Active Naming: Programmable Location and Transport of Wide-Area Resources*

Amin Vahdat
Computer Science Division
University of California, Berkeley

Thomas Anderson
Department of Computer Science and Engineering
University of Washington, Seattle

Michael Dahlin
Department of Computer Science
University of Texas, Austin

Abstract

Active Names are a general framework for the development and composition of wide-area applications. The key insight behind Active Names is the need to introduce programmability of name binding to support the widely varying semantics and requirements of distributed applications in accessing and transforming wide-area resources. Active Names associate an application-specific and location independent program with each name. This paper presents the architecture for evaluating Active Names, including the technique for binding names to programs, and a virtual machine for executing these programs. We demonstrate the utility of the Active Naming framework via four sample applications: mobile distillation of Web content, load balancing and replica selection across wide-area servers, extensible cache management designed to increase the utility of Web caches, and personalization of Web content to client preferences.

1 Introduction

The emergence of wide-area distributed applications as a dominant computing paradigm highlights two major limitations in today's computing infrastructure. First, wide-area applications suffer performance problems in attempting to take advantage of the rich set of available network resources. The Internet today is marked by highly variable and unpredictable performance both in end hosts and the network interconnect. The second major limitation is inflexibility within the infrastructure itself, making it difficult to deploy new network services and applications. For example, a number of schemes for replication and load balancing [Katz et al. 1994, Berners-Lee 1995, Brisco 1995], caching [Gwertzman & Seltzer 1996, Chankhunthod et al. 1996, Zhang et al. 1997, Cao et al. 1998, Tewari et al. 1998], and customization (e.g., Yahoo [Yahoo 1996] or CNN [CNN 1998]) have been proposed recently. While point solutions to point problems have been proposed, some have enjoyed only limited acceptance because the Internet is missing a general infrastructure to develop and deploy new network services.

In this paper, we present the design and implementation of Active Names, a general mechanism to support performance and flexibility in the deployment of network services. An Active Name extends the traditional name/binding translation by executing a program to directly retrieve a resource (not just its binding). Interposing a program on the naming interface provides a natural way to express semantics desired by wide-area network services. For example, code for an Active Name can (i) manage server replicas, choosing the least loaded, closest server to a particular client (ii) utilize application-specific coherence policies to locate caches with fresh entries, and (iii) store user preferences for page contents (e.g., specific headlines, stock quotes, weather reports). As an added advantage, the code for an Active Name is location independent, meaning that it can run at various points in the network to exploit, for example, locality, available computation, and/or available bandwidth.

Active Names are motivated by the same goals of flexibility and performance as Active Networks [Tennenhouse & Wetherall 1996, Wetherall et al. 1998]. Active Networks interpose a program at the IP packet level, introducing programmability into network routers. Such functionality is well matched to introducing new network transport mechanisms, such as RSVP [Zhang et al. 1993], multicast [Deering & Cheriton 1990], or Mobile IP [Perkins 1996]. While there is certainly overlap between functionality provided by Active Names and Active Networks, we will argue that interposing at the naming interface is more convenient for building and deploying higher-level services such as caching, managing failover, replication, and customization.

The principal contribution of Active Naming is a unified framework for extensible and application-specific naming, location, and transformation of wide-area resources. Specif-

*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), 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). For more information, please see `http://now.cs.berkeley.edu/WebOS`.

ically, the goals of the system are to: (i) conveniently express in a single unified framework a wide variety of existing approaches to replication, load balancing, customization, and caching, (ii) allow simple implementation and deployment of new naming extensions as they become necessary and available, (iii) compose any number of Active Name extensions into a single request (as opposed to the more ad-hoc, often mutually exclusive, nature of existing techniques), and (iv) minimize latency and consumed wide-area bandwidth in evaluating Active Names.

To demonstrate the utility of Active Naming we envision the following sample applications within the Active Name prototype; the first four have been prototyped and are described in detail in Section 4.

- *Mobile Distillation*: Distillation [Fox et al. 1996, Fox et al. 1997] transforms wide-area resources to better match client characteristics. For example, a Palm Pilot connected to the Internet over a modem should only retrieve small black and white versions of images to match its screen size and available bandwidth. The current approach is to apply the transformation at a fixed location in the network (the client proxy). An Active Name specifies the operation used to retrieve the image; we demonstrate the utility of flexibly setting the transformation point at variable points in the network. For example, when wide-area bandwidth is limited, transforming an object at the server is more efficient than transmitting a large image to a proxy before reducing the image size.

- *Extensible Cache Management*: A significant portion of Web content cannot currently be cached because servers wish to maintain control over each access. For example, they may wish to track hit counts for advertising revenues, or perform advertising banner rotation or other modifications for each client access [Cao et al. 1998]. Active Names run service-specific code to manage portions of proxy caches. Thus, for every access to a particular service's content in a cache, code specific to that service is run locally to perform any bookkeeping or client-specific customization on behalf of the service.

- *Replication and Load Balancing*: Many Web services are replicated across geographically distributed sites. The proper algorithm for choosing among replicas is service and client specific, as well as adaptive to changes in network/server capacity and workload. Furthermore, clients must also perform failover in a service-specific manner (e.g., maintaining enough state to retransmit a request to an alternate site). Active Names provide the requisite flexibility to direct requests to the replica likely to deliver the best performance.

- *Customization*: Many services are customizing their presentation to individual user preferences. For example, the URL `http://my.yahoo.com` returns a news page customized with the headlines, stock quotes, weather forecasts, etc. of a particular user. With Active Names, a program maintains user preferences and performs customization on behalf of services.

- *Uniform Resource Names*: URN's [Sollins & Masinter 1994] allow location independent names for web resources. URN's present a number of advantages the current URL approach. Perhaps most important is the ability to make object names persistent. Thus, as objects are replicated or moved from host to host or directory to directory, the name remains constant (i.e., once a name is generated it will never become invalid). The current proposal for implementing URN's [Daniel & Mealling 1996] leverages DNS to invoke complex and static re-writing rules of names to translate the name to an address. While we have not yet implemented a prototype URN system, Active Naming would allow for service-specific code to maintain mappings between URN's and the current location of all objects maintained by a service. Of course, the specific name translation policy is entirely under service control.

- *Global Location-Independent Objects*: A number of research projects [Grimshaw et al. 1995, van Steen et al. 1998] envision a wide-area object based computation system. Locating objects in such a system in the face of replication, mobility, and cache consistency in the general case is difficult. However, the problem becomes easier by using application-specific knowledge for object location. Such a model once again fits in well with Active Name model, as Active Name programs can employ application-specific knowledge (e.g., hints as to which regional area an object may be located [van Steen et al. 1998]) to locate objects across the wide area.

