-specific programs for each directory in a file system [27, 44]. A directory's program is completely responsible for managing the bits stored inside the directory; for example, this allows applications to customize on-disk data structures for application-specific reference patterns (e.g., storing HTML files with cross-links in the same disk cylinder).

Of course, to be practical, our Active Name architecture must be able to be smoothly integrated with legacy clients, servers, and name databases. We accomplish this by using either a library or a proxy that provides default translations between legacy names and corresponding Active Names. For example, whenever a program is not defined for a machine name, we fall back on using DNS name bindings.

3.4 Programming Model

The key idea in the programming model is that as Active Name programs in the delegation path execute, they are given the opportunity to insert themselves as filters in the path from the request to the response. Intuitively, if an Active Name program acts as a layer in a protocol stack, each program should also provide a bi-directional pipe between the layer above it and the layer below, to provide a path for bytes to be sent between the client and the server. Each layer would then be able to filter the bytes sent on the connection as needed.

For efficiency reasons, we take a slightly more complicated approach. Frequently, an Active Name program is only a forwarding agent – it points to where the named resource can be found. In this case, it would be inefficient to treat the chain of Active Name programs as a pipe, forcing all bytes to traverse back through the chain of programs that led to the server; the inefficiency is particularly pronounced when the forwarding agent runs on a machine remote from both the client and the server. Rather, our system uses a form of "multi-way RPC": the program forwards the request to the server, and the server can then communicate directly back to the client. Again, this is not new to our system; the same paradigm is used in many modern file systems to avoid putting the file manager in the data path between the disk and the client [6, 30].

At the API level, an Active Name program takes a string (the remaining part of the name to be resolved) and a reference to a data stream, and it either passes that reference to the next Active Name program in the pipeline or it processes the data stream and passes on the resulting stream. So that each Active Name program knows who to invoke next and because a remote invocation may never return back to the node that called it, programs are coded in a distributed continuation-passing style. Before passing control to the next program to interpret the remainder of the name, the current program bundles up its remaining work into an "after-method" and prepends it to the list of after-methods created by earlier programs. The chain of after-methods is effectively a script of filters to apply to the data being returned by the service. For example, a program to compress an image to fit on a PDA would add the compression routine as an after-method; similarly, an anonymizer would push a de-anonymizer onto the after-methods list. Like other Active Name programs, these after-methods are free to run anywhere and subsequent programs may reorder the list, for example, to run the compression routine closer to the server to reduce bandwidth consumption.

To provide a hook for Active Name programs to enforce security, the interface also provides as input a capability certificate that identifies the caller and which may grant a subset of the caller's rights to the callee. If the program is invoked from a remote node, the certificate will be authenticated via encryption techniques; if the program is called locally, the identity of the caller is guaranteed by the integrity of the local operating system. An Active Name program is free to use this information about the caller for access control. For example, a program could choose to run only on behalf of previously registered users. Similarly, if a program needs to enforce that its after-method is invoked, a certificate can be generated by each after-method program, signifying that the program has run to completion. Certificates may also be required from the machines used to run the programs and after-methods, since a program's results should not be trusted unless it is run on trusted machines. We expect Active Names programs to leverage the work of other researchers in showing how to provide authentication and access control for mobile computation [2, 9]. We have implemented a prototype of such a certificate-based capability system, but we have not yet integrated this functionality into the Active Names prototype.

3.5 Bootstrapping

Up to now, we have assumed that the name bindings were automatically available to the client and that the code to be executed was already loaded on the machine. Like local soft state, code in our system is referenced by name. For example, the binding for "cnn" might be "`cnn.com/code/code.jar`". This allows us to use the Active Name system itself to load programs, including the ability to use programs to locate the nearest replica storing the program, to maintain version consistency, and to compress/decompress the binary. Of course, the recursion has to stop somewhere: a small number of initial programs are loaded onto each Active Name machine to get the system started. As we discussed earlier, when a name lookup fails, the system falls back on DNS and traditional protocols for contacting the server at the IP address specified by DNS. When the server replies, it (or an application-level proxy) provides an after-method that installs the code for the new binding in the service's local state. The server can use the same mechanism for lazily installing new bindings/code versions.

4 Applications

Given the Active Names framework described above, this section demonstrates the power of three key Active Name principles: extensibility, location independence, and composibility. First, we describe how Active Names support flexible end-to-end bindings between clients and replicated Internet services. Next, we show how Active Name filters are dynamically allocated to strategic wide-area locations to maximize client performance, reduce consumed wide-area bandwidth and server load. Finally, we demonstrate the generality of the system by showing how individual Active Name extensions are composed together to provide significant performance benefits over any individual extension.

