Operating System Services for Wide-Area Applications

by

Mohammad Amin Vahdat

B.S. (University of California, Berkeley) 1992

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA, BERKELEY

Committee in charge:

Professor Thomas E. Anderson, Cochair
Professor John D. Kubiatowicz, Cochair
Professor Anthony D. Joseph
Professor Kenneth Y. Goldberg

Fall 1998

The dissertation of Mohammad Amin Vahdat is approved:

Cochair Date

Cochair Date

Date

Date

University of California at Berkeley

Fall 1998

Operating System Services for Wide-Area Applications

Copyright Fall 1998

by

Mohammad Amin Vahdat

Abstract

Operating System Services for Wide-Area Applications

by

Mohammad Amin Vahdat

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Thomas E. Anderson, Cochair

Professor John D. Kubiatowicz, Cochair

With over 100 million users and 25 million hosts, the Internet has achieved the critical mass necessary to support new classes of wide-area distributed applications, including electronic commerce, news services, multi-player gaming, and interactive multimedia. Unfortunately, robust deployment of such applications is hampered by limited latency, bandwidth, and availability of Internet services deployed at a single centralized site. This dissertation advocates the construction of virtual services able to dynamically migrate and replicate across the wide area. However, such virtual services requires a rethinking of all aspects of distributed services, including naming, persistent storage, remote execution, and security.

The hypothesis of this dissertation is that remotely programmable network elements allow for a restructuring of wide-area systems that will improve wide-area resource utilization, simplify application development, and improve end-to-end performance. To

support our hypothesis we propose novel solutions to address the unique requirements of distributed applications for wide-area naming, persistent storage, and security. A principle that cuts across all of our solutions is flexibility. The heterogeneity of the wide-area network, clients, and service providers dictates that no single policy is appropriate for all applications in all situations. By including flexibility in our solutions from the ground up, we ensure that our techniques are applicable to a wide variety of distributed applications. Further, with flexibility, our techniques are likely to remain applicable as the Internet and its applications continue to evolve.

This dissertation makes the following specific contributions in support of virtual services by proposing and evaluating: (i) location-independent programs to find and retrieve wide-area resources, (ii) combining communication and persistence in a location-independent file system with flexible cache coherence policies, (iii) techniques for caching the results of dynamically generated Web content, and (iv) a wide-area security system that provides high performance and availability despite network limitations, fine-grained control over remotely programmable resources, and rights transfer and revocation between multiple administrative domains. This dissertation also demonstrates that integrating these techniques simplifies the development of a number of wide-area applications, including a Web server capable of dynamically replicating itself across the wide area in response to client access patterns.

Professor Thomas E. Anderson
Dissertation Committee Cochair

Professor John D. Kubiatowicz
Dissertation Committee Cochair

To my wife Suzanne Vahdat, my constant source of inspiration

Contents

List of Tables viii

List of Figures ix

1 Introduction 1
- 1.1 Background . . . 1
- 1.2 Application Domain . . . 3
- 1.3 Motivation . . . 7
  - 1.3.1 Problem Statement . . . 7
  - 1.3.2 Experimental Validation . . . 9
- 1.4 Our Approach . . . 12
- 1.5 Thesis and Contributions . . . 13
- 1.6 Organization . . . 15

2 System Goals and Design 17
- 2.1 Goals . . . 18
- 2.2 Requirements . . . 19
  - 2.2.1 Example: Rent-A-Server . . . 21
- 2.3 Design . . . 24
  - 2.3.1 Design Overview . . . 24
  - 2.3.2 System Organization . . . 27

3 Active Naming 31
- 3.1 Overview . . . 31
  - 3.1.1 Background and Motivation . . . 31
  - 3.1.2 The Active Name Approach . . . 34
- 3.2 Architecture and Implementation . . . 36
  - 3.2.1 Dynamic Code Location . . . 40
  - 3.2.2 Active Name Resolver . . . 42
  - 3.2.3 After-Methods . . . 46
  - 3.2.4 Example . . . 49
- 3.3 Applications . . . 51

3.3.1 Mobile Distillation . . . 51
3.3.2 Extensible Cache Management . . . 54
3.3.3 Replicated Service Location . . . 58
3.3.4 Personalization . . . 62
3.4 Related Work . . . 63
3.5 Summary . . . 65

4 Global File System **67**
4.1 Overview . . . 67
4.2 Architecture . . . 71
4.2.1 System Overview . . . 71
4.2.2 Naming . . . 73
4.2.3 HTTP Limitations . . . 74
4.2.4 Authentication and Security . . . 75
4.3 Performance . . . 76
4.4 Cache Coherence Policies . . . 77
4.4.1 General File Sharing: Last Writer Wins . . . 78
4.4.2 Internet Chat: Append Only . . . 79
4.4.3 Stock Ticker: Multicast Updates . . . 81
4.5 Related Work . . . 84
4.6 Summary . . . 86

5 Transparent Result Caching **88**
5.1 Overview . . . 88
5.2 System Design and Implementation . . . 92
5.2.1 Architecture . . . 92
5.2.2 Performance . . . 94
5.2.3 Limitations . . . 96
5.3 Applications . . . 98
5.3.1 Unmake . . . 98
5.3.2 Transparent Make . . . 101
5.3.3 Dynamic Web Caching . . . 103
5.4 Related Work . . . 110
5.5 Summary . . . 113

6 Security **115**
6.1 Overview . . . 115
6.2 Design Principles . . . 118
6.3 System Architecture . . . 122
6.3.1 Validating and Revoking Statements . . . 125
6.3.2 Processes and Roles . . . 128
6.3.3 Hierarchical Trust . . . 131
6.3.4 Time . . . 132
6.3.5 Authorization . . . 134
6.3.6 Discussion . . . 136

6.4 CRISIS Protocols . . . 138
6.4.1 Login . . . 138
6.4.2 Accessing a Remote File . . . 143
6.4.3 Running a Remote Job . . . 145
6.5 Performance . . . 146
6.6 Related Work . . . 148
6.7 Summary . . . 150

7 Rent-A-Server 152
7.1 Overview . . . 152
7.2 Motivation . . . 153
7.3 Current Approaches . . . 155
7.4 System Design . . . 156
7.5 Performance . . . 159
7.5.1 Dynamic Recruitment of Resources . . . 159
7.5.2 Reduced Wide-Area Latency and Bandwidth . . . 162
7.6 Summary . . . 166

8 Related Work 167
8.1 Distributed Operating Systems . . . 167
8.2 Cluster Computing . . . 168
8.3 Global Computation . . . 169
8.3.1 GOST . . . 170
8.3.2 Globe . . . 170
8.3.3 Globus . . . 171
8.3.4 Legion . . . 172
8.3.5 Summary . . . 173
8.4 Scalable Internet Services . . . 173
8.4.1 Active Networks . . . 174
8.4.2 TACC . . . 174
8.4.3 Active Services . . . 175
8.5 Remote Computation . . . 176
8.6 Summary . . . 178

9 Conclusions and Future Work 180
9.1 Contributions . . . 180
9.2 Future Work . . . 182
9.2.1 Scalability and Fault Tolerance . . . 183
9.2.2 Performance Isolation . . . 183
9.2.3 Simulation . . . 184
9.2.4 Application Studies . . . 184
9.2.5 Resource Allocation . . . 186
9.2.6 Security . . . 186
9.3 Summary . . . 188

Bibliography **189**

List of Tables

2.1 Required wide-area operations and corresponding system support. 27

3.1 Performance of mobile distillation implemented with Active Names. 53
3.2 Breakdown of misses in large ISP proxy cache. 55

4.1 WebFS performance relative to NFS on the Modified Andrew Benchmark suite. 76

5.1 TREC profiling overhead. 94

6.1 Performance overhead of adding CRISIS protocols to WebFS. 146

List of Figures

1.1 Average client latency to four Internet servers. 9
1.2 Percentage of packets successfully transmitted to four Internet servers. . . . 10

2.1 The Rent-A-Server application. 22
2.2 High-level system design. 24
2.3 Transparent access to remote resources. 25

3.1 The Active Name Evaluation Process. 38
3.2 Interfaces for the Active Naming prototype. 39
3.3 Interaction of after-methods Active Name Evaluation. 46
3.4 Sample name evaluation and after-method invocation. 48
3.5 Performance of multiple wide-area load balancing algorithms. 59

4.1 WebFS architecture. 72
4.2 Implementation of the last writer wins cache coherence policy. 78
4.3 Chat rooms as WebFS files. 80

5.1 TREC architecture overview. 93
5.2 Compilation script and corresponding process lineage tree. 99
5.3 Sample TREC lineage query results. 100
5.4 Performance improvement of CGI object caching. 107

6.1 CRISIS interaction with system services (naming, remote execution, and the file system). 121
6.2 Sample CRISIS execution scenario: delegating execution remote administrative domains. 123
6.3 Structure of a CRISIS transfer certificate. 127
6.4 CRISIS security managers associate processes with privileges. 128
6.5 Authenticating principal login in CRISIS. 139
6.6 CRISIS integration with WebFS. 143

7.1 Rent-A-Server architecture. 157
7.2 Rent-A-Server performance with dynamic client load. 161

7.3 Rent-A-Server wide-area experimental setup. 162
7.4 Rent-A-Server wide-area performance: benefits of dynamic replication. 164

Acknowledgements

I learned a tremendous amount in graduate school, and I have looked forward to thanking those who, in many different ways, contributed to my success. I am fortunate to have the opportunity to work with Tom Anderson as my advisor. Tom has provided timely and valuable technical guidance, criticism, and feedback on all of my work. In addition, he has been instrumental in the development of my writing and presentation skills. Tom has always been very accessible, acting as a sounding board for countless (often half-baked) ideas. He possesses the remarkable capability of extracting the kernel of a good idea from poorly presented ramblings.

I am also fortunate to work with Tom Anderson, David Patterson and David Culler on the Network of Workstations project at Berkeley. They made significant contributions to my understanding of experimental methodology, designing and leading large research projects, and the importance of transferring technology to industry to ensure high quality, high impact research. Having the opportunity to learn from their significant abilities greatly contributed to my graduate education.

Ken Goldberg, Anthony Joseph, and John Kubiatowicz have been kind enough to serve on my thesis committee despite the difficulties of remote interaction. Their feedback contributed to this dissertation. This work also benefited from my interaction with Mike Dahlin. We worked together on many of the ideas presented in this thesis, and I look forward to continuing our collaboration in the future.

At Berkeley, Kathryn Crabtree, Theresa Lessard-Smith, and Bob Miller have shielded myself and countless other students from the hurdles of graduate life at Berkeley. I

am also indebted to all of my fellow students, especially members of the NOW project, who made graduate school such an enriching experience. In particular Remzi Arpaci-Dusseau, Paul Eastham, Douglas Ghormley, Rich Martin, Steve Rodrigues, and Chad Yoshikawa provided feedback, criticism, and friendship that I hope will continue in the years to come.

A fortunate set of circumstances led to my spending the last year of my graduate career at the University of Washington. Everyone at the University of Washington has been very supportive and helpful in making my time at the University of Washington productive and enjoyable. I would like to thank David Becker, Brian Bershad, Ed Lazowska, Hank Levy, Przemyslaw Pardyak, Stefan Savage, and Ashutosh Tiwary for all their advice, time, and technical insight. In particular, my officemate Stefan has gone above and beyond the call of duty in answering my questions and cutting through the local bureaucracy.

The support and example set by my parents put me in a position to pursue this degree. Learning from their example of work, dedication, and the value of education, I gained the tools to succeed in graduate school.

Finally, much of the credit for this dissertation and my success as a graduate student goes to my wife, Suzanne Vahdat. Her support, confidence, and love provides the spark for my creativity and focus. I am also indebted to her for reading this dissertation cover to cover, finding numerous mistakes, and encouraging me to make this work the best it could be.

This research was supported in part by the DARPA (N00600-93-C-2481, 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.

Chapter 1

Introduction

1.1 Background

> *"We reject kings, presidents, and voting; we believe in rough consensus and running code."*
>
> *- Dave Clark*

As captured by Dave Clark, emphasis on practical, running systems has sustained the exponential growth of the Internet for nearly thirty years. The Internet, a vast interconnected mesh of organizational, regional, and transit networks began as the ARPANET in 1969 with four nodes. The designers of this network had the following goals:

1. To develop techniques and obtain experience on interconnecting computers in such a way that a very broad class of interactions were possible and

2. To improve and increase computer research productivity through resource sharing [Heart et al. 1978].

By 1973, 37 nodes were connected to the ARPANET, and in 1977 the Internet Protocol (IP) was adopted as the single universal protocol for transferring packets across the network. When I obtained my first Internet account in 1988, there were 50,000 hosts connected to the Internet. In the ten subsequent years, this number has increased by a factor of 5,000 to 25 million connected hosts. The number of such hosts continues to double every year.

In its early years, use of the Internet was largely restricted to academics and government employees sharing expensive and centralized computational resources and exchanging ideas. The year 1992, however, marked a milestone in the growth of the Internet. Over one million users gained access to the Internet. The majority of Internet users were now corporate employees and, increasingly, private individuals. Also in 1992, HTTP along with the first Web browser became publicly available. Through a graphical user interface (GUI) and multimedia elements, the contents and services of the Internet became accessible to an even larger group of world-wide users.

The growth of the Internet has also led to an expansion in the variety of distributed applications. Traditionally, fairly simple distributed applications, such as electronic mail, file transfer, and remote login were run over the Internet. While these remain important applications, the critical mass in the number of Internet users along with accessible GUI's has lead to the development of new classes of distributed applications, including electronic commerce, news services, digital libraries, multi-player gaming, and interactive multimedia.

Thirty years after its conception, the Internet today continues to be remarkably successful at satisfying its two principal goals as summarized above. From virtually day

one, the Internet has been successful at increasing "research productivity through resource sharing." Further, the emergence of a number of new compelling Internet applications means that the Internet will sustain its growth. This growth ensures that the Internet will continue its success in introducing new classes of interactions and new uses of interconnected computers.

In this spirit of supporting new and emerging classes of wide-area applications, the goal of this dissertation is to simplify programmable access to the rich set of available wide-area resources. To this end, this dissertation describes a number of new contributions in the areas of wide-area naming, persistent storage, and security. To frame our research contributions, we begin by describing our target distributed applications in the next section.

1.2 Application Domain

The amount of information connected to the Internet and the resulting class of online activities is growing at exponential rates. An understanding of the characteristics and requirements of these applications is critical to our efforts in designing a system that supports programmable access to wide-area resources. In this section, we describe a number of wide-area applications that motivate this dissertation. Some of these applications are available today in limited forms, while others are likely to emerge in the next few years. Further, a number of these applications are directly visible to end users, while others provide architectural support for higher-level distributed services.

- *SchoolNet*: One motivating application is providing Internet services such as email, Web page hosting, and chat rooms for very large numbers of school children. A

desirable feature of such a system is to allow geographically distributed children to be able to interact with one another, while keeping both the interactions and the identities of those involved private. Further, to be useful to school children, the system must work with only limited direction from the end users (e.g., you cannot expect fifth graders to correctly set up access control lists).