While versions of each of these applications have been implemented in other contexts, the existing implementations are limited by the lack of a general framework for flexibility, deployment, composability, and resource allocation.

The rest of this paper is organized as follows. Section 2 further motivates our design rationale for Active Names with a discussion of the need for performance and flexibility in network services. Section 3 presents the Active Naming architecture and implementation. Section 4 describes the implementation of the four sample Active Naming applications. Section 5 presents related work and we conclude in Section 6.

2 Motivation

As the popularity of wide-area distributed computing grows with the popularity of the Internet, wide-scale replication of distributed services has emerged as the key technique for scaling to user and program demands. Recent proposals [Grimshaw et al. 1995, Foster & Kesselman 1996, van Steen et al. 1997, Vahdat et al. 1998] suggest that replication of distributed systems will likely become both more dynamic, with automatic replication and tear-down of popular objects, and wide-scale, with some systems scaling to billions of objects. We currently see this trend on a smaller scale, with popular services such as Alta Vista [Dig 1995] and Netscape [Net 1994] replicating their web servers across the wide area for improved scalability, performance, and availability. Interestingly, the performance and availability bottleneck for many services is not local CPU or disk performance. While NOW's [Anderson et al. 1995] and RAID's [Chen et al. 1993] allow for highly available, scalable performance [Brewer 1997], recent studies suggest that the performance limitations often lie in network performance [Paxson 1996].

Replication makes naming more difficult because the name for a service can translate to multiple addresses spread across the wide area. To choose the proper replica the name system must account for current network connectivity along multiple paths, the location of the client making the request, and the processing power and relative load on the replicas. For example, the name server must balance relative load with Internet delay; it can be better to go to lightly-loaded, but distant servers as opposed to a heavily-loaded server on the same LAN. Similarly, two clients at different locations may choose different replicas independent of load because of backbone congestion between a client and replica. As another example, network performance is highly variable meaning that the proper replica for access to a large file (e.g., the latest version of Netscape's Communicator) is likely to change during the course of the access. The current model of binding to an IP address then accessing a resource at that address makes it very difficult to switch replicas midstream (similar problems arise in implementing mobile IP [Ioannidis & Maguire 1993]). Similarly, performing failover on replica failure must be programmable and service-specific. For example, failure semantics vary for Internet chat [Yoshikawa et al. 1997] versus a Bayou managed calendar program [Terry et al. 1995].

Flexibility is important for applications beyond replication and load balancing. Application-specific knowledge can often be used to perform name translation in multiple domains. For example, in the world-wide mobile object application discussed above, the knowledge that mobile users exhibit strong geographic locality can be used to construct a hierarchical regional name lookup system with hints [van Steen et al. 1998]. In this way, mobile objects are likely to be found with a small number of lookups if an object is first bound to a home region. Similar application-specific knowledge can be used to implement efficient name lookup in other domains, such as: hostname to IP address translation [Mockapetris & Dunlap 1988], locating participants of a parallel program [Fitzgerald et al. 1997], or locating an object in a proxy cache [Zhang et al. 1997].

Another advantage of Active Naming is the potential to reduce wide-area communication and improve client latency. Currently, naming is usually a step in a larger process and involves multiple round-trip communications. For example, clients bind to a stub before generating an RPC and browsers translate hostnames to IP addresses through DNS before retrieving Web pages[1]. Active Naming makes use of three-way RPC's to minimize the need for multiple round-trip communications in name resolution. Thus, a request is routed along a path to where it can be satisfied, and the result is transmitted directly back to the client. For example, utilizing hierarchical Web caches can have a negative impact on client-perceived latency because upon locating a page several levels up in a hierarchy the page must travel back down the hierarchy before being transmitted to the client. With three-way RPC's, the web page, once found, is transmitted directly back to the client, with caches back-filled in the background.

Active Names also provide a convenient way to express tradeoffs in function vs. data shipping. Depending on tradeoffs in available bandwidth and local computation power, name resolution can involve either shipping an evaluation function to the data or retrieving data for local evaluation. For example, consider the distillation of a large 100 KB image. In the case where server CPU cycles are available and wide-area bandwidth is scarce, it is more efficient to ship the distillation function to the data at the server. However, if the situation reverses and the server becomes overloaded and bandwidth becomes plentiful, retrieving the data across the network and performing distillation is likely more efficient. Once again, the key observation is that this decision must be application-specific and adaptive to changes in capacity and workload.

3 Architecture and Implementation

This section describes the design and implementation of a name resolution system supporting extensibility and composability. At a high level, our system provides a framework for binding names to programs and a framework for chains of programs to cooperate in translating a name to the data that name represents. Thus, a name lookup consults a distributed name database to retrieve a program. This program

[1] Hostname to IP translation may involve multiple round trips through the DNS hierarchy. One recent study [Thompson et al. 1997] showed that 5-10% of backbone traffic consisted of DNS flows.

owns a portion of the global namespace (e.g., CNN) and has the freedom to bind sub-names (e.g., CNN/frontpage) in a service-specific manner. We also provide a virtual machine to export a standard environment for the execution of Active Name code. This virtual machine provides, for example, access to local cache state and also enforces security and resource limitations. In designing these interfaces, we focus on the twin (and often contradictory) goals of simplifying the implementation of the code bound to an active name, and of allowing the maximum flexibility with respect to performance optimizations.

Figure 1 summarizes the interface to the classes that make up the Active Name architecture. The system is built in Java to leverage a number of system features, such as portability, object serialization, remote method invocation, and stack introspection [Wallach & Felten 1998]. Three main entities make up the system: ActiveNames, NamespacePrograms, and VirtualMachines. An Active Name is made up of two strings. The first is *namespace* which identifies a domain. Service-specific code (a *NamespaceProgram*) exists to evaluate the second portion of an ActiveName, the *name* within a given namespace. To evaluate an Active Name client invokes the `Eval` method of a resolver's VirtualMachine with three arguments. The first argument is the Active Name to be evaluated. The second argument is an array of Active Names called *afterMethods*. The programs associated with this array of Active Names are guaranteed to be called, in order, after the original program completes its evaluation, to support a distributed continuation-passing style of execution. The third argument, *DataStream*, represents the result of evaluating an Active Name and acts as input to after methods. In effect, the DataStream forms a pipeline of filters passed from program to program and, potentially, host to host.

Given this basic architecture, we envision Active Name programs adhering to several conventions. First, the last after method is a program that transmits the final result to the client. This convention allows for the construction of three-way RPC's. Because Active Name programs are location-independent and completely encapsulate the evaluation state, the programs can be routed directly to hosts best able to evaluate a given name. Results can then be transmitted directly to the client via the last after method, conserving bandwidth and latency by avoiding the need to retrace the route taken to evaluate a name. A similar convention is that the array of after methods acts as a stack, with each successive Active Name program able to push additional after methods onto the stack.