4.1 Extensibility

Active Names allow service-specific programs to account for any number of variables in choosing among server replicas, including client, server, and network characteristics. It is beyond the scope of this paper to determine the appropriate replica-selection policy for arbitrary services. However, we attempt to motivate the need for programmability in the

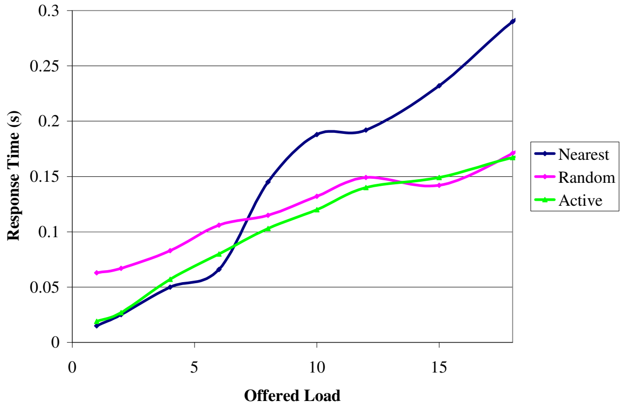


Figure 1: Replicated Service Access.

replica selection mechanism. To this end, Figure 1 summarizes the results of an experiment demonstrating the importance of introducing programmability into the server selection process. For these measurements, between one and eighteen clients located at U.C. Berkeley attempted to access a service made up of two replicated servers, one at U.C. Berkeley and the second at the University of Washington. The clients use one of the three policies to choose among the replicas:

- *DNS Round Robin*: In this extension to DNS, a hostname is mapped to multiple IP addresses, and the particular binding returned to a client requesting hostname resolution is done in a round robin fashion. Services employing DNS round robin achieve randomly load balanced access to replicas. In our experiments, we implement the Round Robin approach in Active Names by randomly choosing among available replicas.

- *Distributed Director*: This product from CISCO [17] executes specialized code in routers to allow services to register their current replica set. Requests (at the IP routing level) bound for a particular service, are automatically routed to the closest replica (as measured by hop count). While still not extensible, Distributed Director achieves geographic locality for service requests. In our experiments, we implement the Distributed Director approach in Active Names by always choosing the nearest replica.

- *Active Names*: With this instance of programmable replica selection, the program uses the number of hops (as reported by traceroute) from the client replica to determine the choice of replica. Replicas further away are less likely to be chosen than nearby replicas. However, this weighing is further biased by a decaying histogram of previous performance. Thus, if a replica has demonstrated better performance in the recent past, it is more likely to be chosen. For example, if replica A is 4 hops from a client, while a replica B is 5 hops away, A is randomly chosen 55% of the time. This probability is equally weighed with observed performance from the replicas. Thus, if performance histograms predict a 5 second access time for A and 4 second access time for B, based on performance alone, B will be chosen 55% of the time. Based on confidence in performance prediction and desire to localize traffic, different weights can be assigned to these two components. For our experiments, the components were weighed equally.

Figure 1 plots the average latency as a function of offered load as perceived by clients continuously requesting a 1 KB file from the replicated service. At low load, the proper replica selection policy is to choose the "nearest" replica at U.C. Berkeley. Thus, the Distributed Director policy shows the best performance at low load. However, as load increases, the U.C. Berkeley replica begins to become over-loaded, and the proper policy is to send approximately half the requests to the University of Washington replica. In this regime, the cost of going across the wide area is offset by the high load at the U.C. Berkeley server. Such load balancing is implemented by DNS round robin, which achieves the best performance at high load.

Note that the simple Active Names policy is able to track the best performance of the two policies by accounting for distance and previous performance. At low load, both factors heavily bias Active Names toward the U.C. Berkeley replica. However, as load increases and performance at the U.C. Berkeley replica degrades, an increasing number of the requests are routed to the University of Washington achieving better overall performance.

Our algorithm for replica selection is not purported to be optimal. However, it demonstrates the utility of programmable replica selection, the importance of avoiding deterministic decisions in wide-area systems, and the benefits of inserting programmability at a high level in the protocol stack. It is straightforward for Active Name programs to calculate end-to-end performance (e.g., the time to retrieve a web page) because they interpose on both the location and retrieval of target resources. Calculating end-to-end performance in, for example, an Active Network environment that operates on individual packets is more difficult.

4.2 Location Independence

As discussed earlier, the proper way to present web content to a particular client depends upon its individual characteristics. For example, it makes little sense to transmit a 200 KB 1024x768 color image to a hand-held device with a 320x200 black and white screen behind a wireless link. To address this mismatch, the current approach [28] is to mediate client access through web proxies. These proxies retrieve requested resources and dynamically distill the content to match individual client characteristics, e.g., by shrinking a color image and converting it to black and white.

At a high level, clients name a web resource but would like the resource transformed based on client-specific characteristics. This model fits in well with Active Names. Clients specify the name of a resource (e.g., a URL retrieved by an HTTP namespace program) and an after-method that specifies the distillation program to be applied on the resource once it is located. The distillation program ensures that the object returned to the client will match its characteristics. A benefit of using Active Names to encapsulate distillation is the ability to flexibly place the transformation of a requested resource at arbitrary points in the network. For example, if the network path between a server and a proxy is congested, it may not make sense to transmit a large image over the congested network to perform a transformation that greatly reduces the size of the image. In a classic function versus data-shipping tradeoff, it is usually more efficient to perform the transformation at the server and then to transmit the smaller image to the proxy (or directly to the client). Conversely, if the transformation function is expensive, a fast network connection is available, and the server CPU is heavily loaded, then it is often more efficient to transmit the larger image to a proxy (or client) where more CPU cycles are available. Thus, the location-independent programs that comprise Active Names allow for flexible evaluation of function versus data shipping, trading off network bandwidth for computation time.