- *Wide-Area Collaboration*: Physically separated users should be able to utilize network resources to collaborate on a common project. For example, a software project's source code repository should be globally accessible to authorized users for check in/check out; in addition, unique hardware (such as supercomputers) should be seamlessly accessible independent of geographic location. An aggressive example of such collaboration is online voting, where users can securely place their votes through the Internet during general elections.

- *Geographically Distributed Internet Services*: If it were easy to geographically replicate and migrate Internet services, end-users would see better availability, reduced network congestion, and better performance (this will be further motivated later in this chapter). Today, only the most popular sites can afford to be geographically distributed; for example, Alta Vista [Dig 1995] has mirror sites on every major continent, but these mirrors are physically administered by DEC, manually kept up to date, and visible to end users. In the future, it should be possible to make all this transparent, to make it feasible for third party system administrators to offer computational resources strategically located on the Internet for rent to content providers; in the limit, content providers could become completely virtual, with the degree and location of

replicas dynamically scaled based on access patterns[1].

- *Large Scale Remote Execution*: Users should be able to exploit global resources to run large scale computations. For example, NASA is placing petabytes of satellite image data on-line for use by earth scientists in predicting global warming. It is impractical to access this information using the current Internet "pull" model; scientists need to be able to run filters remotely at the data storage site to determine which data is useful for download. These filters should have access to necessary input (e.g., the filter executables) and output files (e.g., files into which the results are to be stored) on the scientist's home machine. The remote computation environment should also be protected from any bugs in the filters written by the scientists.

- *Medical Applications*: In the future, network connectivity will be leveraged to perform remote diagnosis. This diagnosis will be facilitated by secure doctor access to patient medical records. Access rights to these records can be transferred from doctor to doctor, though the patient will retain overall control (e.g., access by a certain organization will alway be denied despite any doctor's wishes). In one long-range scenario, specialists will be able to perform surgery locally over real-time and 100% reliable wide-area links.

- *Extensible Cache Management*: Caching of Web content close to clients is important for improving network utilization and end-to-end latency for client requests. However, a significant portion of Web content cannot currently be cached because servers wish to maintain control over each access. For example, servers may wish to track hit counts

[1]Chapter 7 presents an example of one of the first implementations of such an application, Rent-A-Server.

for advertising revenues, or perform advertising banner rotation or other modifications for each client access. This limitation on caching adversely impacts the amount of consumed wide-area bandwidth as a majority of requests miss in local proxy caches. By taking advantage of secure remote computation resources, it is possible to allow server code to run in proxy caches. In this model, each access to a particular service's cache content results in the execution of code specific to that service (e.g., to perform hit counting or cache consistency).

Such programmable access to service content also allows for simplified implementation of content *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. for a particular user. Service programs running at proxy caches or client hosts can perform such customization, potentially further reducing consumed wide-area bandwidth and end-to-end latency.

- *Mobile Distillation*: Distillation [Fox et al. 1996, Fox et al. 1997] transforms pictures and text to better match the limited capabilities and resources of clients such as PDA's or hosts with slow network connections. 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 near the client. A better approach is to allow the distillation function to take place at arbitrary points in the network. For example, when wide-area bandwidth is limited, transforming an object

at the server can be more efficient than transmitting a large image to the client before reducing the image size.

1.3 Motivation

1.3.1 Problem Statement

One approach to providing high performance, highly reliable wide-area services is to carefully choose a high bandwidth connection point to the Internet backbone. Multiple such links can be employed for redundancy to ensure high reliability[2]. Unfortunately, this approach is not viable because there is no single, centralized Internet site with universally good connectivity to all potential clients of a service. As a result of Internet topology and routing characteristics, clients follow very different and unpredictable paths in accessing a given service. For example, there are hundreds of regional and transit links that make up the Internet backbone, each with different performance characteristics. With a centralized service, it is likely that some fraction of clients are forced to traverse slow links in accessing the service. This probability increases with the logical distance[3] of the client from the service provider.

The first cause of unpredictable behavior in client access to a centralized server is highly bursty behavior in Internet access patterns, with an order of magnitude or more variance between average and peak link utilization. Such bursty access indicates a strong

[2]Constructing scalable services that are CPU-bound is beyond the scope of this dissertation. One successful approach is constructing the service itself from a cluster of workstations [Anderson et al. 1995a, Fox et al. 1997]. We will describe how such scalable services fit into the framework described by this dissertation in Chapter 9.

[3]In the Internet, the logical distance between two hosts can be very different from physical distance. In one example, a host in California must go through a router in Illinois to access a second host in California.

probability that a given link is over-subscribed at any given time. For example, the Web site of the Internal Revenue Service is usually overwhelmed in the weeks leading to the April 15 filing deadline. As another example, on days when there is heavy trading on the U.S. stock market, network congestion can prevent users of popular online trading services from not only receiving price updates but also from placing trade orders. As a final example, consider that while the packet switching technology that forms the basis of Internet routing is based upon research designed to survive nuclear war, many experts warned that the Internet could potentially collapse under the weight of the public release of a 400-page report detailing a White House sex scandal [Clausing 1998][4].

Another cause of unpredictable Internet behavior is the highly variable end-to-end latency and bandwidth between clients and servers. Latency is an important consideration when many small messages are exchanged (e.g., a distributed RPC system). The speed of light dictates approximately a 25 ms minimum one-way latency coast-to-coast in the U.S., with 50-60 ms latency values more typical once queuing and multiple hops are considered. Note that computers on a LAN can communicate with 1 μs latency [Boden et al. 1995]. Thus, developers of wide-area distributed applications must cope with 5 orders of magnitude performance variation in latency when moving from local-area to wide-area systems. While related to latency, bandwidth determines performance when bulk data is transferred between end hosts (e.g., for large file transfers or streaming of multimedia data). Clients access the Internet over links with highly variable bandwidth characteristics, varying from 14.4 Kbps modem links to 622 Mbps OC-12 links (and soon 2.5 Gbps OC-48). As with latency,

[4] Such predictions proved to be inaccurate. Severe slowdowns were reported across backbones and sites making the report available became slow or inaccessible. Further, news services such as CNN redesigned their pages to eliminate any graphics during times of peak demand. CNN reported a peak access rate of 340,000 accesses per minute, more than twice the level of previously recorded highs.

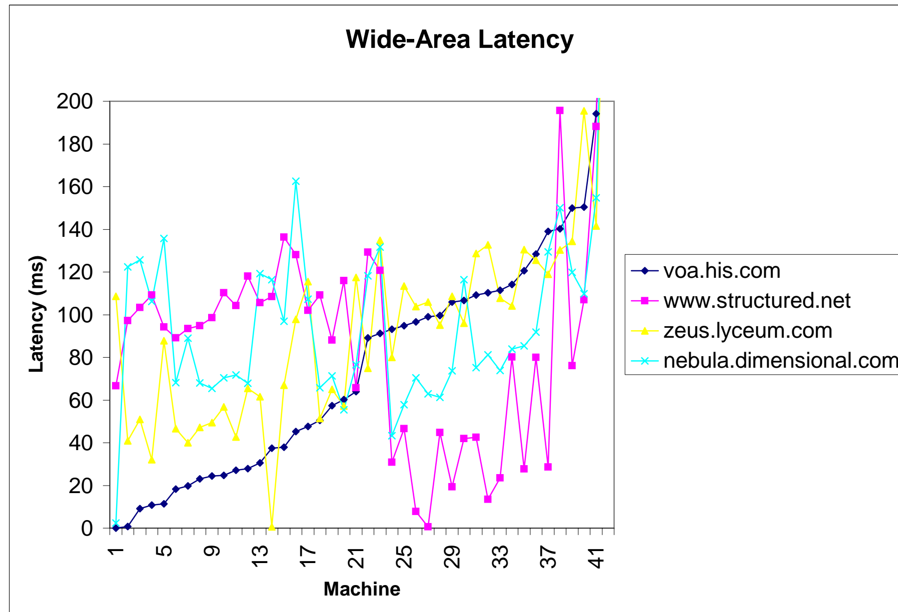


Figure 1.1: Average client latency to four Internet servers.

developers of wide-area applications must address four to five orders of magnitude difference in client bandwidth characteristics. Further, available latency and bandwidth are typically reduced by congestion (relating back to bursty access patterns) in Internet links because an over-utilized link queues and drops packets, degrading both latency and bandwidth.

1.3.2 Experimental Validation

We conducted an experiment to validate the performance limitations of centralized services. We chose 43 servers in the United States to take latency and reliability measurements among themselves (creating a 43x43 grid of latency/reliability measurements) over a four-week period during May and June of 1998. The 43 machines all provided a public

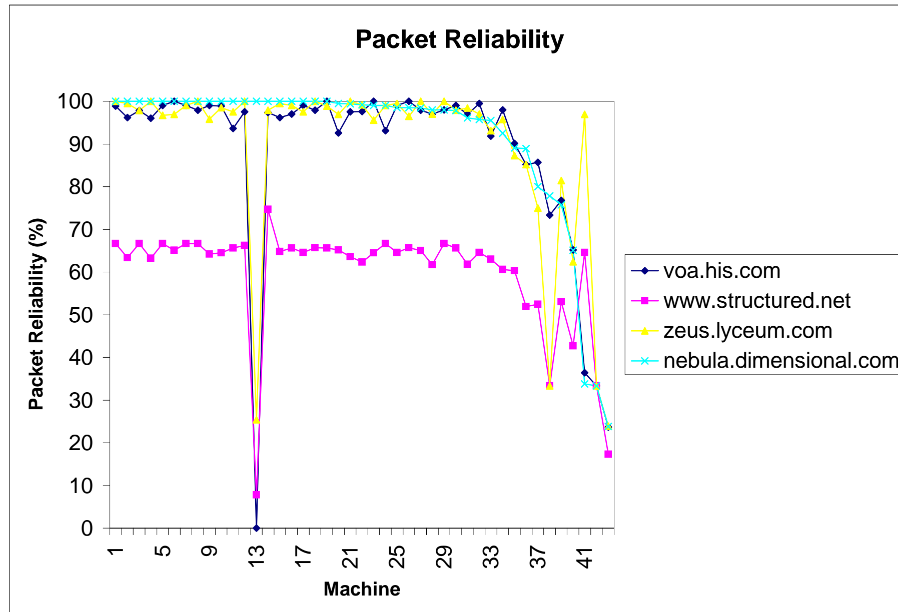


Figure 1.2: Percentage of packets successfully transmitted to four Internet servers.

interface to the "traceroute" service, allowing a remote client to measure the characteristics of a network path between the server and a site chosen by the client. Figures 1.1 and 1.2 summarize the results of these measurements.

For the purposes of both Figures, the 43 machines were assumed to be clients of four Internet services located at different locations in the United States (while measurements to all 43 sites are available, we chose four representatives to facilitate presentation). Figure 1.1 plots time-averaged latency for the duration of the trace (on the y-axis) for each client machine (numbered on the x-axis) for the four representative "services." The clients on the x-axis are sorted based on increasing latency for the fastest of the four services (the traceroute server located at `voa.his.com`). Some servers, not shown, displayed significantly

worse latency (up to 361 ms average round-trip latency from all 43 sites). While the server at `his.com` displays the best overall latency, almost half of the 43 client machines had better average latency to two alternative servers, `dimensional.com` and `structured.net`. We also calculated the standard deviations for the average values reported in Figure 1.1. Standard deviations of 50-75% were common across all clients to all servers. This high variance in latency demonstrates the highly bursty characteristics of Internet traffic, significantly skewing perceived latency from baseline values. Further, at any given point in time, the fastest sites (as measured by average latency across the duration of the trace) may actually perform worse than typically slower sites.

In addition to calculating average latency, we also calculate the reliability of a particular path by tracking the number of failed traceroute requests in our trace. Note that failed requests are not averaged into the results presented in Figure 1.1. Figure 1.2 plots the percentage of successful traceroute requests (on the y-axis) for each client machine (numbered on the x-axis) for the same four representative servers used in Figure 1.1. The client machines are sorted in a different order than presented in Figure 1.1, based on decreasing client reliability to `dimensional.com`, the most reliable of the four sites. Average reliability varied from 59-91% for the four servers, with half of the 43 machines displaying reliability values greater than 87% to the four servers. An interesting result from Figure 1.2 is that the site with the highest average latency in Figure 1.1, `dimensional.com`, actually shows the best overall reliability. This result implies that determining a service's optimal location cannot simply be based on best-case latency since dropped packets can adversely affect both delivered latency and bandwidth. Another result is that, once again, the site with

the best overall reliability is not the most reliable for all clients. In fact, one client (from `berkeley.edu`, labeled 42 on the x-axis of Figure 1.2) had 64% of its packets to the most reliable site (in the average case), `dimensional.com`, dropped, while only 3% of its packets to `lyceum.com` were dropped. Further demonstrating widely varying reliability characteristics, another client (from `lara.on.ca`, labeled 14 on the x-axis of Figure 1.2) shows 100% reliability to `dimensional.com`, but less then 25% reliability to the other servers.

The results from Figures 1.1 and 1.2 imply that a centralized server cannot provide high-performance and highly reliable service to all clients. In fact, since all servers/clients from Figures 1.1 and 1.2 were located in the United States, actual performance and reliability measures are likely to be worse when we take a more global view of performance. For example, the performance of clients in Europe or Asia accessing a centralized service in the U.S. will be limited by the high latency of crossing the Atlantic or Pacific and also the increased contention for a smaller number of inter-continental links (meaning more dropped packets, especially during times of peak demand).

1.4 Our Approach

Our measurements and observations indicate that centralized services are unlikely to deliver optimal performance or reliability to clients. This inadequacy naturally argues for migrating services (either wholly or in part) across the wide area. Given the fact that a single centralized site cannot deliver optimal performance to all clients, it follows that harnessing multiple sites at strategic points in the Internet can result in improved performance and availability. Thus, the key insight motivating the individual contributions

of this dissertation is the realization that providing applications programmable access to remote resources will allow for the construction of wide-area applications with the following advantages:

- *Improved End-To-End Availability:* By distributing service-specific extensions throughout the network, services transparently mask network or server failures from end clients (e.g., by automatically transmitting requests to alternative replicas upon individual server failure).
- *Improved Cost Performance:* Wide-area network resources suffer less congestion and buffer overflow as more client requests are directed toward nearby replicas, instead of traversing multiple network links traveling to distant replicas.
- *Improved Burst Behavior:* Upon a spike in demand, services dynamically recruit resources across the wide area to maintain a desired level of client-perceived system performance.

This dissertation explores the impact of programmable access to remote resources on all aspects of wide-area applications, proposes solutions to the open problems, and evaluates these techniques with real-world wide-area applications.