A third convention concerns failure recovery. Instances of `NamespaceProgram` maintain state on current replica membership. This information, in addition to service-specific state, is used to perform failover in the case where an error is returned or a response is not received after a given timeout period. For backward compatibility, Active Name resolvers can act as a web proxy that understand client HTTP requests, providing a bridge between the current model of naming through URL's/DNS and Active Names. Of course, more sophisticated/demanding programs may violate any of these conventions. For example, a query optimizer may reorder the stack of after methods for better locality or an anonymizer may replace the last after method to hide the identity of a client making a request.

3.1 Dynamic Code Location

All Active Names are implicitly associated with a Java class responsible for evaluating the name and retrieving the associated data. As summarized in Figure 1, the namespace components of an Active Name identifies a Java class that extends the base class `NamespaceProgram`. These subclasses are responsible for translating Active Names to the resources they represent. For example, the class `CNNNamespaceProgram` might be responsible for mediating access to resources provided by the CNN news service.

`NamespaceProgram` classes are instantiated and executed by Active Name resolvers. The primary responsibility of the resolver is to convert a string representation of an Active Name to the class responsible for evaluating the name. Once located, the resolver instantiates the class in a separate thread and invokes the `Eval` method. This method is responsible for transmitting the resource represented by the Active Name back to the client. The Active Name Resolver enforces security and resource limitations on running `NamespaceProgram` classes. Since downloaded code is generally untrusted, it is the responsibility of the resolver to ensure that the code accesses only authorized state (e.g., no access to other thread's state or file data) and does not exceed certain resource limitations (e.g., the thread cannot allocate all available physical memory).

By convention, every Active Name passed to the name resolver matches the class `UberNamespaceProgram`. The code for this class is available at every name resolver and it bootstraps the location and instantiation of the "proper" class necessary to evaluate a given Active Name. For example, an active name that matches the format of a URL will cause loading and execution of the class `UrlNamespaceProgram`. While the assumed format of Active Names is easy to change (by adding or modifying code in the class `UberNamespaceProgram`), the current format for non-URL names is:

```
namespace/name
```

Examples of Active Names include `CNN/frontpage` and `"www.news.com/code/NewsNamespaceProgram.jar"/frontpage`. The string preceding the first slash (or in quotes) specifies the class that should be run to evaluate the ActiveName. As the examples indicate, the namespace can be identified in

```
class ActiveName {
  public ActiveName(String namespace, String name);
  public String GetNamespace();
  public String GetName();
}

abstract class NamespaceProgram { // The code for evaluating an Active Name
  public void Eval(String name, ActiveName[] afterMethods, DataStream data);
  public VirtualMachine GetVirtualMachine();
}

class VirtualMachine {
  public Cache GetCache();
  public NetworkDataStream MakeNetworkConnection(String host, int port);
  public void Eval(ActiveName name, ActiveName[] afterMethods, DataStream data);
  public void EvalAt(NamespaceProgram program, String remoteHost); // For Remote Method Invocation
}
```

Figure 1: This figure presents an outline of the interfaces for the Java classes that make up the Active Naming prototype.

either a shorthand or a fully qualified format. The shorthand format is for user convenience; in these cases, a registry[2] is consulted for code registered to the given shorthand. On the other hand, the fully-qualified namespace identifies the exact piece of code to be retrieved in a URL-like format (in fact, we currently employ HTTP to retrieve the code). The second portion of an Active Name, the name, is opaque to the system and is handed to the `Eval` method of `Namespaceprogram` unmodified.

Active Name resolvers cache the classes responsible for evaluating Active Names to avoid the above bootstrapping process in the general case. Classes are cached for a one month period by default. Of course, each individual class can implement its own coherence policies, retrieving newer versions of itself on a finer granularity. Classes can also serialize their content to the local disk to maintain state information that may be beneficial on subsequent invocation of a particular class.

3.2 Virtual Machine

3.2.1 Interface

Active Name Resolvers export a virtual machine to `NamespaceProgram` classes. Resolvers spawn a separate thread with an associated virtual machine for the evaluation of each Active Name. The virtual machine gates accesses to all resources through a Java Security Manager and by enforcing resource limitations (as described in 3.2.2 and 3.2.3 below). The virtual machine is also used to insert and retrieve data from a local cache and to make network connections.

[2]This shorthand registry is not implemented but one simple way to construct it would be to leverage DNS.

The interaction of the virtual machine exported by the Resolver and classes responsible for Active Name evaluation is described in Figure 1 `VirtualMachine.Eval` locates the class responsible for evaluating an Active Name, sets up a virtual machine to enforce security and resource consumption limits, and spawns a new thread to run the method `NamespaceProgram.Eval`. `GetCache` is used to retrieve the local cache provided by the Active Name Resolver. Namespace Programs can request individual partitions within the cache to locally store and retrieve objects. `MakeNetworkConnection` is used to communicate with remote machines. Finally, `EvalAt` allows instances of `NamespaceProgram` to hand off evaluation of an Active Name to a resolver on a remote machine. This is done through a continuation-passing style (as described below) using Java's Remote Method Invocation.

3.2.2 Security Guarantees

Our security model is oriented toward isolating untrusted `NamespacePrograms` from one another and from the underlying machine both for security and to limit resource consumption (see below). We do this by (i) requiring that all accesses by a `NamespaceProgram` to sensitive system resources or to other `NamespacePrograms` be via explicit calls to the Virtual Machine interfaces, and (ii) requiring `Namespaces` to explicitly identify themselves in all calls to the Virtual Machine so that the Virtual Machine knows what security restrictions to enforce and to whom to charge resource consumption.

To enforce these requirements, the Virtual Machine uses java stack inspection [Wallach & Felten 1998] as implemented in Sun's Java JDK 1.2. The system's `java.security.SecureClassLoader` associates all NamespaceProgram classes loaded with a `java.security.CodeSource`[3] from which the system loaded the NamespaceProgram class. When loading a remote class, the SecureClassLoader initializes a `java.security.ProtectionDomain` for the new class by asking the current `java.security.Policy` for a `java.security.PermissionsCollection` for the class's CodeSource. The PermissionCollection is essentially a list of capabilities for the ProtectionDomain.

Our `ActiveName.SecurityPolicy` class is a subclass of the default `java.security.Policy` and ensures that `NamespaceProgram` classes are loaded with minimal permissions. In particular, we grant exactly two capabilities. The first is the standard java permission "Class.getProtectionDomain"; this permission allows the namespace to learn its protection domain and from that to learn the name of its CodeSource. The second is a new permission formed by concatenating the string `ActiveName.capability` with a string that uniquely identifies the CodeSource.