To demonstrate the above points, we implemented distillation within the Active Names framework and ran the following experiment to evaluate its utility. A client at U.C. Berkeley requests, through a local proxy, an image located at Duke University. This request is made under a number of different circumstances. The first variable is the place where distillation takes place. Active Name resolvers are available at both U.C. Berkeley and Duke so distillation can take place at either location. Three different policies are evaluated in choosing the distillation point: i) Statically assigning distillation at the proxy, the current approach to distillation, ii) Statically assigning distillation to the server, transmitting a smaller image across bottleneck wide-area links, and iii) an active approach where distillation is randomly assigned biased by estimates of end-to-end distillation cost at both the proxy and server sites.

In the active scheme, a Active Name after-method caches CPU load information at both the server and the proxy. Cache values are considered fresh for one minute. When cached load information expires, a separate thread is spawned to refresh cache information (the program responsible for maintaining load information, being an Active Name, can run either locally or remotely). The distillation Active Name program uses this load information, in addition to an estimate of the cost of unloaded distillation based on the size of the image to calculate distillation cost at both the server and the proxy. The program also calculates the cost of transmitting either the full or distilled image to the proxy to arrive at an end-to-end cost of distilling the image at the two locations. The distiller uses this information to bias a random selection of the location for distillation. Thus, if it is estimated that it will be twice as expensive to perform distillation at the proxy, the chance of performing proxy distillation will be one in three.

Another variable considered in our experiments is the load on the server machine at Duke. In one case, the Name Resolver at Duke University runs on an otherwise unloaded machine. In another, the resolver must compete with ten CPU intensive processes. The load on the server CPU will impact the placement of the distillation program. A third variable in our experiment is the dynamically changing available bandwidth between U.C. Berkeley and Duke (located at opposite ends of a continent). For this experiment, only the first two variables are modified. Available bandwidth is kept constant (as much as possible) by running the experiment late at night. In the future, we plan to investigate the use of SPAND [62] to estimate available bandwidth between two wide-area sites, and to use this information to more intelligently choose the location of distillation. The Active Name resolvers (including all distillation code) are compiled and run with the Java Development Kit, version 1.2 beta 4. The target image is 59 KB and is distilled to 9 KB. Distillation of the image consumes approximately 1 second of CPU time.

Figure 2 graphs the client-perceived latency of retrieving distilled versions of the target Jpeg image as a function of the number of clients simultaneously requesting the image from Duke for the three evaluated policies (static proxy, static server, active). Figure 2(a) shows performance in the case where the server is unloaded, while Figure 2(b) addresses a heavily loaded server, competing with ten CPU-intensive processes. Figure 2(a) shows that, at low levels of offered load (few simultaneous clients), unilaterally placing distillation at the server produces the best results because a smaller amount of data (9K versus 59K) is shipped across the wide area. However, as the number of clients increases, the server at Duke becomes overloaded and the performance degrades relative to the active policy that intelligently allocates distillation of a random percentage of the requests to the proxy. Figure 2(b) shows that, at low levels of offered load and high server CPU load, it is beneficial to place distillation at the proxy site. In this case, distillation at the server is an expensive enough operation to justify the larger long-haul transmission costs. However, as offered load increases, the active policy of splitting requests between the server and the proxy sites once again outperforms the static policy because the single processor at the proxy becomes overloaded.

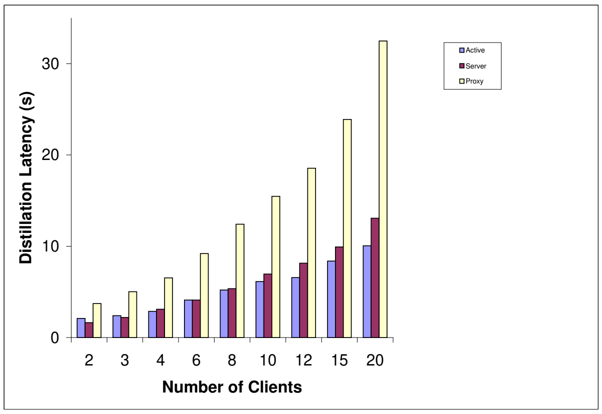
4.3 Composibility

A key design goal of Active Names is composibility. Not only should applications be able to inject extensions into the network, but they should be able to combine these extensions to provide new services and optimize existing ones. In this subsection, we first examine the costs composibility imposes on the system. We then study the benefits composable extensions bring to a key problem: web caching. Caching is a key technique for reducing both long-haul bandwidth and client-perceived latency. Table 1 breaks down the reason for web caching's relatively low hit rates that hang stubbornly near or below 50% [1, 23, 36, 7, 31]. This table suggests that composing different extensions may be a key technique for addressing the web caching problem. Moreover, these extensions are likely to be provided and implemented by a number of different entities, ranging from clients, to service providers, to third party software vendors. The experiments in this section demonstrate how the Active Name framework is utilized to compose multiple independent extensions, resulting in greater end-to-end performance gain than available from any single approach.