1.5 Thesis and Contributions

The thesis of this dissertation is that remotely programmable network elements allow for a restructuring of wide-area systems that will improve wide-area resource utilization, simplify application development, and improve end-to-end performance. Transparent and

programmable access to remote resources requires a rethinking of all aspects of distributed services. Thus, to support our hypothesis we propose novel solutions to address the unique requirements of distributed applications for wide-area naming, persistent storage, and security. A principle that cuts across all of our solutions is flexibility. The heterogeneity of the wide-area network, clients, and service providers dictates that no single policy is appropriate for all applications in all situations. By including flexibility in our solutions from the ground up, we ensure that our techniques are applicable to a wide variety of distributed applications. Further, with flexibility, our techniques are likely to remain applicable as the Internet and its applications continue to evolve.

To support this thesis, we make four primary contributions in this dissertation:

1. We show how programmability can be added to wide-area naming systems. We demonstrate that existing techniques for locating wide-area resources do not provide optimal performance for all applications. This motivates the need for flexibly matching resource location techniques to application semantics. This flexibility is interposed behind the naming interface to allow transparent migration of client and service functionality into the network.

2. We demonstrate the value of combining communication and persistence in a location-independent persistent storage system. Wide-area applications can often trade coherency guarantees for increased performance and availability. We show that flexible, programmable cache coherence protocols allow applications to specify the protocol appropriate for their coherency and performance requirements. We further demonstrate that it is possible to automatically cache a certain class of dynamically generated Web

content and that such caching significantly improves Web server performance.

3. We devise techniques enabling the construction of a wide-area security system with the following properties: high performance and availability despite network limitations, fine-grained control over remotely programmable resources, and rights transfer and revocation between multiple administrative domains. Existing security systems do not work well in a wide-area, agent-based (i.e., mobile computation) environment because of implicit assumptions such as central control, homogeneous hosts, and highly reliable/high performance networks.

4. We show how the above elements can be brought together to enable and simplify the development of a number of wide-area applications, including a Web server capable of dynamically replicating itself across the wide area in response to client access patterns. Our experience qualitatively shows that application development is simplified by our techniques for programmable access to remote resources. Further, our experiments quantitatively demonstrate improved utilization of network resources (e.g., bandwidth) and improved end-to-end system performance relative to traditional centralized approaches.

1.6 Organization

The individual contributions of this dissertation are described through WebOS, a complete interface to remotely programmable resources. Chapter 2 states the goal of our system design, describes the requirements of wide-area applications, and presents an overview of a system design to meet these goals and requirements. The next four chapters describe

individual contributions in the area of system support for wide-area applications. Chapter 3 describes programmable naming of wide-area resources for fault-tolerant, load-balanced access to global services. Chapter 4 presents the benefits of combining communication and storage in a global cache coherent file system. Chapter 5 shows how the global file system is leveraged to enable caching of dynamically generated Web content, improving, for example, the performance of many Internet services. In Chapter 6, we present a fault tolerant security and authentication system supporting the fine-grained transfer of privileges across the wide area. We then show how our individual contributions, as embodied by the WebOS framework, simplifies the implementation of a Web server capable of dynamically replicating itself across the wide area in response to client access patterns in Chapter 7. Next, Chapter 8 summarizes related research. This dissertation concludes with lessons learned, future work and a summary of our work in Chapter 9.

Chapter 2

System Goals and Design

In the previous chapter, we motivated the need for constructing a system that supports transparent application access to remote resources. The thesis of this dissertation is that such a system will allow a restructuring of wide-area applications that will improve wide-area resource utilization, simplify application development, and improve end-to-end performance. It is necessary to design and build such a system to demonstrate its utility in supporting real-world applications.

In this chapter, we describe the design and implementation of the WebOS infrastructure that supports wide-area applications. We begin by describing the high level goals of the system. These goals will be revisited as we describe in detail individual contributions in subsequent chapters. Next, we describe the requirements of a wide-area computation by considering the needs of a mobile Web service as an example. Finally, we describe the design of our system, corresponding to detailed descriptions of individual system components in Chapters 3-6.

2.1 Goals

As motivated by the thesis of this dissertation, we have the following set of high-level goals for the design of a wide-area system supporting transparent access to remotely programmable resources:

- *Uniform Interface to Remote Resources*: The first goal of our work is to make remote resources as easy to use as local ones. Thus, the system must export to application developers a uniform interface to computational resources such as processors and memory. For such access to be viable, the system must deliver security guarantees to both the application developer and remote resource providers.

- *Flexibility*: The highly heterogeneous performance and availability characteristics of wide-area resources described in Chapter 1 dictate the need for flexibly setting the policy behind any wide-area abstractions we develop. Bursty access patters and intermittent availability mean that application developers must cope with orders of magnitude differences in, for instance, computing power, network latency, and network bandwidth. Support for wide-area applications is further complicated by the highly variable requirements of wide-area applications, ranging from simple Web publication with loose consistency and performance requirements to medical applications which require strict consistency, performance guarantees, and high availability. To address heterogeneity across all this axes, a goal of our system design is to support flexible policies independent of individual mechanisms. For example, we will describe cache coherent access to persistent storage. However, we will not dictate the proper cache coherence policy for all applications. Instead, we will provide a number of dif-

ferent policies, allowing application developers to choose the appropriate one. We further allow application developers to extend existing policies if existing ones are not appropriate.

- *Performance*: In the end, providing convenient access to remote resources will not be adopted by many application developers if application performance is significantly reduced. Thus, a goal of our work is to design new techniques for high performance access to remote resources behind system abstractions. In cases where we improve semantic access to remote resources (e.g., providing security between administrative domains), it is important to minimize any associated overhead. Note that the goal of performance is intertwined with the above goal of flexibility, as flexible policies are often necessary to deliver the highest level of application performance

- *Backward Compatibility*: To ensure wide adoptation, it is important for the system to be compatible with existing applications. The disadvantage of this approach is precluding certain design decisions and innovations because of incompatibility with existing applications. In such cases where new development techniques are necessary to innovate in wide-area systems (e.g., naming, as described in Chapter 3), we take the approach of providing a migration path for an existing application to take advantage of our approach without requiring complete reimplementation of the application.

2.2 Requirements

Given the above set of goals for wide-area applications, we now explore some of the requirements for supporting wide-area applications. As motivated in Chapter 1,

programmable access to remote resources presents a number of compelling advantages for wide-area applications and services. However, we must address a number of challenging technical problems before such programmable access becomes a reality. Such programmable access imposes the following principal requirements from the underlying system:

- *Migrate Data:* Applications move data across the wide area to take advantage of the rich set of information available in databases stored across the Internet or, more simply, to move data closer to where it is required. To store and retrieve this data, applications require access to persistent storage space. Access to this data must be location-independent and cache-coherent.

- *Migrate Code:* Once data has been migrated across the wide area, applications will naturally wish to migrate code, either to execute closer to the data or simply to take advantage of available remote computation power. To realize this requirement, applications need access to safe remote execution of their code. Today, Java [Gosling & McGilton 1995] allows portable code to be run at multiple sites, while sandboxing tools such Janus [Goldberg et al. 1996] provide backward compatibility for the safe execution of UNIX code.

- *Locate Mobile Data and Code:* Once data and code are migrated across the wide area, applications must be able to locate data and code given its name. On a single machine, operating systems traditionally provide this functionality by maintaining mappings of names to location. Naming in the wide area, however, is more complicated. Objects frequently migrate from host to host, making it difficult to globally track the location of all objects. Further, many objects are replicated (e.g., a service available at multiple

locations). Heterogeneity in wide-area network performance means that the optimal replica choice will vary from client to client. Thus, the system must perform name resolution in a client and service-specific manner.

- *Secure Access to Data and Code:* In conducting financial transactions or executing sensitive code on private data, applications require guarantees regarding secure access to their data and code. Such assurances are typically provided through authentication, the ability to unambiguously prove and to verify an identity, and access control, the ability to specify the principals allowed to access a resource and to ensure that only those principals are ever granted access. The presence of multiple administrative domains with separate and autonomous security policies complicates security in the wide area. For example, principals place different levels of trust in different administrative domains (e.g., code running in the local domain runs with all of a principal's privileges but code running in a remote domain runs with only a subset of privileges). Changing trust relationships means that a wide-area security system must support the fine-grained transfer of privileges between administrative domains and that privileges must be revocable.

2.2.1 Example: Rent-A-Server

To make the above set of requirements for wide-area applications more concrete, this section considers the requirements of a wide-area application, Rent-A-Server, that exercises all of the above requirements. Chapter 7 describes the design, implementation, and benefits of this application in detail. The previous chapter demonstrated that a centralized

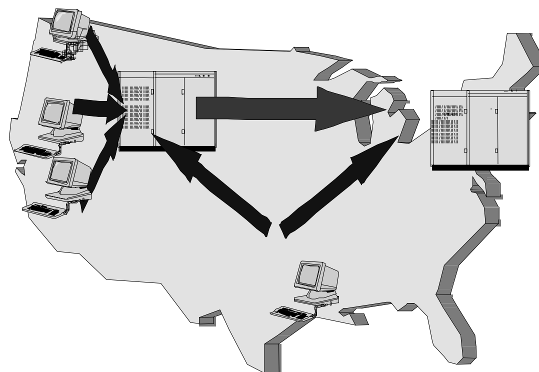


Figure 2.1: The Rent-A-Server application.

Web service cannot deliver optimal performance to all clients. The insight behind our design of Rent-A-Server is to allow for the deployment of *virtual services*, a Web service capable of dynamically replicating and migrating itself geographically in response to client access patterns.

Figure 2.1 pictorially depicts the operation of the Rent-A-Server application. Rent-A-Server is applicable to services that are either overloaded (limited by CPU resources) or simply unable to deliver acceptable performance to end clients (e.g., because of network congestion). Figure 2.1 shows an overloaded server in the Midwest region of the United States dynamically replicating itself to the East coast. Each Client requests service from the replica able to deliver the best performance to that individual client.

We now consider the requirements of building Rent-A-Server from the application developer's perspective. First, the Web service must be able to *migrate data* across the wide area. This data takes the form of, for example, HTTP files of the service, CGI executables, and the executable for the server itself to run on a remote host. Replication of data must

be performed in a cache consistent manner. When a service maintains HTTP files (e.g., product pricing) on a centralized site, any updates are automatically seen by all clients. However, if the data is replicated, the system must ensure that any updates made at one site are propagated to all replicas so that clients do not access stale information.

The next requirement is to *migrate code* geographically. Servers must run code on remote machines to take advantage of geographic locality in client access patterns. For example, if a service in the United States is experiencing heavy demands from Europe, it can make sense to execute a replica of the server's code at a site in Europe to provide a fast path for a majority of clients. Because sites may be running arbitrary programs, the remote execution system must provide guarantees that buggy or malicious code can only hurt itself and cannot compromise other local programs or local resources.

Once data and code are migrated across the wide area, applications must be able to *locate mobile data and code.* As described above, this is the traditional problem of naming. Clients of Rent-A-Server will ideally have a single name for the service (e.g., CNN) and the system will be responsible for translating this name to the replica able to deliver the best performance to the clients. As will be further motivated in Chapter 3, the naming system must support flexible policies for binding clients to replicas because a simple technique such as routing requests to the "nearest" replica will not always result in optimal performance.

The final requirement for Rent-A-Server is *secure access to data and code.* While HTTP files migrated geographically are generally public, other service state, such as the executable program or a customer database may contain sensitive information. The application must be able to specify exactly which sites and which users have access to particular

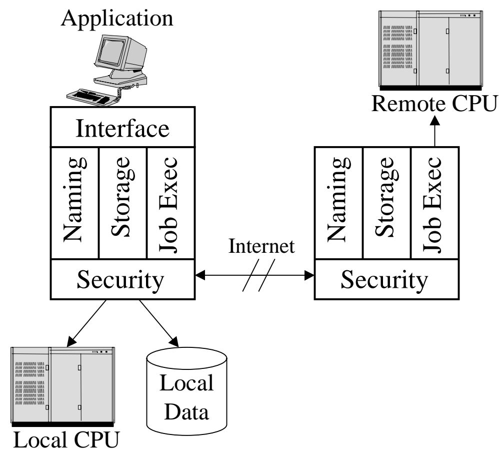


Figure 2.2: High-level system design.

data and code. For instance, a program listening on a wide-area link should not be able to ascertain the contents of sensitive files. Similarly if a remote site is executing multiple Web services, as would be the case with a Web hosting service, the system must ensure the proper isolation of access rights among concurrently executing Web services. For example, a site concurrently hosting Rent-A-Servers for both Netscape and Microsoft should not allow process's from one service to access data files from the other.

2.3 Design

2.3.1 Design Overview

This section presents the architecture of a system designed to meet the goals and requirements of supporting transparent access to remotely programmable resources, as outlined in earlier sections of this chapter. Figure 2.2 presents the design of the system at

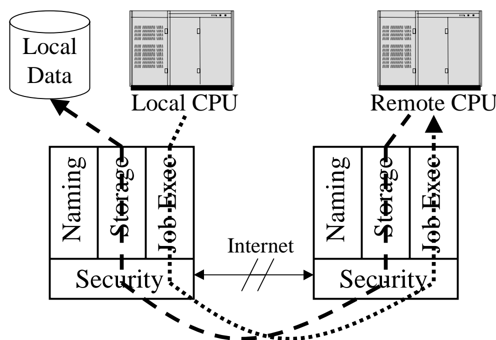


Figure 2.3: Transparent access to remote resources.

a high level. Applications are presented with a uniform interface to global computational resources. The three primary system interfaces include naming, persistent storage, and job execution. A security system cuts across all three interfaces, allowing, for example, authenticated access to global data or remote computational servers. Individual interfaces allow for existing applications to take advantage of new functionality (i.e., transparent access to network resources) without, in many cases, requiring redesign or even recompilation of the application.

The system interface attempts to make remote resources as easy to access as local ones. Thus, the job execution interface allows for the spawning of processes both locally and remotely. For remote execution, the job execution system on one machine contacts its counterpart on a remote machine to request remote execution, as illustrated in Figure 2.2. The security system mediates all cross-domain requests to ensure that the communication is secure (through encryption), that the identity of the requester is authenticated, and that access control lists are properly consulted for authorization on the remote site.

To illustrate the utility of the system design, consider the following example. A

user wishes to spawn multiple simulations. To distribute load, some of the simulations will run on a local processor, while others will run on a remote processor. Job requests for both local and remote execution will utilize the same interface. For local execution, the request will flow from the application to the job execution system. In turn, the security system is used to authenticate the identity of the principal requesting execution and also to determine whether the principal is authorized to run jobs on the local processor. If so, the requested job is spawned in a virtual machine to ensure that a buggy or malicious program does not compromise local system integrity. The steps for remote execution are similar and summarized in Figure 2.3. The request for remote execution flows from the local processor through the job execution system and into the local security system. The local security system then establishes a secure channel to the security system on the target host. The job execution request is transmitted over this channel along with the identity of the requester. Once the request reaches the remote host, authentication, authorization, and execution in a virtual machine proceeds as in the local case.