Whenever a NamespaceProgram makes a call to the Virtual Machine, it includes its CodeSource as an explicit argument. The Virtual Machine invokes the stack inspection mechanism to verify that the identity of the caller matches the call's claim by verifying that the current stack has a valid permission for `ActiveName.capability.CodeSource`. If not, the request is rejected; otherwise, the call may proceed and the Virtual Machine knows the identity of the caller making the request. This allows it to, for instance, restrict local disk cache accesses by a NamespaceProgram to a specific subdirectory in the system.

It might appear that the java stack inspection mechanism could provide security without requiring each AN-VM request to include a correct CodeSource. For instance, the SecurityPolicy could grant each CodeSource a FilePermission to read and write a particular subdirectory. However, this allows programs direct access to the filesystem, which makes it more difficult for the Virtual Machine to enforce resource limitations.

3.2.3 Resource Limitations

Instances of `NamespaceProgram` can be arbitrary, untrusted code. Therefore, the Virtual Machine must ensure that these programs do not consume arbitrary resources as they evaluate an Active Name. We focus on five local resources: disk bandwidth, disk space, network bandwidth, CPU cycles, and memory.

Disk and network resources are easy to manage because all accesses to the disk and network must go through the resolver's Virtual Machine. The security policy ensures that direct access to the network and disk are automatically rejected. The methods on the Virtual Machine used to access disk and network resources track consumption on a per-Namespace basis to guarantee that pre-defined limits are not exceeded.

Our CPU scheduler manages cpu resources on a per-request granularity. When a new request enters the system, the Virtual Machine assigns it to a thread group. The request may spawn multiple sub-threads, but all cycles used will be charged to the originating thread group and request. We grant each request a soft limit and a hard limit of CPU time. While a request has consumed less than its soft limit of cycles, it runs with normal priority. When it has used more than its soft limit but less than its hard limit, it runs with reduced priority. The CPU scheduler kills thread groups that exceed hard CPU limits.

The CPU scheduler is a very high priority Java thread that manipulates Thread and ThreadGroup priorities both to track and limit the resources consumed by each request. When the Virtual Machine creates a new thread group for a new request, it registers the thread group with the scheduler. The scheduler spins in a loop that sleeps for a random period of time and then determines the highest priority of any job's active threads and how many active threads each thread group that are running at that high priority. The scheduler estimates how many cycles each thread group by assuming that the CPU was time-sliced evenly among all highest-priority active threads while the scheduler slept. For example, if the scheduler sleeps for t milliseconds and a job has h high-priority threads out of H high priority threads among all jobs, the scheduler increments the job's CPU consumption estimate by $t * h/H$. After updating its accounting on all jobs, the scheduler reduces the priority of jobs that have exceeded their soft limits and kills jobs that have exceeded their hard limits.

Our prototype system does not presently limit memory consumption. Because the Java language and runtime environment present no satisfactory way to monitor or control the memory usage of classes and threads, rationing memory will require modifications to the underlying JVM. This feature is planned as future work.

3.3 After Methods

One of the arguments for the evaluation of any Active Name (the method `Eval`) is an array of Active Names called `afterMethods` (see Figure 1). Similar to after methods in CLOS [Steele Jr. 1990], the array of after methods represents a list of Active Name classes that will be invoked in order once evaluation of the main class has completed. As depicted in Figure 2, after methods are typically responsible for returning the result of evaluating an Active Name back to the client, perhaps performing some transformation on the data along the way. By convention, each Active Name class

[3] A `CodeSource` encapsulates the URL from which code was fetched and any public keys that signed the code.

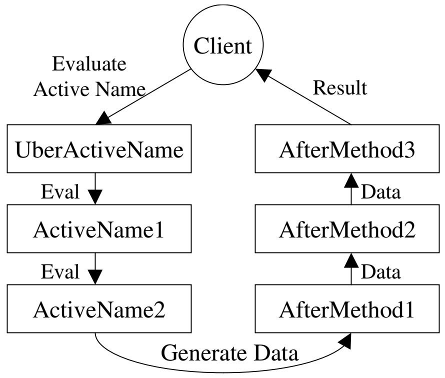

Figure 2: This Figure describes how after methods interact with the evaluation of Active Names. Each after method acts as a filter on the result (a data stream) of evaluating an active name.

promises to run the first element of the array of AfterMethods after completing its own task. The array can be empty, in which case no action will be taken upon completion of the main class. Typically however, at least one after method with the namespace “ReplyNamespaceProgram” will appear on the array. The name field of the Active Name is a string containing a hostname and a port where the client awaits the evaluation of the Active Name it originally generated.

For example, if a client requests evaluation of the Active Name “yahoo/search”, it will listen on a local port for HTML representing the contents of that object. Thus, when requesting evaluation of the object, the client invokes code similar to that outlined in Figure 3(a). The client first creates and runs a thread that will listen for the result of the evaluation on, say, port 8040 of the local host. It then creates array of Active Names with a single element specifying the namespace, “ReplyNamespaceProgram,” and the name to be evaluated within this namespace, “myhost:8040” (the address where the client expects to get its answer). It then invokes the method `Eval` on the virtual machine implemented by the local Active Name Resolver. `UberNamespaceProgram` evaluates the Active Name “yahoo/search” and determines that the class `YahooNamespaceProgram` is responsible for the “yahoo” namespace. `UberNamespaceProgram` spawns a thread to invoke the Yahoo code and then exits. Once the Yahoo code completes evaluation of the Active Name (and stores the result in the DataStream), it will pop the first element off of `afterMethods` and request that the virtual machine evaluate this Active Name, as summarized by the sample Java code in Figure 3(b). Evaluation of `ReplyNamespaceProgram` will cause the contents of the data stream (HTML for “yahoo/search”) to be transmitted to the waiting client.

Most classes follow the convention of setting up a stream of data that flows from class to class to avoid store and forward delays. Thus, after methods are often invoked before the previous class has completed its evaluation (i.e., as soon as it is ready to start producing data on the `DataStream`). After methods allow a natural implementation of continuation-passing style evaluation of Active Names and the integration of 3-way RPC's in name resolution. A single After methods of class `ReplyNamespaceProgram` allows the system to pass name evaluation from resolver to resolver, with the result directly transmitted back to the client (rather than working its way back through the entire chain of resolvers).

Another utility of after methods is the opportunity to flexibly perform client or service-specific transformation or customization on a resource before returning the resource to the client. For example, the mobile distillation application described in Section 1 is implemented through an after method that transforms pictures into a format suitable to particular clients. As will be described in Section 4.1, a mobile distillation after method has the ability to insert itself into optimal points in the evaluation stream to take advantage of available network bandwidth and computation power. Similarly, after methods could expand Server Side Include (SSI) directives in HTML as a filter on data returned to the client.