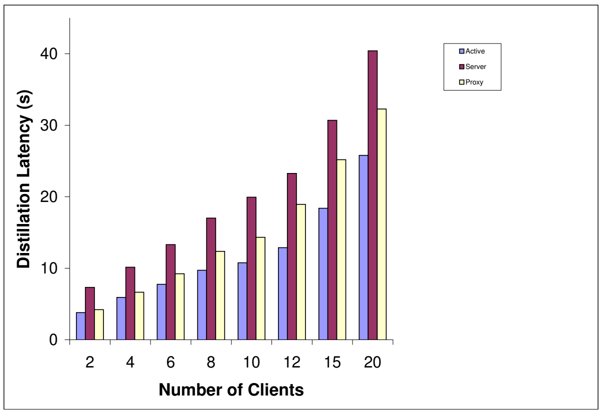
Our continuation passing architecture imposes the overhead of indirection through the "after methods" script when one Active Name program transfers control to another. On a Sun Ultra-10 with a 300MHz UltraSparc-II process running JDK1.2fcs, it takes 3.2 μs for one Active Name program to call another and return via this mechanism, compared to 0.2 μs if it were allowed to make the procedure call directly. Although this is more than an order of magnitude worse, the performance is sufficient to support the composition of relatively coarse-grained services such as we envision.

To determine whether compobility is worth this cost, we examine the composition of several server-initiated customizations with a client-initiated customization. The semantics of the sample service we implement are that when a client fetches a base page, the service (1) uses the server-side include interface to update the page for the current request, (2) randomly selects two out of eight candidate "banner ad" inline images, repeating the random selection on each viewing of the page, and (3) logs the cookies provided by and the ads selected for each request. We implement these server semantics in two ways. First, we construct them using standard mechanisms that run at the server: the server uses server-side includes to update the page and to execute a perl program that randomly selects advertisements to include on the page; we do not add additional logging to that already provided by the HTTP server. Second, we implement a version of the service by delegating a portion of the HTTP namespace to a set of three Active Name programs (provided by the service). The default HTTP namespace delegates control of all HTTP requests destined to `www.cs.utexas.edu/users/anonymized/service` to a "controller" Active Name program that alters the return path for such requests through a "ssi" Active Name program that provides server side-include semantics and through an "advertise" Active Name program that does ad rotation and logging. In the first configuration, all requests for the base page must go to the server; requests for the inline images may be cached. In the second configuration, once the delegation Active Name programs have been installed, both the base page and the inline images may use the cache because cached results will pass through the ssi and advertise Active Name programs before being returned to the client.

For the client-initiated customizations, clients use Active Names to customize their namespace to transcode images sent across a slow modem link. Because clients control their own namespace, adding this transformation to the pipeline is straightforward. The main subtlety is that clients cannot store the distilled images in the standard HTTP cache lest one client's mapping of the URL to the customized image



(a) Unloaded Server



(b) Loaded Server

Figure 2: Mobile Distillation Performance.

disturb other clients. Rather than cache such results in the HTTP namespace, the client caches such results in the "distiller" namespace instead.

Our experimental set up consists of three machines. The client, a 133 MHz Pentium machine running Microsoft NT3.5 and Sun JDK1.2beta3, communicates with the proxy, a 300MHz Sun UltraSPARC-II machine running Solaris 5.6 and JDK1.2fcs, over a 28.8 kbit/s modem. Both the client and proxy run the Active Name framework. The service being tested is hosted on a departmental web server running on a dual-processor Sun SPARCServer 1000e running Solaris 5.5.1 and the Netscape Enterprise Server 3.0(J) HTTP server. The proxy and server are connected by a department-wide switched 100 Mbit/s Ethernet.

The base page and its header are between 657 bytes and 2393 bytes (depending on where customization occurs) and the advertising banner images range in size from 8421 to 16928 bytes before distillation and from 2595 to 4365 bytes after distillation. The JAR files containing the server's controller, advertise, and ssi customization programs are 2622, 4700, and 3274 bytes, respectively. We begin the experiment with cold caches, except that we fetch two unrelated HTTP documents through the system to cause the JVMs to pre-load most of the basic classes associated with the system's standard HTTP data path, and we fetch two unrelated image files to cause the proxy to load the client's distiller Active Name program.

Figure 3 shows four cases representing the permutations of distillation (on/off) and server customization (on/off). Our client driver program uses the Active Name system running at the client to fetch the base document and then, using parallel connections, to fetch all inline images specified by the base document. After the driver receives each page and associated inline images, it pauses five seconds and repeats the process. The variation in response times from request to request is caused by cache hits and misses to the base page and the randomly selected inline images.

With respect to server customizations, there are three phases to consider. On the first request, no delegation has yet been specified to the client's Active Name system, so the *Server: on* performance closely matches the *Server: off* performance. Reacting to the delegation directive in the first request, the client's Active Name system spawns a background thread to download and install the specified customizations. This background thread is active during the second phase of the experiment—request two for the case when distillation is turned off and requests two and three when distillation is on. As a result, performance for the these requests is noticeably worse under server customization than for the standard case. In the third phase—after request three—the client has installed the server customization into its namespace and thus no longer needs to go to the server for ad rotation, hit logging, or SSI expansion. Performance is now significantly better under the customized version (modulo cache hits to the inline images). For example, as Figure 3-a indicates, after the cache is warm and when the inline images are hits, the *Server: on* case provides response times under 0.26 s while the other case requires over 1.3 s per request. In this situation, the Active Names system provides a 5-fold performance improvement. This result is particularly significant in light of human factors studies that suggest that driving computer response time from about a second to significantly less than a second may result in more than a linear increase in user productivity as the system becomes truly interactive [43, 12].