Continuing with this example, simulations spawned on remote hosts will still require data files stored on the user's local host. To satisfy such data requests, the application utilizes the system interface to access the persistent storage system as described in Figure 2.3 (e.g., using a globally unique name provided by the naming system to uniquely identify the desired data file). Requests flow from the remote processor into the storage system and then to the security system. The security system establishes the identity of the process requesting the data file. As with job execution, the security systems on the remote and local nodes establish a secure channel to transmit the file request along with the identity

Operation	**System Support**	**System Component**
Migrate Data	Persistent Storage	*WebFS*
Migrate Code	Safe Remote Execution	*Java and Janus*
Locate Data and Code	Naming	*Active Naming*
Secure Access to Data and Code	Authentication and Access Control	*CRISIS*

Table 2.1: Required wide-area operations and corresponding system support.

of the process making the request. The local storage system, in cooperation with the local security system, determines if the requester possesses the proper set of privileges to obtain the identified data file. If so, the contents of the file is returned encrypted to the remote node for processing by the simulation. The remote storage system caches the contents of the file for optimal performance, avoiding the need for potentially slow wide-area access on subsequent file accesses. The local and remote storage systems cooperate to provide cache consistency in the case where the data file is updated to ensure proper behavior of the simulation program.

2.3.2 System Organization

Given the high level description of system components in our design we now present the specific components, loosely corresponding to the individual contributions of this dissertation summarized in Chapter 1. Table 2.1 presents a description of the system components used to demonstrate the contributions of this dissertation. The operations in the first column of the table correspond to the requirements presented in Section 2.2. Each of these requirements corresponds to required system support described in the second column of the table. The third column corresponds to the names of the system components described in

subsequent chapters. This dissertation is organized around presenting the design, implementation, and novel features of a system supporting the development and deployment of emerging wide-area applications.

Each chapter of this dissertation describes design decisions and contributions associated with the system components presented in Table 2.1. For each component, we describe how our design satisfies the goals presented in Section 2.1 and how our contributions support and prove the thesis of this dissertation (from Section 1.5). We close this chapter with a brief overview of the rest of this dissertation.

In Chapter 3, we describe Active Names, a system that interposes location-independent and application-specific programs behind the traditional naming interface. Using this interface, we are able to match required application semantics with the behavior of the system in locating wide-area resources. We demonstrate how Active Names meets the wide-area design goals described in this chapter through experience with the development of a number of compelling wide-area applications.

Chapter 4 presents the design of WebFS, a wide-area persistent storage system with flexible cache coherence policies. Applications can tradeoff coherence guarantees with improvements in performance and availability. We demonstrate that it is possible to build and deploy a wide-area file system that provides multiple coherence policies, and that such flexibility improves application performance and simplifies development.

Chapter 5 builds on the contributions of WebFS to solve an outstanding problem with wide-area caching. Currently, caching of wide-area content is employed to reduce client latency and reduce consumed wide-area bandwidth in accessing Web resources. However,

an increasing percentage of content cannot be cached because it is dynamically generated at servers in response to client requests. We demonstrate that profiling program execution through Transparent Result Caching (TREC) allows for caching of a certain class of dynamically generated content at multiple points in the network. Such caching reduces latency, consumed bandwidth, and server utilization.

Chapter 6 demonstrates that is possible to build on existing principles of secure system design to build a wide-area security infrastructure that supports mobile agents. Existing security systems are not appropriate for mobile agents because of a number of assumptions inappropriate for the wide area, including the nature of trust between administrative domains, network performance, availability. Through the design and deployment of CRISIS, we demonstrate how to build a security system that supports mobile programs. Further, we introduce transfer certificates to simplify the design and implementation of the system, with the additional benefit of making it easier to reason about correctness.

This dissertation does not make a new contribution to mechanisms supporting code migration. However, executing remote code is fundamental to a general-purpose system supporting wide-area applications. Chapter 8 describes how we leverage two existing systems, Java and Janus, to support code migration.

Chapter 7 describes the design and deployment of a wide-area application that exercises each of the the individual system components and contributions described in earlier chapters. We show how our wide-area infrastructure simplifies the development of the Rent-A-Server application described earlier in this chapter. Measurements of the application demonstrate wide-area performance improvements relative to existing techniques for

deploying high performance and highly available Web services.

Chapter 3

Active Naming

This chapter describes the first contribution of this dissertation, Active Names, a mechanism allowing applications to interpose programs behind the naming interface. As described in the previous chapter, replication and migration can make it difficult for wide-area applications to locate geographically distributed data and code. The naming system must be flexible enough to support multiple techniques for locating and retrieving resources on an application's or end user's behalf. Active Names introduce programmability into the naming interface, allowing applications to change the behavior of the naming system through location-independent programs.

3.1 Overview

3.1.1 Background and Motivation

People use descriptive and human-understandable names to identify resources. To be useful in computation, a name is traditionally translated into an address identifying the

exact location of a resource. This functionality is typically provided through the *naming* system. For example, users employ URL's to access Web services today. A URL such as `http://now.cs.berkeley.edu/WebOS` is the name of a resource (the WebOS project home page in this case). Before retrieving this resource, the naming system must first be employed to determine the address of the host containing this resource. In this example, the Domain Name System (DNS) [Mockapetris & Dunlap 1988] is employed to translate the hostname embedded within the URL (now.cs.berkeley.edu) into an IP address (128.32.44.96). This computer-understandable address is then used to contact the host for the target resource.

Traditional naming systems, such as DNS, typically make the following assumptions: i) if a name maps to multiple addresses, all mappings are considered equivalent (no per-client or per-service information is maintained), ii) bindings change slowly, if at all (thus, weak consistency is sufficient for updates), iii) updates for a given name are made at a centralized site, and iv) a single algorithm is appropriate for performing all name lookups and updates. To achieve our goal of providing transparent high performance and highly available access to remote resources, we require a naming system that invalidates all of these assumptions. As discussed in Chapter 1, resources are replicated and frequently migrating in our model (e.g., Web services growing and shrinking to meet client demands). A naming system for access to globally replicated services must maintain mappings between a name and multiple addresses. Further, this mapping can be updated frequently with updates coming from multiple sites. Finally, given the heterogeneous nature of the network and the unique requirements of individual clients and services, the naming system must employ multiple application-specific name resolution mechanisms for optimal performance

(e.g., current network connectivity along multiple paths, the location of the client making the request, or the processing power and relative load on the replicas).

Another shortcoming with current naming systems is the difficulty in coping with failures because a name is usually bound to a single address (Internet host). If that host becomes unavailable, the service is often unable to communicate the presence of alternate sites to the client. For example, because of highly variable network performance, 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 access. The current model of binding to an IP address, and then accessing a resource at that address makes it difficult to switch replicas midstream (similar problems arise in implementing mobile IP [Ioannidis & Maguire 1993]). Further, performing failover on replica failure must be programmable and service-specific. For instance, failure semantics are very different for an Internet chat [Yoshikawa et al. 1997] application where loose consistency and lost updates are tolerable than for a meeting scheduling program [Terry et al. 1995] where strict consistency and well-defined ordering is essential.

Finally, wide-area naming services often add unneeded latency to wide-area requests. Retrieving the binding of a name is rarely a goal in itself. Rather, it is a step in a larger process (e.g., logging into a remote machine or retrieving a Web page). However, because the naming service is logically separated from other wide-area services, clients must engage in separate round-trip communication with a name service before accessing the service they are actually interested in. For example, utilizing hierarchical Web caches can have a negative impact on client-perceived latency; once the page is located, it must travel all

the way back down the hierarchy before reaching the client. Further, as described above, browsers translate hostnames to IP addresses through DNS before retrieving Web pages. Unfortunately, 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, which can at times contribute to wide-area congestion.

3.1.2 The Active Name Approach

In this chapter, we present the design and implementation of Active Names to overcome the shortcomings of existing naming systems, as described above. Active Names are 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.

The principal contribution of Active Naming is a unified framework for extensible and application-specific naming, location, and transformation of wide-area resources. Specifically, 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 by integrating naming with higher level Web services.

In addition to the above advantages, Active Names also provide a convenient way to express tradeoffs in function versus 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 time efficient to ship the distillation function to the data at the server. However, if the situation reverses and the server becomes overloaded, then retrieving the data across the network and performing distillation locally is likely to be more efficient. Once again, the key observation is that the decision must be application-specific and adaptive to changes in capacity and workload.

Chapter 1 presented a number of wide-area services to motivate the need for programmable access to remote resources. Section 3.3 of this chapter describes how Active Names are used to implement four of those applications. With *mobile distillation*, 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 *extensible cache management*, we show the potential benefits of executing service-specific Active Name code to manage portions of proxy caches on a client and service-specific basis. For

replicated service access, Active Names provide the requisite flexibility to direct requests to the replica likely to deliver the best performance. Finally, we show how to implement *customization* through Active Name programs that maintain user preferences and filter site contents on behalf of services. While versions of each of these applications have been built in other contexts, the existing systems are limited by the lack of a general framework for flexibility, deployment, composability, and resource allocation.

The rest of this chapter is organized as follows. Section 3.2 presents the Active Naming architecture. Section 3.3 describes the implementation of the four sample Active Naming applications. Section 3.4 presents related work and we conclude in Section 3.5.

3.2 Architecture and Implementation

In this section, we describe the architecture and implementation of a naming system able to support the goals outlined in the previous section. At a high level, our system provides a framework for binding names to programs and for chains of programs to cooperate in translating a name to the data that name represents. Active Names are strings that take the following format:

`namespace/name`

The namespace field of an Active Name uniquely identifies a *Namespace Program* to be run to translate the name field of an Active Name into the resource it represents. The technique utilized to locate this Namespace Program is described in Section 3.2.1. A Namespace Program 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. Active Name evaluation

requests are transmitted to *Active Name Resolvers.* These resolvers are arranged in a cooperating mesh distributed across the wide area, with a local resolver assumed to be available on client machines or on local proxies in the case of under-powered clients. Active Name Resolvers export a standard environment for the execution of Active Name code. The resolver allows, for example, access to a local cache and also enforces security and resource limitations. In designing the interface to system resources, 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.

Namespace programs are location independent, meaning that they can be evaluated at the Active Name Resolver capable of most efficiently evaluating a particular Active Name. Responsibility for name resolution is often successively passed from resolver to resolver (e.g., as in locating an object in a distributed cache with a hint-based directory [Sarkar & Hartman 1996]). Since name resolution can take place multiple hops away from the client, our name resolution model supports 3-way RPC's. In this way, we reduce latency by avoiding multiple round-trip communications in returning the resource represented by an Active Name back to the client. With 3-way RPC's, the resolver that successfully evaluates an Active Name transmits the resource directly back to the client. We support this model with *after-methods*, a list of additional Namespace Programs that run after evaluation of the initial Active Name completes. One instance of an after-method supports 3-way RPC's by encapsulating the state necessary to transmit results back to the client. After-methods also conveniently express client or service-specific filters on data once the raw resource is produced at an Active Name Resolver. After-methods are described in detail in Section 3.2.3.

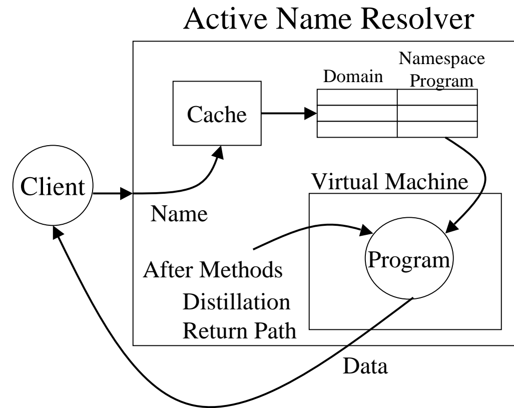


Figure 3.1: The Active Name Evaluation Process.

The process of evaluating Active Names is summarized in Figure 3.1. Clients transmit an Active Name to a nearby Active Name Resolver that is either co-located with the client or available on a local proxy. The resolver consults its local cache for the availability of the resource corresponding to the Active Name. Assuming a miss, the resolver consults a table mapping Active Name domains to the Namespace Program responsible for evaluating it. If the code is not available locally, techniques described in Section 3.2.1 are used to retrieve the code. The resolver executes the Namespace Program in a virtual machine to enforce security and resource consumption limitations. The resolver also associates a list of per-client after-methods with the Namespace program, e.g., a distillation program and program responsible for returning data back to the client once evaluation of the Active Name completes. Note that, if necessary, Active Name evaluation can be handed off to a remote resolver and that Namespace Programs can attach additional after-methods.

Figure 3.2 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

```
// An Active Name is made up of a namespace and a name within that
// namespace.  The namespace uniquely identifies a Namespace Program
class ActiveName {
  public ActiveName(String namespace, String name);
  public String GetNamespace();
  public String GetName();
}
// A Namespace Program is responsible for retrieving or generating the
// resource represented by an Active Name
abstract class NamespaceProgram {
  public void Eval(String name, ActiveName[] afterMethods,
                   DataStream data);
  public ActiveNameResolver GetActiveNameResolver();
}
// An Active Name Resolver is responsible for safe and secure execution
// of Namespace Programs.  It provides an interface to system resources
class ActiveNameResolver {
  public Cache GetCache(CodeSource cs);
  public NetworkDataStream MakeNetworkConnection(CodeSource cs,
                                                  String host, int port);
  public void Eval(CodeSource cs, ActiveName name,
                   ActiveName[] afterMethods, DataStream data);
  // For Remote Method Invocation on a remote resolver
  public void EvalAt(CodeSource cs, NamespaceProgram program,
                     String remoteHost);
}
```

Figure 3.2: Interfaces for the Active Naming prototype.

portability, object serialization, remote method invocation, and stack introspection [Wallach & Felten 1998]. To evaluate an Active Name, a client invokes the `Eval` method of an Active Name Resolver with four arguments: (i) the Code Source identifies the caller and is used for security checks by the resolver (described in more detail in Section 3.2.2), (ii) the Active Name specifies the name to be evaluated, (iii) After-Methods are, by convention, invoked in order after the original program associated with the Active Name is completed, and (iv) the Data Stream represents the result of evaluating an Active Name and acts as input to

after-methods; the Data Stream forms a pipeline of filters passed from program to program and, potentially, host to host.

Given this basic architecture, the following subsections describe the individual components of the Active Naming architecture in more detail.

3.2.1 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 3.2, the namespace component of an Active Name identifies a Java class that extends the base class `NamespaceProgram`. This program is 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.

Namespace Program 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 either producing or retrieving the resource represented by the Active Name. The Active Name Resolver enforces security and resource limitations on running Namespace Programs. 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

`RootNamespaceProgram`. The code for this class is available at every name resolver and 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`.

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 either a shorthand or a fully qualified format. The shorthand format is for user convenience; in these cases, a registry[1] 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 unmodified to the `Eval` method of a Namespace Program.

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 programmable time period. Classes can also serialize their content to local disk to maintain state information that may be beneficial on subsequent invocations of a particular class.