3.4 Example

As a concrete example of the functionality described in the previous subsections, consider the evaluation of the Active Name “CNN/Frontpage”. The evaluation of this name will be customized along a number of axes: the user sits behind a slow modem and prefers small, black and white pictures to conserve bandwidth. The HTML representing CNN's front page is customized to contain the news stories, stock quotes, and weather forecasts preferred by the user; the service selects among a number of different advertisements based on the object requested and information on the user; the service also wishes to keep track of the number of accesses to a given page, both for advertising revenue and to track the popularity of different pages. Each customization can be thought of as a filter on data. These filters are implemented through NameSpace program. In this example, a number of different programs cooperate to produce the desired resource for the end client. The steps are summarized below:

- The client generates the Active Name “CNN/Frontpage” and requests that the local name resolver evaluate this name along with a array of after-methods containing two entries. The first is responsible for distillation, transforming images to match client requirements (in this case, reducing size and converting to black and white). The second after

```
t = new ListenForAnswerThread(8040); // client chooses port
t.run();
afterMethods[0] = new ActiveName("ReplyNamespaceProgram", "myhost:8040");
VirtualMachine.Eval("UberNamespaceProgram", "yahoo/search",
                    new DataStream(), afterMethods);
```

(a) Name Evaluation

```
// remove the first element, e.g., ReplyNamespaceProgram/myhost:8040
ActiveName afterMethod = afterMethods.pop(); // remove first element
VirtualMachine.Eval(afterMethod, dataStream, afterMethods);
```

(b) After Method Invocation

Figure 3: Part (a) of this figure presents sample Java code for a client wishing to evaluate the Active Name "yahoo/search." Part (b) presents sample Java code for invoking the first after method, in this case an Active Name responsible for delivering the resource represented by "yahoo/search" back to the client.

method is a program that knows how to transmit the result of evaluating an Active Name back to the waiting client. Note that (i) these after methods will not be run until after the name has been evaluated, (ii) the client only specifies the name of programs; retrieving the actual code is the responsibility of name resolvers.

- After requesting evaluation, the client listens for the result (this operation can be either blocking or non-blocking). However, the result does not have to come from the local name resolver. Active Name evaluation is accomplished through a continuation-passing style, so the result of evaluation will often come back from a third party. Such multi-way RPC's reduce the need for multiple round-trip communications to evaluate a name, allowing requests (and their evaluation state) to be routed directly to locations best able to carry out evaluation.

- Given the Active Name, the local name resolver will retrieve code specific to the CNN service for evaluating the name. This code: (i) contacts servers for any user preferences and caches them for future reference, (ii) retrieves the appropriate news stories, stock quotes, etc. from CNN servers, (iii) inserts advertising, either randomly or tailored to the particular user, and (iv) maintains a profile of access characteristics that are periodically transmitted back to servers.

- Once the CNN program completes its work, the name resolver executes all after methods associated with the evaluation of the Active Name. In this example, one after method distills images to client preferences and the second transmits the result back to the waiting client.

4 Applications

4.1 Mobile Distillation

Clients accessing Web resources vary widely in their characteristics from powerful workstations with fast Internet links to hand-held devices with small black and white displays, little processing power, and slow connections. The proper way to present Web content depends on these client 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 [Fox et al. 1996, Fox et al. 1997] is to mediate client accesses 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 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 requirements. Active Names allow clients to specify and control the contents of the objects returned to them. Because Active Names take advantage of arbitrary computation resources distributed throughout the network, they free clients from being bound to a single proxy or from locating appropriate proxies in the case of mobile clients. Further, since the clients specify (and potentially provide) the programs any Active Name resolver distributed throughout the network can potentially satisfy client requests.

An added benefit of taking advantage of network resources with respect to distillation is the ability to flexibly choose the transformation point of a requested resource at

	Proxy Distillation	Server Distillation
No Server Load	3.8 s / 15.8 s	3.1 s / 11.2 s
High Server Load	3.6 s / 16.0 s	14.5 s / 117.4 s

Table 1: This table describes the performance of Dynamic Distillation implemented with Active Names. Each entry reports client latency in retrieving a small and a large image. The proxy is located at U.C. Berkeley and the server is located at Duke University. Server load is varied from no load to ten competing CPU-intensive processes.

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 at the proxy that greatly reduces the size of the image. In a function versus data-shipping tradeoff, it is usually more efficient to perform the transformation at the server and then 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 CPU server is over 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 have implemented dynamic distillation within the Active Naming framework. A distillation after method applies filters to images, shrinking images and converting color images to grayscale. To evaluate the utility the utility of flexibly setting the distillation point, we ran the following experiment. A client at U.C. Berkeley requests an image located at Duke University. This request is made under a number of different circumstances. The first variable is where distillation takes place. Active Name resolvers are available at both Berkeley and Duke so execution of the program can take place at either location. The second variable is the load on the machine at Duke, influencing the utility of choosing distillation location. In one case, the Name Resolver at Duke runs on an otherwise unloaded machine. In another, the resolver must compete with ten CPU intensive processes. The third variable is the size of the original and distilled image. For our experiments, we use one 59 K image that is distilled to 9K and a 377 K image that is distilled to 19 K. Machines at both Berkeley and Duke are Sun Ultrasparcs. At the time of our measurements, transferring 377 K from Duke to Berkeley achieves 90 KB/s, while transferring 59 K achieves 38 KB/s. Note that the measurements are taken on a weekend night to minimize variability in wide-area performance. The Active Name resolvers (including all distillation code) are compiled and run with the Java Development Kit, version 1.2 beta 4.

Table 1 shows the client-perceived latency of retrieving distilled versions of the two Jpeg images of varying sizes. For each entry in the table, the left number represents client-perceived latency in obtaining the smaller (59 K distilled to 9 K) image, while the right number is the latency for obtaining the smaller (377 K distilled to 19 K) image. For example, when the image is distilled at the server with no load at the server, the client waits 3.1 seconds for the smaller image and 11.2 seconds for the larger image. Table 1 shows that with no load on the server, distilling the image at the server produces between 20-40% better than performing distillation at the proxy. Larger savings come from performing distilling the larger picture because of the higher cost of transmitting 377 K across the Internet. Also note that the reported performance numbers are pessimistic because are measurements where taken at a time (a weekend night) when the network link was not congested and did not display packet drops. Larger performance improvements should be seen from server distillation during the day when transmitting large amounts of data across the Internet is more expensive.

The performance improvement for server distillation comes from two sources: (i) distillation takes place close to the data meaning that less data must be transmitted across the wide area, and (ii) the server transmits the image directly to the client (as opposed to going through the proxy) via three-way RPC meaning that one hop is avoided. The second row of the table shows that when the server becomes heavily loaded (competing with ten other CPU intensive processes), shipping the data to the proxy where CPU cycles are plentiful produces much better performance: distillation at the proxy performs between four to seven times better than server distillation. In this case, it is preferable to pay the cost of wide-area transmission for available CPU cycles. Also note that the streaming nature of Active Name transmission means the proxy begins performing image transformation as soon as the image begins arriving, a performance enhancement over waiting for the entire image to arrive before beginning computation as is often done with current proxy architectures.