Figure 3 also shows that distillation significantly improves performance for the initial series of requests, and makes little difference once the images are cached at the client. For example, when server customization is turned on, the five most expensive requests require an average of

Source of Miss	Fraction of Requests	Available Approaches	Client/Server Initiated
Compulsory	19%-30%[64] 45%(*ISP*)	Prefetching [37, 47, 59] Server replication [66] or push caching [40] Increase number of clients sharing cache system [13, 23, 36, 64] Transcoding [28, 3], compression and delta-encoding[53, 8]	either server client client or either
Consistency verify (unmodified)	10%(*ISP*) 2-7%[23] 4-13%[7]	Server-driven consistency [50, 46, 73]	server
Consistency miss	0-4%[23]	delta-encoding[53, 8]	either
Dynamic (cgi or query)	21%(*ISP*) 0-34%[51]	Active cache [14], HPP [22] TREC [65]	server server
Pragma: no-cache	9%(*ISP*) 5.7-7.2%[36]	Hit logging Active cache [14], function-shipping Server-driven consistency [46, 73]	server server server
Redirection	3.7%(*ISP*)	Server selection/anycast [74, 11]	server

Table 1: Workload requirements. Numbers are taken from the literature as noted or from our trace of a large ISP that serves seven million requests containing 65.4GB to 23080 clients over a six-day period .

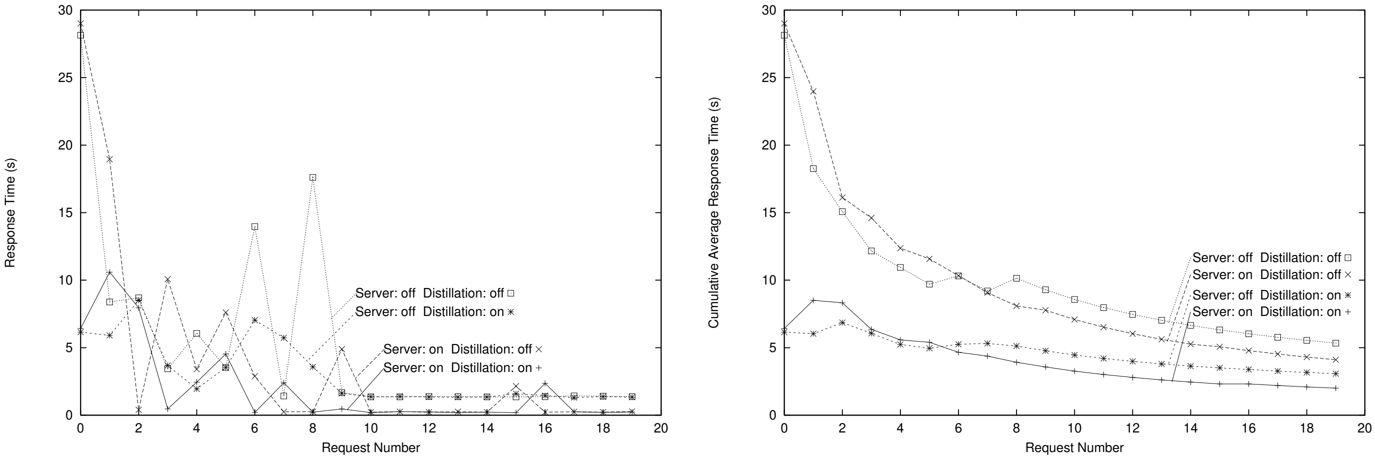


Figure 3: (a) Per-request and (b) cumulative average performance of customizations of the HTTP namespace.

15.3 s without distillation, whereas the five most expensive requests averaged 6.67 s under distillation, a speedup of 2.3. Without server customization, the five most expensive requests average 14.1 s and 6.39 s when distillation is off or on, for a speedup of 2.2 for distillation.

Other researchers have noted the advantage of server-controlled caching [14] and distillation [28]. The above experiment suggests that a combination of server and client customizations may be particularly effective. On average for the 20-request sequence, the combination outperforms the distillation-only case by 50% and it outperforms the server-only case by 104%.

5 Related Work

As discussed in Section 2, Active Names are inspired by and provide an alternative to related research in Active Networks and Active Services. Also closely related to Active Names are Transaction Processing monitors [34] (TP monitors). TP monitors provide functionality similar to Active Names for access to databases. The TP monitor directs transactions to *resource managers*, accounting for load on machines, the RPC program number, and any affinity between client and server. Resource managers are usually SQL databases, but can be any server that supports transactions. While Active Names and TP monitors target a number of similar applications, Active Names provides a more general environment for programmable access to wide-area resources. In contrast, TP monitors tend to be more static and more closely associated with the service.