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

3.2.2 Active Name Resolver

Interface

Active Name Resolvers allow Namespace Programs to access local system resources through a uniform interface. Resolvers spawn a separate thread for the evaluation of each Active Name. The resolver gates accesses to all resources through a Java Security Manager and by enforcing resource limitations (as described below). The resolver is also used to insert and retrieve data from a local cache and to make network connections.

The interaction of the Active Name Resolver and classes responsible for Active Name evaluation is described in Figure 3.2. The `Eval` method on a resolver locates the class responsible for evaluating an Active Name, enforces security and resource consumption limits, and spawns a new thread to run the `Eval` method on a Namespace Program. The `GetCache` method is used to access the local cache provided by the Active Name Resolver. The cache ensures that different Namespace Programs see distinct partitions within the cache to locally store and retrieve objects. `MakeNetworkConnection` is used to communicate with remote machines. Finally, `EvalAt` allows instances of a Namespace program 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.

Security Guarantees

Our security model is oriented toward isolating untrusted Namespace Programs 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 Namespace

Program to sensitive system resources or to other Namespace Programs be via explicit calls to the Active Name Resolver interface, and (ii) requiring Namespaces to explicitly identify themselves in all calls to the Active Name Resolver so that the Active Name Resolver knows what security restrictions to enforce and to whom to charge resource consumption.

To enforce these requirements, the resolver uses Java stack inspection [Wallach & Felten 1998] as implemented in Sun's Java JDK 1.2. The system's `java.security.SecureClassLoader` associates all Namespace Programs with instances of `java.security.CodeSource`, from which the system loads the Namespace Program class[2]. 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 identified by the CodeSource. This collection is essentially a list of capabilities for the ProtectionDomain.

Our `ActiveName.SecurityPolicy` class is a subclass of `java.security.Policy` and ensures that Namespace Program 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 learn from that 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 Namespace Program makes a call to the Active Name Resolver, it includes its CodeSource as an explicit argument. The resolver invokes the stack inspection

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

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 resolver knows the identity of the caller making the request. This allows it to, for instance, restrict local disk cache accesses by a Namespace Program to a specific subdirectory in the system.

It might appear that Java stack inspection 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 file system, which makes it more difficult for the Active Name Resolver to enforce resource limitations.

Resource Limitations

Instances of Namespace Program can be arbitrary, untrusted code. Therefore, the Active Name Resolver 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 straightforward to manage because all accesses to the disk and network must go through the Active Name Resolver. The security policy ensures that direct access to the network and disk are automatically rejected. The methods on the resolver 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 processor resources on a per-request granularity. When a new request enters the system, the resolver assigns it to a thread group. The request

may spawn multiple sub-threads, but all consumed cycles are 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 to track and limit the resources consumed by each request. When the Active Name Resolver creates a new thread group for a new request, it registers the thread group with the scheduler. The scheduler spins in a loop that first sleeps for a random period of time and then performs accounting to determine the CPU resources consumed by all Thread Groups. This estimate is made 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.

The 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. Open issues with resource allocation are further discussed in Section 9.2.5.

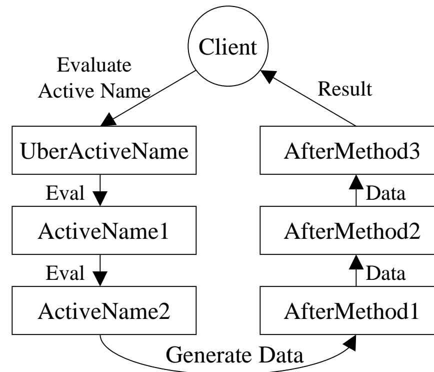


Figure 3.3: Interaction of after-methods Active Name Evaluation.

3.2.3 After-Methods

One of the parameters for the evaluation of any Active Name (the method `Eval`) is an array of Active Names called `afterMethods` (see Figure 3.2). 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 3.3, after-methods are typically responsible for returning the result of evaluating an Active Name back to the client (for 3-way RPC's), perhaps performing some transformation on the data along the way. By convention, each Active Name class promises to run the first element of the array of AfterMethods upon 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. In this case, the after-method (itself an Active Name) encapsulates a string containing a hostname and a port where the client awaits the result of evaluating the Active Name it originally generated.

Most Active Name programs 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-method 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 Chapter 1 is implemented through an after-method that transforms pictures into a format suitable to particular clients. As will be described in Section 3.3.1, 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.

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 3.4(a). The client first creates and runs a thread that will listen for the result of the

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

(a) Name Evaluation

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

(b) After-Method Invocation

Figure 3.4: Sample name evaluation and after-method invocation.

evaluation on, for example, port 8040 of the local host. Next, it creates an array of Active Names (the after-methods) 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 its local Active Name Resolver. `RootNamespaceProgram` evaluates the Active Name "yahoo/search" and determines that the class `YahooNamespaceProgram` is responsible for the "yahoo" namespace. `RootNamespaceProgram` 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 Active Name Resolver evaluate this Active Name, as summarized by the sample Java code in Figure 3.4(b). Evaluation of `ReplyNamespaceProgram` will cause the contents of the data stream (HTML for "yahoo/search") to be transmitted to the waiting client.

3.2.4 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. First, the user sits behind a slow modem and prefers small, black and white pictures to conserve bandwidth. Next, the HTML representing CNN's front page is customized to contain the news stories, stock quotes, and weather forecasts preferred by the user. Further, 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 (in fact, a separate impartial auditing service may be employed to track hit counts). Each customization can be thought of as a filter on data. These filters are implemented through Namespace Programs. 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 an array of after-methods containing two entries. The first is responsible for distillation, transforming images to match client requirements (in this case, reducing the size of the image and converting it to black and white). The second after-method is a program that knows how to transmit the result of evaluating an Active Name back to the waiting client. Note that these after-methods will not be run until after the name has been evaluated, and that 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. A potential downside to multi-way RPC's is difficulty at caching results at multiple steps in a hierarchy. This problem can be addressed by either backfilling caches in the hierarchy or by multicasting [Deering 1991] results to caches and the client simultaneously.

- 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 (e.g., hit counts) 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. As will be described in Section 3.3.1, significant performance advantages can be gained from flexibly determining the location in the network where

the distillation function executes.

3.3 Applications

3.3.1 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 connected to a wireless network. 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.

For distillation, clients, at a high level, name a Web resource but would like the resource transformed to be based on client-specific characteristics. Mapping distillation to the Active Naming model, 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. Further, because Active Names take advantage of 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 transformation 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 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 this case, it is usually more efficient to perform the transformation at the server and to 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 server's CPU is over loaded, then it can be more efficient to transmit the larger image to a proxy (or client) where more CPU cycles are available. 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 mobile 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 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 the distillation location. In one case, the Name Resolver at Duke runs

	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 3.1: Performance of mobile distillation implemented with Active Names.

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 network performance. However, variable wide-area network performance still results in an error rate of 2-4% in our measurements. The Active Name resolvers (including all distillation code) are compiled and run with the Java Development Kit, version 1.2 beta 4.

Table 3.1 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 larger (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 3.1 shows that with no load on the server, distilling the image at the server produces between 20-40% better performance than distilling 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 our measurements were taken at a time (a weekend night) when the network link was not congested and did not drop packets. Larger performance improvements should be seen from server distillation during the day when transmitting large amounts of data across the Internet is slower.

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 multi-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 in a tradeoff 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).

3.3.2 Extensible Cache Management

The explosive growth of Internet usage has led to a 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

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 3.2: Breakdown of misses in large ISP proxy cache.

network backbones [Thompson et al. 1997]. Caching of Web content at local proxies is an important technique for improving client-perceived performance and reducing consumed bandwidth. Unfortunately, the performance of current caching schemes is limited by low hit rates that are stubbornly near 50% [Abrams et al. 1995, Duska et al. 1997, Gribble & Brewer 1997, Arlitt & Williamson 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 7.6 million requests for 64.5GB of data by 23,080 clients[3] 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 6.4 million requests for 54.8GB. Assuming current caching technology – a 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. In this section we will break down the cause of the remaining cache misses and argue that Active Names (i.e., service-controlled programs mediating service access) can eliminate a portion of the misses.

[3] The ISP uses dynamic IP for dial-up modem users. The trace contains requests by 23,080 distinct IP addresses.

Table 3.2 summarizes the cause of the 59.8% of 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 3.2 is examined below along with an explanation of how Active Naming code could reduce misses of the specified variety.

- 20.7% of miss requests are for URLs that contain either the string "cgi" or the character '?', indicating that the URL's invoke 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). For example, if a Web server is personalizing content for individual users, techqniues such as those presented in Section 3.3.4 can be employed to allow personalization to take place at a proxy cache. Note that we describe another technique for caching dynamically generated content at both servers and proxies in Section 5.3.3.

- 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 a typical shared proxy cache, 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. For example, Ac-

tive Name programs can maintain service-specific information about how frequently certainly objects change, removing the need to check with the service on each client request.

- 9.2% of the miss requests contained 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 cacheable. For example, hit counts can be maintained by persistent Active Name programs and periodically transmitted back to the server. Similarly, service-specific coherency mechanisms can be employed to ensure that stale versions of rapidly changing objects are not returned to clients. In general, programs that do not require access to difficult to migrate state (i.e., a commercial database) can potentially be migrated to Active Name resolvers for better performance through locality.

- 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]. Service-specific Active Name code can use service knowledge of access patterns to drive prefetching (as opposed to forcing the proxy to determine such access patterns on its own). For example, many services display common URL access patterns (i.e.,

if URL A is accessed, there is a high probability that URL B will also be accessed). Active Name programs can use service-specific information to perform prefetching, potentially reducing a portion of the compulsory misses.

- 3.7% of the miss requests are redirections to a new server. Most of these redirections are used to perform load balancing. As described in Section 3.3.3, Active Name code can perform load balancing without requiring 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.
- 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 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.

3.3.3 Replicated Service Location

Today, many service providers are replicating their services across both the local and wide area to achieve better performance, scalability, and availability. A principal prob-

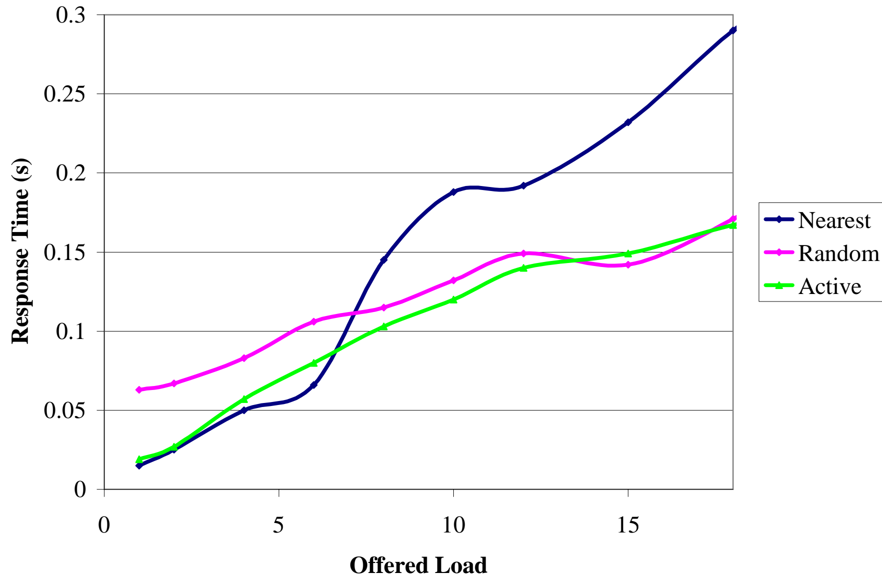


Figure 3.5: Performance of multiple wide-area load balancing algorithms.

lem 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 [Katz et al. 1994]). 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 where only portions of a site are actually replicated). For example, even considering only variable CPU load and network connectivity (bandwidth, latency), one replica will often be 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 dissertation to determine the appropriate replica-selection policy for any

particular service. However, Figure 3.5 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 attempt 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 following policies to choose among the replicas:

- *DNS Round Robin*: In this extension to DNS (labeled "Random" in Figure 3.5), 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 (e.g., all replicas are considered equal).

- *Distributed Director*: With this product from CISCO [Cisco 1997] (labeled "Nearest" in Figure 3.5), 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 some level of geographic locality for service requests[4].

- *Active Naming*: With this instance of programmable replica selection (labeled "Active" in Figure 3.5), 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

[4]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.

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 3.5 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 (varying from 1 to 18). 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.

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.

3.3.4 Personalization

The final application we describe 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 browser-maintained "cookie" uniquely identifying a user to an entry in the server's database describing such preferences. Such personalization is implemented with greater generality and location independence with 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 improve end-to-end latency by providing customization closer to the client. The Active Name program also tracks user accesses, ensuring that hit counts are correctly transmitted to the service (i.e., for advertising revenue). Further, the program can perform service-specific cache consistency. For example, a service might update a set of headlines every two hours, meaning that it it is unnecessary 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].

3.4 Related Work

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 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.

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 very successful. However, as new requirements are introduced it is difficult to expand such systems. A primary goal of Active Naming is to flexibly support new naming semantics as applications require them.

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 because any of them could be incorporated within the extensible Active Naming framework.

Prospero [Neuman 1992a, 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 from 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 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 3.3.2. 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).

3.5 Summary

This chapter describes 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, multi-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 chapter describes the utility 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.

The Active Naming system is designed to meet our goals for a wide-area computation system, as described in the previous chapter. Active Names provide a *uniform interface to remote resources* allowing applications to locate these resources independent

of where they are stored or replicated. *Flexibility* is a major component of the Active Naming design, allowing application and service-specific programs to use the most appropriate technique for locating and retrieving target resources. Active Names improve system *performance* through multi-way RPC's that reduce consumed wide-area bandwidth and client latency. Performance is also improved through, for example, improved load balancing and more functional proxy caching. Active Names may require modification to existing applications and services, but the system is *backward compatible* with legacy distributed applications by interposing naming functionality behind existing proxy caches.

Chapter 4

Global File System

The previous chapter described a programmable interface to the naming subsystem for wide-area application. This chapter describes WebFS, a persistent storage system that allows unmodified applications to access cache consistent files replicated across the wide area. WebFS provides location transparency, allowing the same name to be used to retrieve data independent of where a program is invoked, and it abstracts the wide-area communication necessary to access remote data by exporting the familiar file system interface.

4.1 Overview

Storing and retrieving data is fundamental to most distributed applications. Currently, file systems optimized for local-area networks [Walsh et al. 1985, Howard et al. 1988, Nelson et al. 1988, Anderson et al. 1995b] allow any machine connected to the network to transparently store and retrieve data located anywhere in the network. Cache

coherence policies are responsible for guaranteeing file consistency in the face of simultaneous readers and writers (though most LAN file systems do not provide strict consistency). Similar uniform access to global data will be required by applications and services as they migrate to the wide area. The high-level goal remains unchanged from the local-area case: to allow applications to read and write data physically residing anywhere in the network as if it were stored locally.