4.2 Extensible Cache Management

The explosive growth of Internet usage has led to similar growth in consumed wide-area bandwidth. Today, network performance is likely the dominant component of accessing remote resources. HTTP traffic currently comprises the majority of of bytes across network backbones [Thompson et al. 1997]. Caching of Web content at local proxies is an important technique for improving performance and reducing consumed bandwidth. Unfortunately, the performance of current caching schemes is limited by low hit rates stubbornly near 50% [Abrams et al. 1995, Duska et al. 1997, Gribble & Brewer 1997, Arlitt & Williamson

Reason	Percent Requests	Percent Bytes
Dynamic	20.7%	14.5%
Consistency	9.9%	8.1%
No Caching	9.2%	12.3%
Compulsory	44.8%	58.0%
Redirection	3.7%	0.1%
Misc	11.5%	7.4%

Table 2: This table presents the breakdown among all requests that missed in the proxy cache of a national ISP. The first column accounts for requests that missed, while the second column breaks down misses by the number of bytes returned by the requests.

1996, Glassman 1994] even for large client populations.

We set out to discover the cause of the large percentage of requests that miss in the cache. We examined a 6-day trace of 7618537 requests for 64.5GB of data by 23080 clients [4] of a national commercial dial-up internet service provider. We simulated the behavior of a proxy cache during this six day period. We used the first day of the trace to warm the cache and gathered statistics for the remaining 5 days that contained 6461733 requests for 54.8GB. Assuming current caching technology – an shared proxy with infinite storage and a cache consistency algorithm based on client polling – 40.2% of all requests (for 29.3% of all bytes) could be satisfied by the cache.

Table 2 summarizes the cause of the remaining accesses that miss in the cache. Because Active Naming allows service-specific code to be run in caches before returning an object, we hypothesize that Active Names could satisfy significant fractions of the requests to proxy caches that miss. Each row of Table 2 is examined below along with an explanation of how Active Naming code could reduce misses of the specified variety.

- 20.7% of miss requests contain URLs that contained either the string "cgi" or the character '?', indicating that they invoked a program at the server. Of course, some of these requests access large or valuable proprietary databases where shipping the function will not be appropriate. Still, we hypothesize that a significant fraction of such requests could be satisfied in a proxy cache by Active Name code (i.e., the code is not necessarily bound to the server).

- 9.9% of the miss requests would be cache hits, but they are delayed by cache consistency polling messages to the server ("Get-if-modified" requests answered with "304 Not Modified" replies) required for client-driven cache consistency. As currently done by typical shared proxy caches, our simulated cache verifies data by querying the server before delivering it to clients. Although cache consistency will still require some communication, Active Naming allows more sophisticated consistency policies to remove polling from the critical path of many such requests.

- 9.2% of the miss requests had headers that forbid caching. HTTP headers generally turn off caching for one of three reasons: i) although the request does not contain a "cgi" or a '?', the object is dynamically generated at the server, ii) the server wishes to count the number of accesses to the page to, for instance, maximize advertising revenue, or iii) the page changes rapidly and the service wishes to avoid use of stale data. Active naming has the potential to make significant fractions of all of these types of requests cachable.

- 44.8% of the miss requests are compulsory misses associated with the first access to a distinct URL in the trace. Prefetching could avoid a significant fraction of these misses [Gwertzman & Seltzer 1996, Padmanabhan & Mogul 1996, Kroeger et al. 1997]. Using service-specific Active Name code has the added benefit of using service knowledge of access patterns to drive prefetching (as opposed to forcing the proxy to determine such access patterns on its own).

- 3.7% of the miss requests are redirections to a new server. Most of these redirections are used to perform load balancing. As the discussion in Section 4.3 demonstrates, Active Name code can usually perform load balancing without the benefit of such redirects. Note that while only a small number of bytes are involved in redirects, they nonetheless make a significant contribution to latency because connection time often dominates web accesses. requests.

- 11.4% of the requests are of a miscellaneous variety, largely comprised of requests that encountered an error and requests for objects that actually had changed at the server. Active Name programs can eliminate avoid forwarding most error requests to the server by maintaining knowledge of the names of objects stored at servers. Similar to the compulsory miss category, Active Names can also eliminate many of the misses to data that actually has changed through prefetching.

Clearly, Active Naming will not make hits out of all of the above categories of misses. Yet, given the difficulty of improving cache hit rates beyond 50% with current cache architectures, pursuing an Active Naming based strategy will be productive even if only half of the current misses are converted to hits.

[4] The ISP uses dynamic IP for dial-up modem users. The trace contains requests by 23080 distinct IP addresses.

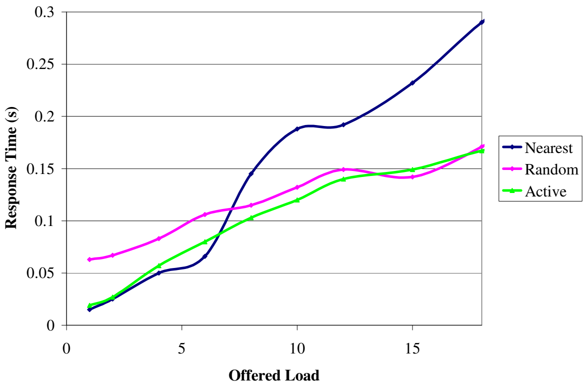

Figure 4: This Figure presents the performance of three different algorithms for choosing among a set of replicas. The x-axis is the number of clients simultaneously requesting a 1 KB file from the replicas. The y-axis shows the average latency perceived by the clients in retrieving the file.

4.3 Replicated Service Location

Today, many service providers are replicating their service across both the local and wide area to achieve better performance, scalability, and availability. A principal problem with replication is determining which replica to access from the client's perspective. Current techniques range from placing the onus entirely on the end user to randomly directing requests to a set of replicas at the IP level (e.g., DNS round robin). Unfortunately, a large number of variables must be considered to optimally choose among replicas. Considerations include replica load/processing power, network connectivity, and incomplete replication (as is often the case with, for example, FTP mirror sites). For example, even considering variable CPU load, network connectivity (bandwidth, latency) may make one replica optimal for one client, whereas a different replica is optimal for a second client.