Current wide-area computing research proposals, such as Globe [67], Globus [25], and Legion [38], propose a number of schemes for locating computational resources across the wide area. These proposals are orthogonal to our work as any of them could be incorporated within the extensible Active Names framework. Ninja [35] proposes a pipelined data flow model for composing services in a clustered and/or mobile environment. While this model is attractive, it does not to date address dynamic migration of computation, as we have demonstrated is crucial for performance. A recent proposal [20] for implementing URN's advocates leveraging DNS and rewriting of names through regular-expression matching in an iterative manner to locate wide-area resources. This scheme could also be implemented more generally within the Active Names framework with namespace programs responsible for name rewriting and namespace delegation.

Prospero [54, 55] also supports extensible naming to support mobile hosts and the integration of multiple wide-area information services (e.g., WAIS and gopher). Prospero allows users to customize their own namespaces, grouping related information (from an individual's perspective) together. However, customization code runs on the client. Relative to Prospero, our work demonstrates the utility of location-independent and portable programs for name resolution. Programmability in Active Names is similar to Smart Clients [74]. Smart Clients retrieve service-specific code into the client to mediate access to a set of server replicas. Active Names are more general than Smart Clients, with location independent code able to run anywhere in the system allowing for the deployment of a broader range of applications.

Anycasts [11, 24], Nomenclator [57], and Query Routing [48] also allow for resource discovery and replica selection. Anycasts allow a name to be bound to multiple servers, with any single request transmitted to a single replica according to policy in routers or end hosts. Nomenclator uses replicated catalogs with distributed indices to locate wide-area resources. The system also integrates data from multiple repositories for heterogeneous query processing. Query Routing uses compressed indexes of multiple resources and sites to route requests to the proper destination. These approaches show promising results and should, once again, fit well within our extensible framework.

Active Caches [14] allow for customization of cache content through Java programs similar to our extensible cache management system. With Active Caches however, retrieved data files contain programs, with the cache promising to execute the program (which may change the contents of the file) before returning the data to the client. On the other hand, our extensible cache management system uses service-specific programs to mediate all accesses to a service. This approach is more general allowing, for example, the program to manage local cache replacement policy or to perform load balancing on a cache miss.

6 Conclusion and Future Work

This paper describes a framework supporting extensibility for wide-area distributed services through the introduction of location-independent programs that interpose on the naming interface. These Active Names can, for example, customize how a service is located and how its results are transformed and transported back to the client. Our approach is compared to existing schemes for introducing programmability into the network such as Active Networks and Active Services. The paper then describes the implementation of the Active Name prototype and illustrates its utility through a number of sample services, including replicated service selection and mobile distillation of service content. In each case, end-to-end application performance information is leveraged to match or exceed existing static approaches. The need for composibility is illustrated through Internet service access that incorporates extensions from multiple sources. Our results show that Active Name extensions can offer up to a five-fold performance improvement relative to existing static approaches, and that it is necessary to compose multiple extensions to achieve this benefit: no single extension achieves comparable performance.

References

[1] M. Abrams, C. Standridge, G. Abdulla, S. Williams, and E. Fox. Caching Proxies: Limitations and Potentials. In *Proceedings of 1995 World Wide Web Conference*, 1995.

[2] D. Scott Alexander, William A. Arbaugh, Angelos D. Keromytis, and Jonathan M. Smith. Safety and Security of Programmable Network Infrastructures. *IEEE Communications Magazine*, 36(10):84–92, 1998.

[3] Elan Amir, Steven McCanne, and Randy Katz. An Active Service Framework and its Application to Real-Time Multimedia Transcoding. In *Proceedings of SIGCOMM*, September 1998.

[4] Eric Anderson, David Patterson, and Eric Brewer. The Magicrouter, an Application of Fast Packet Interposing. See http://-HTTP.CS.Berkeley.EDU/˜eanders/magicrouter/, May 1996.

[5] Thomas E. Anderson, David E. Culler, David A. Patterson, and the NOW Team. A Case for NOW (Networks of Workstations). *IEEE Micro*, February 1995.

[6] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless Network File Systems. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 109–126, December 1995.

[7] Martin F. Arlitt and Carey L. Williamson. Web Server Workload Characterization: The Search for Invariants. In *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 126–137, May 1996.

[8] Gaurav Banga, Fred Douglis, and Michael Rabinovich. Optimistic Deltas for WWW Latency Reduction. In *Proceedings of the USENIX 1997 Technical Conference*, December 1997.

[9] Eshwar Belani, Amin Vahdat, Thomas Anderson, and Michael Dahlin. The CRISIS Wide Area Security Architecture. In *Proceedings of the USENIX Security Symposium*, San Antonio, Texas, January 1998.

[10] Tim Berners-Lee. Hypertext Transfer Protocol HTTP/1.0, October 1995. HTTP Working Group Internet Draft.

[11] S. Bhattarcharjee, M. Ammar, E. Zegura, V. Sha, and Z. Fei. Application-Layer Anycasting. In *Proceedings of IEEE Infocom*, 1997.

[12] J.T. Brady. A Theory of Productivity in the Creative Process. In *IEEE CG&A*, May 1986.

[13] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. On the Implications of Zipf's Law for Web Caching. Technical Report 1371, University of Wisconsin, April 1998.