Global access to data requires a rethinking of traditional persistent storage. Experience with LAN file systems has taught us that there is a tradeoff between consistency guarantees, performance, and availability. A file system with strict consistency guarantees will generally suffer poor availability and performance. Improved availability and performance are usually granted by relaxing consistency guarantees. Traditional LAN file systems stake a claim at a single point in this space of consistency, availability, and performance. This point is picked to be appropriate for a majority of the applications targeted for the file system. However, a number of applications today do not run properly on LAN's because of improper consistency guarantees. For example, the popular UNIX `make` utility cannot be properly parallelized across a LAN running NFS [Walsh et al. 1985] because updates are not sent to servers for 30 seconds, meaning that certain file dependencies are not properly triggered.

In the wide area, problems with choosing consistency mechanisms is exacerbated by the sheer scale and performance of the Internet. Availability becomes more important as remote hosts are more likely to be unavailable because of network or end host failures. As motivated in Chapter 1, performance is highly variable because of heterogeneous wide-area

latency and bandwidth.

Continuing our theme of flexibility, in this chapter we demonstrate the performance and availability benefits of choosing cache coherence mechanisms in an application-specific manner. Thus, this chapter identifies and argues for the following requirements of a wide-area persistent storage system:

- *Global Namespace*: Applications should be able to access global data in a location-independent manner. Thus, applications should be able to use a single name to access data, independent of where the data is stored, replicated, or cached. While the initial prototype uses URL's to name files, we plan to leverage the Active Names described in Chapter 3.

- *Cache Coherence*: The weak coherency guarantees available in the Internet is sufficient for Web surfing. However, many Internet services require more strict coherency guarantees. Further, the consistency mechanism will help determine application availability and performance. Thus, the persistent storage system must support application-specific consistency policies.

- *Fault Tolerance*: In the scale of the Internet, individual host failures and network partitions are common. Wide-area services must employ techniques such as replication to identify and mask such failures.

- *Security and Authentication*: For Internet services to remain viable they must be able to authenticate the identity of clients as well as the identity of peer servers requesting access to global data. Security mechanisms for access to global data are discussed in

Chapter 6.

In this chapter, we argue that the above requirements can be addressed in a consistent and natural way by using the file system as the basic abstraction. To validate this claim, we built WebFS, a global cache coherent file system. Given the immense popularity of the Web, it is essential that this file system be compatible with the existing HTTP namespace. An initial WebFS prototype is complete and is operational on the Solaris operating system. WebFS is implemented as a loadable kernel module and can be mounted just like a normal file system. For example, if WebFS is mounted on `/http`, the following sequence of operations is supported for unmodified applications:

```
% cd /http/now.cs.berkeley.edu/WebOS
% ls
% cd papers
% ls
% lpr *.ps
```

Given this base-level functionality of access to the global HTTP namespace, WebFS provides a testbed for exploration of the cache coherence, security, and authentication issues associated with Internet services. We implemented two sample distributed applications, Internet chat and a stock ticker (providing regular stock price updates) to explore a number of cache coherence policies in the context of the Web. We have identified three policies appropriate for different classes of applications: last writer wins, append only, and multicast updates. Our experience in developing these applications and the matching coherence policies suggests that the use of application-specific cache coherence policies as an underlying abstraction can simplify the implementation of our target Internet services.

In particular, the integration of multicast into WebFS allows for efficient delivery

of frequently updated files to large numbers of clients interested in the most up to date file contents. As a motivating example, very popular Internet services often distribute identical content (e.g., a breaking news story) to all clients. Unfortunately, as described in Chapter 1 Internet services are often unable to keep up with the massive request stream from around the world. Given the fact that users are often interested in all file updates (e.g., election results or stock prices), we hypothesize that the use of multicast to deliver updates can improve client latency while simultaneously reducing server load.

The rest of this chapter is organized as follows. Section 4.2 details our implementation of WebFS and Section 4.3 summarizes system performance. In Section 4.4, we describe the three cache coherence policies currently implemented in WebFS and two sample Internet services built on top of these policies, chat and a stock ticker. Section 4.5 presents related work, and we summarize in Section 4.6.

4.2 Architecture

4.2.1 System Overview

Our goals for the design of WebFS are: (i) cache coherent access to the global namespace for unmodified applications, (ii) a fully functional file system interface supporting arbitrary file/directory operations, and (iii) performance comparable to standard file systems for cached access. To accomplish these goals, we decided to build WebFS at the UNIX vnode layer [Kleiman 1986] with tight integration to the virtual memory system for fast access to cached data. Using the vnode layer allows for portability across various UNIX variants and potentially Windows NT. For example, two popular distributed file systems,

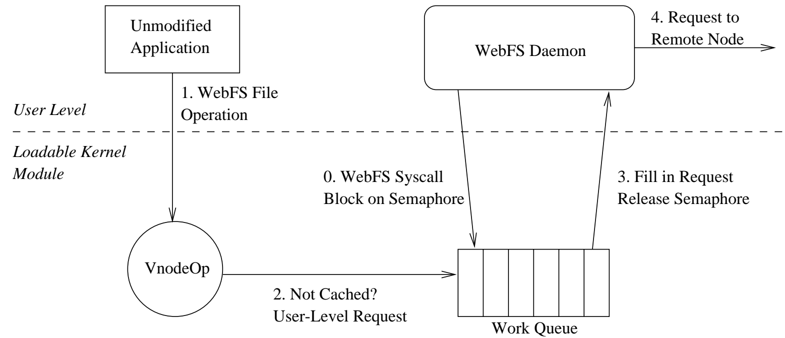


Figure 4.1: WebFS architecture.

NFS [Walsh et al. 1985] and AFS [Howard et al. 1988], were developed at the vnode layer.

Figure 4.1 summarizes the WebFS system architecture, which consists of two parts: a user-level daemon and a loadable vnode module. Consider the example of a read to a file in the WebFS namespace. Initially (step 0), the user-level daemon spawns a thread which makes a special WebFS system call intercepted by the WebFS vnode layer. The layer then puts that process to sleep until work becomes available for it. When an application makes the read system call requesting a WebFS file, the operating system translates the call into a vnode read operation (step 1). The vnode operation (step 2) checks to see if the read can be satisfied in the kernel (i.e., if the page being requested is cached in the kernel). If the operation cannot be satisfied in the kernel, the vnode operation fills in a structure requesting a read of the file page in question and wakes up one of the sleeping threads in the work queue (step 3). The user level daemon is then responsible for retrieving the page, by contacting a remote HTTP or WebFS daemon (step 4).

Once the page is retrieved remotely, the reverse path is followed into the kernel. The WebFS daemon packages the reply into another WebFS system call. The vnode op-

eration then copies the page into the virtual memory system, ensuring that subsequent accesses to the page can be handled entirely in the kernel. In this way, concurrent WebFS file accesses are completed as quickly as possible. For maximum concurrency, the WebFS daemon is multi-threaded: multiple threads simultaneously make themselves available for kernel requests (step 0 from Figure 4.1).

In summary, the WebFS architecture provides a number of key advantages. First, interfacing at the vnode layer allows unmodified applications access to the global HTTP namespace. Second, tight integration with the virtual memory system provides for excellent performance on cached accesses. Next, the vnode interface is supported by most UNIX variants and potentially Windows NT, simplifying ports to new platforms. Finally, the user level daemon allows for simplified development: much of the file system functionality and data structures are maintained at the user level, facilitating experimentation and simplifying debugging.

4.2.2 Naming

Upon mounting WebFS, the root directory is defined to contain all HTTP/WebFS sites that the file system is initially aware of. Thus, the root directory is initially empty. Directory entries are created on reference. That is, the first time an application attempts access to an HTTP or WebFS site, the system first checks for the presence of a WebFS server. If a WebFS server is not running on that site then the system checks for the availability of an HTTP server. If either server exists, a directory entry is created in the root WebFS directory. The name of the directory is equal to the hostname of the remote server (note that partial hostnames are first expanded by DNS and that the hostname is

case insensitive). The URL naming convention of appending a colon plus a port number is maintained for naming servers running on non-standard ports. In the future, we plan on leveraging the Active Naming framework described in Chapter 3 to allow users to retrieve files independent of where the file is physically located or replicated.

4.2.3 HTTP Limitations

In cases where WebFS is not running on remote nodes, our system falls back to HTTP to retrieve directory and file contents. This technique has the advantage of allowing read access to the existing HTTP namespace. Providing access to this namespace simplifies the implementation of many existing applications, including Web crawlers. Unfortunately, its very simplicity presents a number of challenges in layering UNIX file semantics on top of HTTP.

On a client request for the contents of a directory containing a pre-determined filename (e.g., index.html), HTTP returns the contents of the file rather than the contents of the directory. In this case, WebFS presents a single entry for the directory with the same predetermined filename. The following techniques are utilized to work around this limitation. WebFS allows creation of directory entries on the fly: on access to a file that WebFS is unaware of, it attempts to retrieve the file from the remote site. If the file does indeed exist, a directory entry is created (failure is returned otherwise). Building on this technique, we also provide a utility program which parses an HTML file (e.g., index.html) creating entries for all links found in the file. Using these techniques, applications can use WebFS to access all files normally available through HTTP.

A second limitation that WebFS must address is the limited file statistics exported

by HTTP when providing directory information. For example, HTTP servers express file size in units of kilobytes or megabytes, making it impossible to ascertain exact file size unless the file is actually retrieved[1]. Upon entering a previously unknown directory, WebFS uses the values retrieved from the HTTP server as the initial file sizes. Upon read access, the file is retrieved from the remote site allowing WebFS to update the actual size of the file. While this may confuse applications which attempt to stat a file before reading it, we have found that in practice standard UNIX applications gracefully deal with such semantics. Thus, we believe the performance savings associated with not retrieving files before access to determine file size is well worth the slightly different semantics. Note that both these limitations are only associated with providing compatibility with HTTP servers; they are eliminated if WebFS is running on the remote site.

4.2.4 Authentication and Security

WebFS uses Public Key Cryptography [Diffie & Hellman 1977, Rivest et al. 1978] to authenticate the identity of individual servers as well as clients. Associated with each file is an access control list (ACL) that enumerates users who have read, write, or execute permission on individual files. Users are uniquely identified by their public keys. Files inherit default permissions from the containing directory. Thus, in the common case, no action is necessary on the user's part when creating a new file or directory. Full details of the WebFS security system are presented in Chapter 6.

[1]Fortunately, reported file sizes are the ceiling of actual file sizes, meaning that applications relying on, for example, the results of a `stat` system call are less likely to fail.

Phase		NFS(s)	WebFS(s)
I	Create	1	1
II	Copy	7	7
III	Stat	3	10
IV	Scan	5	6
V	Compile	15	14
Total		31	38

Table 4.1: WebFS performance relative to NFS on the Modified Andrew Benchmark suite.

4.3 Performance

WebFS has been in day to day use for approximately two years by twenty users at U.C. Berkeley. It has also been installed at the University of Texas at Austin and the University of Washington to conduct wide-area application measurements and experiments, as summarized in Chapter 7. The system is publicly available for download and has been successfully installed at a number of remote sites. The performance of remote file access is highly dependent upon the network connection to the remote site. The cost of the network connection and subsequent transfer dominates the added overhead of making an upcall to the user-level. Once transferred through the network, file pages are cached in the kernel file cache. Thus, once a file has been transferred from a remote site, the performance of cached access through WebFS versus cached access through NFS [Walsh et al. 1985] is virtually identical.

To validate the performance of our system, we ran the modified Andrew Benchmark suite [Ousterhout 1990] for both WebFS and NFS [Walsh et al. 1985]. The NFS measurements were taken on a Sun Ultra Server making requests to an Auspex NFS server over Ethernet. For WebFS, a Sun Ultra Server made requests to a WebFS server running

on a second Ultra Server also over Ethernet. The performance results are summarized in Table 4.1. Given the largely untuned nature of the code and the extra overhead associated with making upcalls to the user level, the performance of WebFS is reasonable for most phases of the benchmark. Significant slowdown is apparent in phase III, which executes a stat operation on 70 files. The factor of 3 slowdown is caused by the upcall that is made to the user level for every operation. One approach to addressing this performance issue is to store these values in the kernel when the file is initially copied.

4.4 Cache Coherence Policies

For wide-area distributed applications, the choice of caching policy is crucial for application correctness, performance, and ease of development. We expect to implement application-specific caching policies based on our experience with the development of various distributed applications. In the future, we hope to allow for application programmers to develop their own coherence policies if none of the provided techniques are appropriate (such a technique has been explored in other contexts [Cao et al. 1994, Cao et al. 1995]). We plan to allow programs to select caching policies on a per file descriptor basis allowing different applications to access the same file with different coherence policies. In this section, we will describe three contexts for distributed computation and file sharing. We describe a cache coherence policy appropriate for each application.

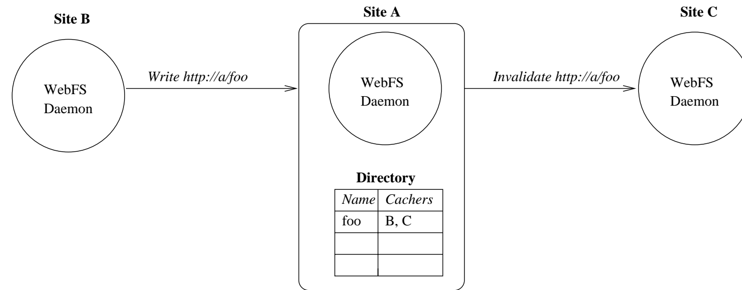


Figure 4.2: Implementation of the last writer wins cache coherence policy.

4.4.1 General File Sharing: Last Writer Wins

For general file sharing in such cases as distributed program development, WebFS provides last writer wins cache coherence similar to what was originally developed for AFS [Howard et al. 1988]. Figure 4.2 describes the WebFS implementation of last writer wins. Each WebFS daemon maintains a directory of files exported from their site. For each file, a list of all sites caching the file is maintained. In this example, site B commits a write (corresponding to either a close or flush of a modified file) of a file *foo* hosted at site A. The WebFS daemon consults its directory and discovers that sites B and C are caching *foo*. An invalidation message is sent to site C, which is responsible for invalidating any cached kernel pages for file *foo* through a WebFS system call. As an optimization, the invalidate for site B is omitted since its cache value is current. Finally, the directory is cleared of all cachers which were sent invalidate messages (in this case, only site C).

The last writer wins coherence policy is appropriate in the case where a file is being widely shared for read access and only a single site is likely to be modifying a file at any given time. The callbacks provided for cache invalidates make this policy scalable to

hundreds of simultaneous read cachers [Howard et al. 1988]. In the common case, clients access cached files without contacting the server and have reasonable guarantees of accessing the most recent version of the file. Note, however, that last writer wins is unlikely to scale well to the context of providing cache coherence for generalized Web browsing. Maintaining directory information for potentially millions of simultaneous file cachers (many of which will access the file exactly once) would impose severe storage and performance bottlenecks.