Active Naming allows service-specific programs to account for any number of variables in choosing a replica, including client, server, and network characteristics. It is beyond the scope of this paper to determine the appropriate replica-selection policy for any service. However, Figure 4 summarizes the results of an experiment demonstrating the importance of introducing programmability into the decision making 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 used one of the three following 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 randomized load balanced access to replicas. However, no programmability or extensibility is possible with this scheme.
- *Distributed Director*: With this product from CISCO [Cisco 1997], specialized code is run in routers that allows services to register the current set of replicas making up a service. Any 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[5].
- *Active Naming*: With this instance of programmable replica selection, the program uses the number of hops (as reported by traceroute) from the client replica to bias the choice of replica. Replicas further away are less likely to be chosen than nearby replicas. However, this weighing is 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. While, such an algorithm is not purported to be optimal it will demonstrate the utility of programmable replica selection.

Figure 4 shows the average latency perceived by clients continuously requesting the same small 1 KB file from the replicated service. The x-axis presents offered load, varying the number of clients simultaneously requesting the file. The y-axis shows the average latency in seconds perceived by the clients. The graph shows that at low load the proper replica selection policy is to choose the "nearest" replica in Berkeley because accessing the Seattle server would consistently incur higher latency as a number of wide-area links must be traversed. This is the policy implemented by Distributed Director and this policy shows the best performance at low load. However, as load increases, the Berkeley replica begins to become over-loaded, and the proper policy is to send approximately half the requests to the Seattle replica. In this regime, the cost of going across the wide area is amortized by the high load at the Berkeley server. Such load balancing is implemented by DNS round robin, which achieves the best performance at high load.

Note that the simple Active Naming policy is able to track the best performance of the two policies by accounting for not just distance, but also previous performance. At low levels of load, both distance and previous performance heavily bias Active Naming toward the Berkeley replica. However, as load increases and performance at the Berkeley replica degrades, an increasing number of the requests are routed to Seattle achieving better overall performance.

[5]Note that minimizing hop count is not guaranteed to achieve geographic locality or choose the server able to deliver the lowest latency/highest bandwidth to a particular client.

4.4 Personalization

The final application we have implemented in the Active Naming framework is service-controlled personalization of a Web service. For example, many news services [Yahoo 1996, CNN 1998] allow a single name (a URL to a front page) to be customized to user preferences for headlines, stock quotes, weather forecasts, etc. Currently, this mapping is accomplished by translating a "cookie" uniquely identifying a user to an entry in the server's database describing such preferences. Such personalization is implemented with greater generality with and location independence Active Names.

An Active Name program stores user preferences in the cache of an Active Name Resolver (the entire class, along with user preferences is serialized to disk). Using Active Names has a number of benefits. For example, today proxies cannot cache objects associated with cookies; provided the server trusts the cache to execute the Active Name, we can provide customization near the client because the Active Name is an agent of the service. The Active Name program also tracks user accesses, ensuring that hit counts are correctly transmitted to the service (i.e., for advertising revenue). The program also performs service-specific cache consistency. For example, a service might update a set of headlines every two hours, meaning there is no need to check for updates at other intervals. Finally, the Active Name program personalizes objects by customizing advertisement banners to user preferences, or by simply rotating among a set of available banners [Cao et al. 1998].

5 Related Work

Our work in Active Naming introduces programmability and location independence to existing wide-area naming systems such as Appollo Domain [Leach et al. 1983], DNS [Mockapetris & Dunlap 1988] and LDAP [Bolot & Afifi 1993]. For their target domains, these systems have been successful. However, as new requirements are introduced it is difficult to expand such systems. Current wide-area computing research proposals, such as Globe [van Steen et al. 1998], Globus [Fitzgerald et al. 1997], and Legion [Grimshaw et al. 1995], propose a number of schemes for locating computational resources across the wide area. These proposals are orthogonal to our work as any could be incorporated within the extensible Active Naming framework.

Prospero [Neuman 1992, Neuman et al. 1993] also supports extensible naming to support mobility and the integration of multiple wide-area information services (e.g., WAIS and gopher). Relative to Prospero, our work demonstrates the utility of location-independent and portable programs for name resolution, a security and resource allocation model for extensions, and implementation of a different set of wide-area applications. Programmability in Active Names grew out of earlier work on Smart Clients [Yoshikawa et al. 1997]. 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 [Bhattarcharjee et al. 1997, Fei et al. 1998], Nomenclator [Ordille & Miller 1993], and Query Routing [Leach & Weider 1997] 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 [Cao et al. 1998] allow for customization of cache content through Java programs similar to our extensible cache management system described in Section 4.2. 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 and allows, for example, the program to manage local cache replacement policy or to perform load balancing (on a cache miss).

6 Conclusions

In this paper, we described Active Names, a general framework for the development and composition of applications requiring access to wide-area resources. The key insight behind Active Names is the need to introduce programmability to support the widely varying semantics and requirements of distributed applications. To this end, an application-specific and location independent program is associated with each name resolution. Location independence allows the system to take advantage of remote computational resources and provides a framework to evaluate tradeoffs in function versus data shipping. Further, three-way RPC's are utilized to route results directly back to clients, reducing latency and consumed bandwidth. To demonstrate the utility of the Active Naming framework, this paper describes the design and implementation of four sample applications: mobile distillation of Web content, load balancing and replica selection across wide-area servers, extensible cache management designed to increase the utility of Web caches, and personalization of Web content to client preferences.

References

[Abrams et al. 1995] 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.

[Anderson et al. 1995] T. E. Anderson, D. E. Culler, D. A. Patterson, and the NOW Team. "A Case for NOW (Networks of Workstations)". *IEEE Micro*, February 1995.

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

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

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

[Bolot & Afifi 1993] J. Bolot and H. Afifi. "Evaluating Caching Schemes for the X.500 Directory". In *Proceedings the 13th International Conference on Distributed Computing Systems*, pp. 112–119, Pittsburgh, PA, 1993.

[Brewer 1997] E. Brewer. Personal Communication, March 1997.

[Brisco 1995] T. Brisco. "DNS Support for Load Balancing", April 1995. Network Working Group RFC 1794.

[Cao et al. 1998] P. Cao, J. Zhang, and K. Beach. "Active Cache: Caching Dynamic Contents on the Web". In *Proceedings of Middleware*, 1998.

[Chankhunthod et al. 1996] A. Chankhunthod, P. Danzig, C. Neerdaels, M. Schwartz, and K. Worrell. "A Hierarchical Internet Object Cache". In *Proceedings of the 1996 USENIX Technical Conference*, January 1996.

[Chen et al. 1993] P. M. Chen, E. K. Lee, A. L. Drapeau, K. Lutz, E. Miller, S. Seshan, K. Shirriff, D. A. Patterson, and R. H. Katz. "Performance and Design Evaluation of the RAID-II Storage Server". In *Proceedings of International Parallel Processing Syposium 1993 Workshop on I/O*, 1993.