[14] Pei Cao, Jin Zhang, and Kevin Beach. Active Cache: Caching Dynamic Contents on the Web. In *Proceedings of Middleware*, 1998.

[15] David Cheriton and Dale Skeen. Understanding the Limits of Causally and Totally Ordered Communication. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 44–57, December 1995.

[16] Stuart Cheshire and Mary Baker. Internet Mobility 4x4. In *Proceedings of the ACM SIGCOMM'96 Conference*, August 1996.

[17] Cisco. Distributed Director. `http://www.cisco.com/warp/public/751/distdir/technical.shtml`, 1997.

[18] CNN. CNN Custom News. `http://customnews.cnn.com`, 1998.

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

[20] Ron Daniel and Michael Mealling . Internet Draft. "resolution of uniform resource identifiers using the domain name system". Internet Draft, see `http://www.acl.lanl.gov/URN/naptr.txt`, May 1997.

[21] M. DeMoney, J. Moore, and J. Mashey. Operating System Support on a RISC. In *Proceedings of the 31st Computer Society International Conference (Spring Compcon '86)*, pages 138–143, March 1986.

[22] Fred Douglis, Antonio Haro, and Michael Rabinovich. HPP: HTML Macro-Preprocessing to Support Dynamic Document Caching. In *Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems*, Monterey, California, December 1997.

[23] Brad Duska, David Marwood, and Michael J. Feeley. The Measured Access Characteristics of World Wide Web Client Proxy Caches. In *Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems*, Monterey, California, December 1997.

[24] Zongming Fei, Samrat Bhattacharjee, Ellen W. Zegura, and Mostafa Ammar. A Novel Server Selection Technique for Improving the Response Time of a Replicated Service. In *Proceedings of IEEE Infocom*, July 1998.

[25] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke. A Directory Service for Configuring High-Performance Distributed Computations. In *Proc. 6th IEEE Symp. on High-Performance Distributed Computing*, pages 365–376, 1997.

[26] Marc E. Fiuczynski, Vincent K. Lam, and Brian N. Bershad. The Design and Implementation of an IPv6/IPv4 Network Address and Protocol Translator. In *Proceedings of the 1998 USENIX Conference*, June 1998.

[27] Martin Fouts, Tim Connors, Steve Hoyle, Bart Sears, Tim Sullivan, and John Wilkes. Brevix design 1.01. Technical Report HPL-OSR-93-22, HP Laboratories, April 1993.

[28] Armando Fox, Steven Gribble, Yatin Chawathe, and Eric Brewer. Cluster-Based Scalable Network Services. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, Saint-Malo, France, October 1997.

[29] Douglas P. Ghormley, Steven H. Rodrigues, David Petrou, and Thomas E. Anderson. SLIC: An Extensibility System for Commodity Operating Systems. In *USENIX 1998 Annual Technical Conference*, June 1998.

[30] Garth Gibson, David Nagle, Khalil Amiri, Jeff Butler, Fay Chang, Howard Gobioff, Charles Hardin, Erich Reidel, David Rochberg, and Jim Zelenka. A Cost-Effective, High-Bandwidth Storage Architecture. In *8th ASPLOS*, 1998.

[31] Steve Glassman. A Caching Relay for the World Wide Web. In *First International World Wide Web Conference*, 1994.

[32] James Gosling and Henry McGilton. The Java(tm) Language Environment: A White Paper. `http://java.dimensionx.com/whitePaper/java-whitepaper-1.html`, 1995.

[33] R. Govindan, C. Alaettinoglo, and D. Estrin. A Framework for Active Distributed Services. Technical Report 98-669, Information Sciences Institute, University of Southern California, 1998.

[34] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[35] Steve Gribble, Matt Welsh, Eric Brewer, and David Culler. The MultiSpace: an Evolutionary Platform for Infrastructural Services. In *Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems*, June 1999.

[36] Steven D. Gribble and Eric A. Brewer. System Design Issues for Internet Middleware Services: Deductions from a Large Client Trace. In *Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems*, Monterey, California, December 1997.

[37] J. Griffioen and R. Appleton. Automatic Prefetching in a WAN. In *IEEE Workshop on Advances in Parallel and Distributed Systems*, October 1993.

[38] A. Grimshaw, A. Nguyen-Tuong, and W. Wulf. Campus-Wide Computing: Results Using Legion at the University of Virginia. Technical Report CS-95-19, University of Virginia, March 1995.

[39] Rajarshi Gupta. WebTP: A User-Centric Receiver-Driven Web Transport Protocol. Master's thesis, University of California, Berkeley, December 1998.

[40] James Gwertzman and Margo Seltzer. World-Wide Web Cache Consistency. In *Proceedings of the 1996 USENIX Technical Conference*, pages 141–151, January 1996.

[41] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A Packet Language for Active Networks. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming Languages*, pages 86–93, 1998.

[42] Norm C. Hutchinson and Larry L. Peterson. The x-Kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[43] IBM. *The Economic Value of Rapid Response Time*, pages 11–82. Number GE20-0752-0. White Plains, N.Y., 1982.

[44] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hctor M. Briceo, Russell Hunt, David Mazires, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application Performance and Flexibility on Exokernel Systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, October 1997.