While last writer wins provides coherence for relatively small-scale file sharing, it does not preserve strict UNIX write semantics in the following case. If two sites simultaneously open a file for writing, then the updates of the last site to flush changes will overwrite any earlier modifications. Since last writer wins is most appropriate for cases where write sharing is the uncommon case, we believe that the advantages of simple, scalable implementation and coherence guarantees outweigh any differences from strict UNIX semantics. In practice, it has been found that simultaneous write sharing is fairly rare for UNIX workloads [Spasojevic & Satyanarayanan 1994], though such sharing is likely to be much more common in other types of workloads, such as groupware applications.

4.4.2 Internet Chat: Append Only

The second cache coherence policy implemented in WebFS was driven by the implementation of an Internet chat application [Yoshikawa et al. 1997]. For this application, we required scalability to thousands of geographically distributed users. For chat, we determined that updates need not appear in the same order at all sites (as is currently the case with popular Internet chat services such as Internet Relay Chat). Of course, all updates should still be propogated within a reasonable time bound, e.g., a few seconds. The append

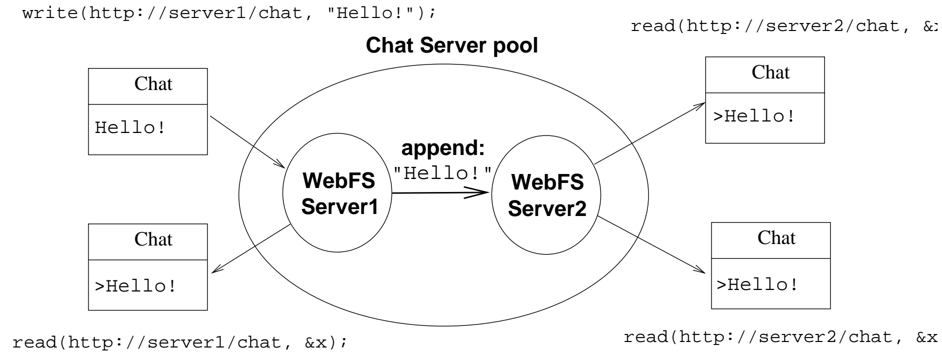


Figure 4.3: Chat rooms as WebFS files.

only coherence policy satisfies the needs of the chat application as summarized in Figure 4.3. A WebFS file is associated with each chat room. However, multiple WebFS servers, called a server group, can be contacted for sending and retrieving updates to the chat log. A read operation on the chat file corresponds to a poll for availability of new messages, while a write corresponds to appending a new message to the chat file. The client application (through Active Names described in Chapter 3) is responsible for choosing the closest server in the group. WebFS servers package multiple file updates and synchronize their updates with the rest of the server group periodically. This policy ensures that clients connected to the same server will all see updates in the same order. However, clients connected to different server groups can potentially see updates out of order. With a synchronization period of 300 ms, we found that out of order updates were not a perceptible semantic problem.

The file system interface is well-matched to chat semantics in a number of ways: (i) file appends and reads abstract away the need to send messages, (ii) the chat file provides a persistent log of chat activity, and (iii) access control lists allow for private and secure (through the security mechanisms described in Chapter 6) chat rooms. To quantify the

benefits available from the WebFS framework, we implemented two versions of chat with identical semantics, both with and without WebFS. The initial implementation consisted of 1200 lines of Java code in the client and 4200 lines of C++ code in the server. By using WebFS to handle message transmission, failure detection, and storage, the size of the chat client code was reduced to 850 lines, while the WebFS interface entirely replaced the 4200 lines of chat server code. The main reason for this savings in complexity was the replacement of separate code for managing communication and persistent storage of chat room contents with a single globally accessible and consistent file.

As an added benefit, this common WebFS interface is also available for other distributed applications. For example, a shared distributed whiteboard application can also be implemented using this interface. Further, we believe distributed applications which are not necessarily bound by connection rate to the server, but rather suffer from weak connectivity (i.e., mobile clients), can benefit from the use of WebFS server groups to synchronize only when network connectivity becomes available. For example, a meeting room scheduler and a distributed bibliography server were implemented in the Bayou project [Terry et al. 1995] using a similar notion of server groups.

4.4.3 Stock Ticker: Multicast Updates

The final application developed in the WebFS framework supports an emerging class of new Internet services, including whiteboards, video conferencing, interactive games, and news services. These applications have two interesting properties. First, all clients of the system are interested in all updates, and second, a significant amount of overlap is present among the updates. Unfortunately, the current practice of individually sending

updates to each client is wasteful of Internet resources. A better solution is to use IP Multicast [Deering 1991] to transmit updates for applications which in essence desire to periodically broadcast updates. Using router support, multicast replicates packets only along the necessary routes. For example, if a message is destined for multiple clients on a single subnet across the continent, a single packet is transmitted across the continent and is replicated only at the destination subnet. This approach contrasts with the traditional approach of unicasting individual updates to each of the clients, wasting bandwidth and increasing average latency.

We added multicast support to WebFS to experiment with various caching policies for broadcast applications. Currently, WebFS associates two multicast addresses with each file designated to use multicast for coherency. The first address, called channel A, is used to send invalidates of the file contents to interested clients. The second address, channel B, is used to send out updates of the file to all clients subscribing to the address. Channel A is used by clients which need to ensure that stale cache entries are not used, while channel B is used by clients needing the most current file contents (such as subscribers to broadcast services). The use of these multicast channels also simplifies server design and fault recovery. Since interested clients subscribe to either the invalidate or update channel, the server need not maintain a directory of file cachers as was necessary for the last writer wins protocol. On a server crash, for example, the client directory does not need to be reconstructed as any updates or invalidates are simply sent to the appropriate multicast address.

To explore the utility of multicast support in WebFS, we implemented a stock ticker which graphs stock trading price over the course of the day. The WebFS stock ticker is a

Java [Gosling & McGilton 1995] applet which uses the WebFS protocol to receive periodic updates of stock charts. A centralized daemon continually retrieves current stock prices from a stock quote service, updates the graphs for the day, and uses WebFS to multicast a bitmap representation of a new stock chart every 10 seconds. The use of a Java interface to WebFS not only makes the code portable across multiple platforms but also makes it unnecessary to install WebFS on the client machine for this application. Unfortunately, many popular Web browsers do not yet contain multicast support. Thus, the applet must currently be run through the appletviewer which is part of the Java Development Kit (JDK) provided by JavaSoft. We hope that future versions of browsers will include native multicast support. An alternative is to develop a browser plug-in supporting Java applets that require multicast support.

Multicast support can also be useful in the context of Web browsing and wide-scale shared access in general. For example, users can select coherency for sites where they desire to see the most current page contents on every visit (such as news services). In the background, the browser can listen on the invalidate channel; on the first invalidate, the page is flushed from the browser's cache. At this point, the browser can cease listening on the channel as it is not interested in further invalidates. On the next client access to the page, the browser misses in its cache and retrieves the current page from the server. This argument can be extended to cache proxies in a straightforward manner.

A number of concerns are associated with such widespread deployment of multicast. First, the number of IP addresses reserved for multicast is limited to about one million in the current version of IP. However, the impending release of IPv6 [Deering & Hinden

1996] will increase this number to 2^{112}. Further, the number of addresses currently available more than suffice for any initial deployment of multicast for Web updates. A second concern is the space required in routing tables for each multicast address. If multicast use becomes very popular, the amount of memory required for routing tables of heavily used routers in the Internet backbone could become prohibitive. We believe that once multicast achieves such popularity, it is likely that compression techniques can be developed to ensure that the size of routing tables remains manageable. Next, multicast is not currently ubiquitously deployed. However, all major IP router manufacturers have implemented or are implementing multicast support in their products and all major OS vendors have added multicast support to their kernels. Finally, reliability is a concern in multicast since it is layered on top of UDP making duplicate and lost packets fairly common. We plan to leverage techniques from Scalable Reliable Multicast (SRM) [Floyd et al. 1995] for applications requiring guarantees with respect to lost or duplicated messages.

4.5 Related Work

A number of previous efforts including AFS [Howard et al. 1988], Alex [Cate 1992], and UFO [Alexandrov et al. 1997] have produced file systems which export a global namespace. One of the first distributed file systems to provide access to a global namespace across the wide area was AFS. The original goal of AFS was to improve the scalability of local-area file servers. WebFS uses the notion of server-callbacks to implement its last writer wins cache coherency. While AFS possesses many strengths, a number of system limitations prevents AFS from supporting distributed applications. AFS source code is not

freely available and thus does not facilitate experimentation. Further, AFS assumes the traditional UNIX file access model where widespread read/write sharing is the uncommon case. For many distributed applications, this assumption is invalid. We hope to use WebFS to determine cache coherence policies appropriate for a number of different distributed applications. Finally, AFS does not currently support the HTTP namespace which can make it less attractive for many applications.

UFO provides access to a read-only HTTP namespace, while providing read/write access to remote files using the FTP protocol. UFO has the advantage of being implemented entirely at the user-level using the Solaris proc file system to intercept file system relevant system calls. Thus, installation of UFO is straightforward and does not require root privileges. However, use of the proc file system limits performance. Further, in contrast to WebFS, UFO is only able to make very loose consistency guarantees in the case of FTP, and no guarantees for HTTP. Finally, accounts are required to access files on remote machines through FTP. We believe the requirement of providing a full user account to individuals hoping to access remote resources may limit widespread deployment of the system.

Similar to UFO, Alex is a user-level NFS server providing read only access to global anonymous FTP servers. The use of an NFS server implies improved performance of Alex relative to UFO. However, once again, Alex can only provide limited cache coherence guarantees. Further, the read-only nature of the file system makes Alex inappropriate for many distributed applications.

Recently, research on WebNFS [Microsystems 1996] has proposed replacing HTTP with NFS as the transport layer for the Web. In some cases, an order of magnitude speedup

over HTTP is claimed because of the great amount of tuning that has gone into NFS. WebFS is largely independent of the transport protocol underneath. If in the future WebNFS gains popularity, it should be relatively straightforward to support the new protocols in WebFS.

A number of research efforts, including Coda [Mummert et al. 1995] and Bayou [Terry et al. 1995] have addressed file system and application semantics in the context of disconnected operation (mobile machines connected to the network intermittently or over slow links). We plan to leverage the techniques utilized for disconnected operation to make WebFS resilient to failures (which we believe will likely be the common case in the scale of the Internet). It is interesting to note that many of the challenges of mobile computing can also apply in the context of the Web.

Projects such as Legion [Grimshaw et al. 1995], Globus [Foster & Kesselman 1996], and Atlas [Baldeschwieler et al. 1996] aim to provide an infrastructure for global computation (i.e., a world wide supercomputer). WebFS is complimentary to such projects in the sense that it can provide the global namespace, authentication, and cache coherence necessary to execute such global applications. We plan to explore cache coherence policies appropriate for different classes of parallel applications. For example, one can imagine a distributed shared memory (DSM) abstraction built on top of a memory-mapped WebFS file.

4.6 Summary

WebFS is a global file system providing read/write access to the Web namespace to unmodified UNIX applications. WebFS is compatible with HTTP and provides access to

the Web namespace for unmodified applications. By interfacing at the vnode layer, WebFS provides cached access performance comparable to existing file systems. Using multicast technology, WebFS is able to efficiently distribute updates of widely shared Internet data. To demonstrate the functionality of our system, we have developed two distributed applications, Internet chat and a stock ticker. Our initial results indicate that a global cache coherent file system is a useful substrate and abstraction for implementation of scalable Internet services.

Further, WebFS successfully meets the goals for a wide-area computation system, as presented in Chapter 2. WebFS provides a *uniform interface to remote resources*, allowing users to access wide-area content through the same convenient and familiar file system interface independent of where the files are physically located or replicated. The file system allows for *flexibility* in the specification of cache coherence protocols. Such flexibility is necessary to allow applications to match coherence algorithms to their particular performance and correctness requirements. By caching and replicating content across the wide area, WebFS maintains a high level of *performance*, with access to data at speeds comparable to local file access in the case where locality in access patterns is present. Finally, WebFS is *backward compatible* with existing applications because accesses to global files are intercepted in the kernel, allowing for maximum transparency and performance.

Chapter 5

Transparent Result Caching

The previous chapters described Active Names, a programmable interface for locating wide-area resources, and WebFS, a wide-area file system providing persistent storage for distributed applications. This chapter describes Transparent Result Caching (TREC), a technique for profiling program execution to determine file input/output dependencies. By leveraging WebFS, one use of TREC is to cache dynamically generated Web content at both servers and proxies. Further, as discussed in this chapter, the TREC framework enables a number of interesting applications outside the context of the principal focus of this dissertation, remotely programmable resources.

5.1 Overview

Today, many distributed services perform one of two types of operations to satisfy user requests: i) the contents of a file are read and transmitted to the client, or ii) a program is locally executed and the results of the program are transmitted to the client.

Chapter 4 demonstrated how a wide-area persistent storage system with flexible coherence mechanisms simplifies the development and improves the performance of applications that align with the first class of operations above. However, as described, it cannot improve performance of applications that generate their data by executing a program as opposed to using a pre-existing file.

Dynamically generating content is becoming increasingly popular in the Internet (as first described in Section 3.3.2). For example, a program may run to generate data, such as stock quotes or a database query to an Internet search engine. The results of such programs cannot typically be cached at servers, at Web proxies, or at clients, because the results of the program can change quickly. The inability to cache such programs adversely affects system performance and consumed wide-area bandwidth (all requests must go across the wide area to a central server). This chapter describes a set of techniques to allow file system caching of a certain class of dynamically generated data.

In designing our solution, we observed that many dynamically generated Web pages are valid ("cacheable") as long as all of the state used to generate the page is unchanged. For example, the results of a consumer query pricing a computer remains valid until the underlying price/part list is updated. Thus, if we can determine and track the dependencies of an object (e.g., the file representing a Web page), we can cache that object pending violation of any of its dependencies.

In this context, our goal is to develop a general framework for transparently managing the interactions and dependencies among input files, development tools, and output files. By unobtrusively monitoring the execution of unmodified programs, we are able to

track *process lineage*—each process's parent, children, input files, and output files, and *file dependency*—for each file, the sequence of operations and the set of input files used to create the file. This information can be used to determine the exact sequence of operations used to create any system file or to keep the contents of output files synchronized as dependent input files are modified. In this chapter, we describe the implementation of such a framework for *Transparent Result Caching* (TREC).

The combination of transparently obtaining process lineage and file dependency information provides a powerful substrate for developing applications in a wide range of application domains. Our primary motivation in designing this framework is supporting caching of dynamically generated wide-area content. However, as the applications below demonstrate, the framework is also useful beyond the context of virtual services.

- *Unmake:* Unmake allows users to query the system for file lineage information, including the full sequence of processes executed to create a particular output file. For example, users running simulations who neglect to document the specific parameters used to generate output files can query Unmake for the program used to create the output, the specific command line parameters, and the environment variables in effect when the command was executed.

- *Transparent Make:* This version of the `make` utility allows users to specify the sequence of operations for constructing output files as simple shell scripts. The first time the shell script is run, TREC determines the set of files that affect output files through empirical observation. During subsequent executions of the shell script, TREC can re-run only those commands that have been invalidated by changes to input files. This

approach has two advantages. First, it frees users from manually specifying dependency information in a language that can be restrictive [Levin & McJones 1993]. Next, transparent make does not require users to manually update dependency information.