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

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

[Daniel & Mealling 1996] R. Daniel and M. Mealling. "Resolution of Uniform Resource Identifiers using the Domain Name System". Internet Draft, see `http://www.acl.lanl.gov/URN/naptr.txt`, September 1996.

[Deering & Cheriton 1990] S. E. Deering and D. R. Cheriton. "Multicast Routing in Datagram Internetworks and Extended LANs". In *Transactions on Computer Systems*, 1990.

[Dig 1995] Digital Equipment Corporation. *Alta Vista*, 1995. `http://www.altavista.digital.com/`.

[Duska et al. 1997] B. Duska, D. Marwood, and M. 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.

[Fei et al. 1998] Z. Fei, S. Bhattacharjee, E. W. Zegura, and M. Ammar. "A Novel Server Selection Technique for Improving the Response Time of a Replicated Service". In *Proceedings of IEEE Infocom*, July 1998.

[Fitzgerald et al. 1997] 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*, pp. 365–376, 1997.

[Foster & Kesselman 1996] I. Foster and C. Kesselman. "Globus: A Metacomputing Infrastructure Toolkit". In *Proc. Workshop on Environments and Tools*, 1996.

[Fox et al. 1996] A. Fox, S. Gribble, E. Brewer, and E. Amir. "Adapting to Network and Client Variability via On-Demand Dynamic Distillation". In *Proceedings of the Seventh International Conference on Archictectural Support for Programming Languages and Operating Systems*, Cambridge, MA, 1996.

[Fox et al. 1997] A. Fox, S. Gribble, Y. Chawathe, and E. Brewer. "Cluster-Based Scalable Network Services". In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, Saint-Malo, France, October 1997.

[Glassman 1994] S. Glassman. "A Caching Relay for the World Wide Web". In *First International World Wide Web Conference*, pp. 69–76, May 1994.

[Gribble & Brewer 1997] S. D. Gribble and E. 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.

[Grimshaw et al. 1995] 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.

[Gwertzman & Seltzer 1996] J. Gwertzman and M. Seltzer. "World-Wide Web Cache Consistency". In *Proceedings of the 1996 USENIX Technical Conference*, pp. 141–151, January 1996.

[Ioannidis & Maguire 1993] J. Ioannidis and G. Q. Maguire. "The Design and Implementation of a Mobile Internetworking Architecture". In *Winter Usenix Conference*, pp. 491–502, January 1993.

[Katz et al. 1994] E. D. Katz, M. Butler, and R. McGrath. "A Scalable HTTP Server: The NCSA Prototype". In *First International Conference on the World-Wide Web*, April 1994.

[Kroeger et al. 1997] 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.

[Leach & Weider 1997] 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*, pp. 82–86, Cape Code, MA, 1997.

[Leach et al. 1983] P. Leach, P. H. Levine, B. Douros, J. Hamilton, D. Nelson, and B. Stumpf. "The Architecture of an Integrated Local Network". In *IEEE Journal on selected Areas in Communications*, volume SAC-1, pp. 842–856, 1983.

[Mockapetris & Dunlap 1988] P. Mockapetris and K. Dunlap. "Development of the Domain Name System". In *Proceedings SIGCOMM 88*, volume 18, pp. 123–133, April 1988.

[Net 1994] Netscape Communications Corporation. *Netscape Navigator*, 1994. http://www.netscape.com.

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

[Neuman et al. 1993] B. C. Neuman, S. S. Augart, and S. Upasani. "Using Prospero to Support Integrated Location Independent Computing". In *Proceedings of the Symposium on Mobile and Location Independent Computing*, August 1993.

[Ordille & Miller 1993] J. Ordille and B. P. Miller. "Distributed Active Catalogs and Meta-Data Caching in Descriptive Name Services". In *IEEE International Conference on Distributed Computing Systems*, pp. 120–129, May 1993.

[Padmanabhan & Mogul 1996] 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*, pp. 22–36, July 1996.

[Paxson 1996] V. Paxson. "End-To-End Routing Behavior in the Internet". In *Proceedings of the ACM SIGCOMM '96 Conference on Communications Architectures and Protocols*, August 1996.

[Perkins 1996] C. Perkins. "IP Mobility Support". RFC 2002, October 1996.

[Sollins & Masinter 1994] K. Sollins and L. Masinter. "Functional Requirements for Uniform Resource Names". RFC 1737, December 1994.

[Steele Jr. 1990] G. L. Steele Jr. *Common LISP: The Language*. Digital Press, second edition, 1990.

[Tennenhouse & Wetherall 1996] D. Tennenhouse and D. Wetherall. "Towards an Active Network Architecture". In *ACM SIGCOMM Computer Communication Review*, pp. 5–18, April 1996.

[Terry et al. 1995] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. "Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System". In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pp. 172–183, December 1995.

[Tewari et al. 1998] R. Tewari, M. Dahlin, H. Vin, and J. Kay. "Beyond Hierarchies: Design Considerations for Distributed Caching on the Internet". Technical Report TR98-04, University of Texas at Austin, Feb 1998.

[Thompson et al. 1997] K. Thompson, G. J. Miller, and R. Wilder. "Wide-Area Internet Traffic Patterns and Characteristics". In *IEEE Network*, Nov/Dec 1997.

[Vahdat et al. 1998] A. Vahdat, T. Anderson, M. Dahlin, E. Belani, D. Culler, P. Eastham, and C. Yoshikawa. "WebOS: Operating System Services for Wide Area Applications". In *Proceedings of the Seventh IEEE Symposium on High Performance Distributed Systems*, Chicago, Illinois, July 1998.

[van Steen et al. 1997] M. van Steen, P. Homburg, and A. S. Tanenbaum. "The Architectural Design of Globe: A Wide-Area Distributed System". Technical Report Technical Report IR-422, Vrije Universiteit, March 1997.

[van Steen et al. 1998] M. van Steen, F. Hauck, P. Homburg, and A. Tanenbaum. "Locating Objects in Wide-Area Systems". In *IEEE Communications Magazine*, pp. 104–109, January 1998.

[Wallach & Felten 1998] D. S. Wallach and E. W. Felten. "Understanding Java Stack Inspection". In *IEEE Symposium on Security and Privacy*, May 1998.

[Wetherall et al. 1998] D. Wetherall, U. Legedza, and J. Guttag. "Introducing New Network Services: Why and How". In *IEEE Network Magazine, Special Issue on Active and Programmable Networks*, July 1998.

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

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

[Zhang et al. 1993] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. "RSVP: A New Resource ReSerVation Protocol". In *IEEE Network*, September 1993.

[Zhang et al. 1997] L. Zhang, S. Floyd, and V. Jacobsen. "Adaptive Web Caching". In *Web Caching Workshop*. National Laboratory for Applied Network Research, June 1997.