[45] Eric Dean Katz, Michelle Butler, and Robert McGrath. A Scalable HTTP Server: The NCSA Prototype. In *First International Conference on the World-Wide Web*, April 1994.

[46] Anne-Marie Kermarrec, Ihor Kuz, Maarten van Steen, , and Andrew S. Tanenbaum. A Framework for Consistent, Replicated Web Objects. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, May 1998.

[47] T. Kroeger, D. Long, and J. Mogul. Exploring the Bounds of Web Latency Reduction from Caching and Prefetching. In *Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems*, December 1997.

[48] P. Leach and C. Weider. Query Routing: Applying Systems Thinking to Internet Search. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems*, pages 82–86, Cape Code, MA, 1997.

[49] Jacob Y. Levy, Laurent Demailly, John Ousterhout, and Brent B. Welch. The Safe-Tcl Security Model. In *Proceedings of the 1998 USENIX Conference*, June 1998.

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

[51] S. Manley and M. Seltzer. Web Facts and Fantasy. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, 1997.

[52] P. Mockapetris and K. Dunlap. Development of the Domain Name System. In *Proceedings SIGCOMM 88*, April 1988.

[53] J. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential Benefits of Delta Encoding and Data Compression for HTTP. Technical report, Digital/Compaq Western Research Lab, December 1997.

[54] B. Clifford Neuman. Prospero: A Tool for Organizing Internet Resources. In *Electronic Networking: Research, Applications and Policy*, pages 30–37, Spring 1992.

[55] B. Clifford Neuman, Steven Seger Augart, and Shantaprasad Upasani. Using Prospero to Support Integrated Location Independent Computing. In *Proceedings of the Symposium on Mobile and Location Independent Computing*, August 1993.

[56] H. Nielsen, J. Gettys, A. Baird-Smith, E. Prud'hommeaux, H. Lie, and C. Lilley. Network Performance Effects of HTTP/1.1, CSS1, and PNG. In *Proceedings of SIGCOMM '97*, September 1997.

[57] Joann Ordille and Barton P. Miller. Distributed Active Catalogs and Meta-Data Caching in Descriptive Name Services. In *IEEE International Conference on Distributed Computing Systems*, pages 120–129, May 1993.

[58] John Ousterhout. *CMU Computer Science: A 25th Anniversary Commemorative*, chapter The Role of Distributed State. ACM Press, 1991.

[59] V. Padmanabhan and J. Mogul. Using Predictive Prefetching to Improve World Wide Web Latency. In *Proceedings of the ACM SIGCOMM '96 Conference on Communications Architectures and Protocols*, pages 22–36, July 1996.

[60] David A. Patterson, Garth Gibson, and Randy H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM Conference on Management of Data (SIGMOD)*, pages 109–116, Chicago, IL, June 1988.

[61] D.M. Ritchie and K. Thompson. The Unix Time-Sharing System. *Communications of the ACM*, 17(7):365–375, July 1974.

[62] Srinivasan Seshan, Mark Stemm, and Randy H. Katz. SPAND: Shared Passive Network Performance Discovery. In *Proc. 1st Usenix Symposium on Internet Technologies and Systems (USITS '97)*, December 1997.

[63] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 172–183, December 1995.

[64] R. Tewari, M. Dahlin, H. Vin, and J. Kay. Design Considerations for Distributed Caching on the Internet. In *Proceedings of the Twentieth International Conference on Distributed Computing Systems*, May 1999.

[65] Amin Vahdat and Thomas Anderson. Transparent Result Caching. In *Proceedings of the 1998 USENIX Technical Conference*, New Orleans, Louisiana, June 1998.

[66] Amin Vahdat, Thomas Anderson, Michael Dahlin, Eshwar Belani, David Culler, Paul Eastham, and Chad Yoshikawa. WebOS: Operating System Services for Wide-Area Applications. In *Proceedings of the Seventh IEEE Symposium on High Performance Distributed Systems*, Chicago, Illinois, July 1998.

[67] M. van Steen, F.J. Hauck, P. Homburg, and A.S. Tanenbaum. Locating Objects in Wide-Area Systems. In *IEEE Communications Magazine*, pages 104–109, January 1998.

[68] Robert Wahbe, Steven Lucco, Thomas Anderson, and Susan Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 203–216, December 1993.

[69] Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten. Extensible Security Architectures for Java. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 116–128, Saint-Malo, France, October 1997.

[70] David Wetherall, Ulana Legedza, and John Guttag. Introducing New Network Services: Why and How. In *IEEE Network Magazine, Special Issue on Active and Programmable Networks*, July 1998.

[71] Yahoo. My Yahoo. `http://my.yahoo.com`, 1996.

[72] Y. Yemini and S. da Silva. Towards Programmable Networks. In *IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, October 1996.

[73] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Using Leases to Support Server-Driven Consistency in Large-Scale Systems. In *Proceedings of the Nineteenth International Conference on Distributed Computing Systems*, May 1998.

[74] Chad Yoshikawa, Brent Chun, Paul Eastham, Amin Vahdat, Thomas Anderson, and David Culler. Using Smart Clients to Build Scalable Services. In *Proceedings of the USENIX Technical Conference*, January 1997.