- *Dynamic Web Object Caching:* Today many Web pages are constructed dynamically as a result of user input. One example in the Web today is using CGI-bin programs to produce HTML pages. For instance, to download the latest version of Netscape's Navigator, users answer a number of questions about their platform before being presented with a page enumerating URLs for the correct binary. The disadvantage of using CGI-bin programs are: (i) servers, proxies, and clients are normally unable to cache the page because the contents might change for every invocation of the program, and (ii) retrieving such content incurs extra overhead at servers because a program must be run to generate the content (as opposed to transmitting an existing file). Given some locality in user input, it would be cheaper to cache the results of CGI-bin program execution with popular input patterns (e.g., users wishing to download the latest English version of Navigator for Windows95/NT), reducing both server load and client latency. While existing work allows for applications to be written that can cache their results [Iyenger & Challenger 1997], TREC automates this process by automatically caching program results and invaliding such results when the input to a CGI program changes (e.g., a new version of Navigator becomes available). This application is more active than the previous two examples: TREC dependency information is used to generate specific actions whenever a pre-specified operation takes place (an output file is invalidated when its input files are modified).

The rest of this chapter is organized as follows. The implementation of TREC, its baseline performance, and limitations of TREC profiling are described in detail in Section 5.2. In Section 5.3, we describe how TREC is used to implement three of the sample applications described above: unmake, transparent make, and dynamic Web object caching. Section 5.4 describes work specifically related to TREC, and a summary of the chapter is presented in Section 5.5.

5.2 System Design and Implementation

5.2.1 Architecture

TREC captures file dependency information by intercepting a small number of system calls using native kernel tracing mechanisms. Using the information from these calls, TREC maintains the following information for each process: command line arguments, environment variables, process parent, process children, files read (input files), and files written (output files). TREC then organizes this information hierarchically, allowing users to query the system for file lineage, for example, to determine all processes involved in creating an output file. This section describes this system architecture in more detail.

TREC is implemented using the Solaris `proc` file system (most major UNIX variants provide a substantially similar interface) to intercept the set of system calls necessary to build dependency information. For our target UNIX architecture this set includes the following system calls: `open`, `fork`, `fork1`, `creat`, `unlink`, `exec`, `execve`, `rename` and `exit`. By catching these system calls, TREC is able to determine full process lineage information. Command line parameters and environment variables are available from the `exec` system

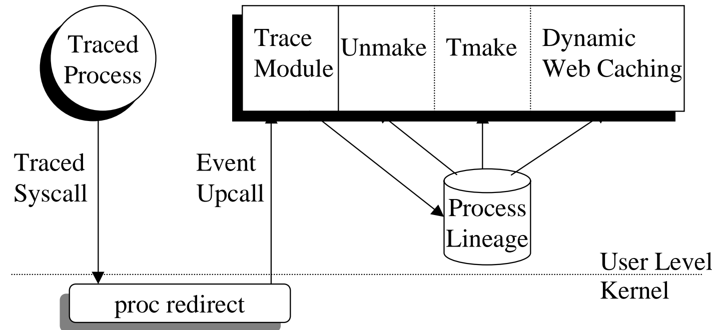


Figure 5.1: TREC architecture overview.

calls. TREC determines the set of input files and output files by examining the options to the `open` system call (targets of `creat` are assumed to be output files). This design choice potentially over-constrains the set of input and output files because, for example, a file opened for writing that is not actually written to with a `write` system call is considered an output file. We made this design decision because of the added overhead of intercepting all `read` and `write` system calls in addition to the set of calls already being monitored. For some programs, we observed that `read` and `write` operations were executed four times as often as all other traced system calls combined. Note that earlier file system tracing studies also inferred read and write operations to avoid the extra overhead of tracing the read/write system calls [Baker et al. 1991].

The overall TREC system architecture is summarized in Figure 5.1. TREC runs as a multi-threaded process that attaches to a target process using the `proc` file system interface. A trace thread is responsible for building lineage information. Other threads use this lineage information to implement the higher-level services described in Section 5.3. Given a list of target system calls, `proc` forces a context switch to the TREC tracing thread

Operation	**Baseline**	**Traced**	**Syscall Rate**	**Added Overhead**
`open` syscall	12.4 s	19.3 s	403 calls/s	54.8%
Compile	128.4 s	146.2 s	160 calls/s	13.9%
Latex	35.1 s	36.3 s	16 calls/s	3.5%

Table 5.1: TREC profiling overhead.

whenever a relevant call is executed by the attached process or any of its children. The tracing module exports a callback-based interface to application modules. Modules, such as transparent make, use the callbacks to determine when output files are modified. Thus, when a callback is received that a file is modified, a module can take action on dependent output files—e.g., invalidating the output or re-generating it. Currently, callbacks are exported for file read and write events.

TREC uses the system call information to build a lineage tree of the target process and all of its children. Each node of the lineage tree represents an executed process and contains the following information: execution time, command line arguments, environment variables, files read, and files written (a file that is both read and written by a program is considered to be only an output file). An example of this lineage information is presented in Section 5.3.1.

5.2.2 Performance

Since the context switches imposed by the `proc` file system required to perform our tracing can impose significant overhead, we took a number of measurements to quantify the slowdown. Table 5.1 quantifies the TREC overhead for three simple benchmarks. All benchmarks were conducted on a 167 Mhz Sun Ultra/1 workstation running Solaris 2.5.1.

The first benchmark, open, calls `open` and `close` on the same file in a tight loop 5000 times (note that only the open call is actually traced). While the 54.8% overhead imposed by TREC is significant, the next two benchmarks demonstrate that the slowdown of individual system calls do not adversely affect the performance of real applications. The next benchmark is a compilation of the Apache HTTP server, version 1.2.4 [Apa 1995]. The source tree consists of 38,000 lines of C code and was compiled over NFS. While the 13.9% slowdown is noticeable, we believe it to be tolerable. The final benchmark, Latex, involved running `latex` four times, `bibtex`, and finally `dvips` to produce postscript for a 17 page document. For this benchmark, only a 3.5% overhead is introduced. As indicated by the "Syscall Rate" column in Table 5.1, the measured slowdown directly corresponds to the rate at which processes execute traced system calls. Since the Latex benchmark executes only 16 traced system calls per second, it suffers the smallest slowdown.

To address the overhead imposed by the `proc` and related tracing facilities, we could implement TREC functionality in the kernel. Various tools such as Watchdogs [Bershad & Pinkerton 1988], Interposition Agents [Jones 1993], or SLIC [Ghormley et al. 1998b] can be used to trace system call activity with little or no overhead. However, such tools often require root access to install, can be difficult to use without kernel source, and can also be difficult to distribute since kernel copyright restrictions may prevent distribution of source code. We opted for the user-level approach for portability, and ease of distribution and installation. If performance becomes an issue, we believe switching to a kernel implementation will be straightforward.

5.2.3 Limitations

As motivated earlier, a number of applications are able to benefit from TREC functionality. However, TREC can produce incorrect results for applications that base their results on non-deterministic or difficult-to-trace input. The following are some behaviors necessary for the proper function of TREC applications:

- Programs must be deterministic and repeatable. Each of the programs that contribute to the creation of a file must behave the same way each time it is run, assuming that its own inputs have not changed. A compiler invoked with a given set of options will generally produce the same object file, as long as the source code has not changed. A simple example of a program that violates this restriction is UNIX `date`, whose output is virtually never the same twice. Any file that relies on the output of `date` cannot be guaranteed to be up-to-date, nor can it be reliably re-created. As discussed in Section 5.2.3, TREC does not currently automatically determine the class of applications that produce non-deterministic output.

- Programs cannot rely on user input. Related to the requirement for determinism, programs that rely on user input or GUI operations are not automatically re-creatable. For example, if an output file incorporates user-input to a text editor, TREC cannot accurately model program dependencies.

- Interaction with environment variables must be reproducible. TREC stores all environment variables in effect when a program runs. On subsequent runs of the same program, TREC can check to see if the currently set environment variables match

the environment variables in effect when the program was originally run. If the variables do not match, the program should be re-run (as opposed to using the cached results). TREC assumes that a program's interaction with environment variables is predictable: all other things being equal, a program run multiple times with the same set of environment variables should produce the same results.

- File contents must be static, as long as the modification time has not changed. While seemingly a trivial constraint, certain special files do not follow this convention. Virtually all of the files in `/dev` violate this restriction. For example, each time the tape drive `/dev/rmt0` is read, it appears to have different data, for example, because an operator has exchanged one tape for another.

- File contents must be changed locally. For example, an NFS mounted file might be modified at any of a number of machines, not all of which may be traced by TREC. Applications requiring callbacks on file modification rely on TREC's ability to intercept all file updates.

- In general, TREC cannot profile programs that rely on network communication to produce their output[1]. For example, an application that communicates with a remote server may receive different results for each run of the program.

- Programs must run to completion while being profiled. For example, a program that terminates prematurely because it received a signal may not have generated complete dependency information, leaving TREC in an inconsistent or incorrect state.

[1] Note that techniques from the fault tolerance community could potentially address this limitation [Alvisi & Marzullo 1996].

Despite the limitations outlined above, we will demonstrate that TREC remains a useful tool in a number of different contexts. Our current approach to handling output files produced by programs that fall into the above categories is to allow users to specify a set of program names whose output cannot be cached or re-created. TREC uses a configuration file containing a list of programs whose output is potentially uncacheable. If any of these programs are involved in producing an output file, TREC caching is disabled (e.g., by dynamic object caching).

5.3 Applications

In this section, we describe three utilities built on top of the TREC tracing module: unmake, transparent make, and dynamic Web object caching. The functionality to query the system for dependency information provided by unmake is fundamental to the operation of dynamic object caching. The transparent make application is presented as an example of the general utility of the TREC framework.

5.3.1 Unmake

The TREC tracing module builds a process lineage tree as described in the previous section. To demonstrate the use and utility of this information, we describe a simple TREC service, unmake, that allows for a number of simple queries. For example, users might request information about all processes, and their parents, that read a particular file.

As a concrete example, consider the shell script used to compile a simple program in Figure 5.2(a). When this script is run in a shell traced by TREC, the tracing module

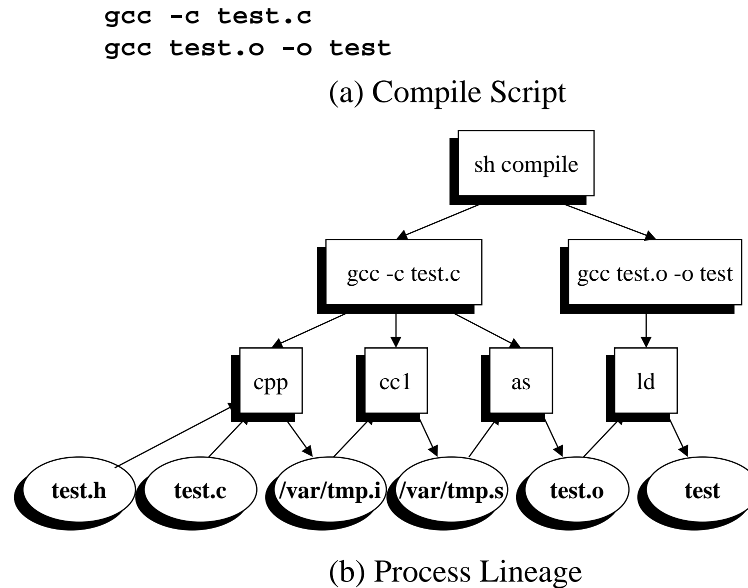


Figure 5.2: Compilation script and corresponding process lineage tree.

automatically builds lineage information for the processes executed by the script. The complete process lineage tree for producing the `test` executable is presented in Figure 5.2(b). Arrows between rectangles indicate process parent information. For the leaf processes, we indicate the files (indicated as ovals) read and written by each process. Notice that while the compilation script makes no reference to a header file (`test.h`) or to the assembler and loader programs, TREC is able to build full lineage information by observing the execution of all processes and sub-processes spawned by the compilation (e.g., TREC is able to transparently deduce a dependency to `test.h` by observing the fact that cpp read `test.h` for input).

The unmake module can be interactively queried for process lineage. Figure 5.3 shows partial output of a run for a query requesting lineage information for processes reading

Query: read test.c

```
Parent ID: 28426 Program ID: 28428
Argv:   /usr/local/lib/gcc-lib/sparc-sun-solaris2.5/2.7.2.f.1/cpp
        -lang-c -undef -D__GNUC__=2 -D__GNUC_MINOR__=7 -Dsun -Dsparc
        -Dunix -D__svr4__ -D__SVR4 -D__GCC_NEW_VARARGS__ -D__sun__
        -D__sparc__ -D__unix__ -D__svr4__ -D__SVR4 -D__GCC_NEW_VARARGS__
        -D__sun -D__sparc -D__unix -Asystem(unix) -Asystem(svr4)
        -Acpu(sparc) -Amachine(sparc) test.c /var/tmp/cca006u2.i
Envp:   COLLECT_GCC=gcc HOME=/homes/rivers/vahdat HOST=tolt HOSTNAME=tolt
        HOSTTYPE=sun4 LOGNAME=vahdat MACHTYPE=sparc OSTYPE=solaris
Children: (none)
Input:  test.c test.h /usr/include/sys/feature_tests.h /usr/lib/libc.so.1
        /usr/local/lib/gcc-lib/sparc-sun-solaris2.5/2.7.2.f.1/include/stdio.h
        /usr/lib/libdl.so.1 /usr/platform/SUNW,Ultra-1/lib/libc_psr.so.1
Output: /var/tmp/cca006u2.i
========
Parent ID: 28424 Program ID: 28426
Argv:   gcc -c test.c
Envp:   HOME=/homes/rivers/vahdat HOST=tolt HOSTNAME=tolt HOSTTYPE=sun4
        LOGNAME=vahdat MACHTYPE=sparc OSTYPE=solaris
Children: 28428 28430 28432
Input:  /usr/lib/libc.so.1 /usr/lib/libdl.so.1
        /usr/local/lib/gcc-lib/sparc-sun-solaris2.5/2.7.2.f.1/specs
        /usr/platform/SUNW,Ultra-1/lib/libc_psr.so.1
Output: (none)
```

Figure 5.3: Sample TREC lineage query results.

`test.c`. Unmake searches the lineage information for records where the set of input files contains `test.c`, in this case returning execution of the C pre-processor, `cpp`. Unmake returns the following information about the execution environment of all traced processes. All command line arguments, in addition to the environment variables, are listed (note that for brevity, the list of environment variables is truncated). All of the program's input and output files are listed along with a unique program ID (currently the UNIX pid) and the ID of the process's parent. The children field specifies all spawned processes (no processes

were forked in the case of `cpp`). The query also recursively provides information on the process's parent, `gcc` in this case. While omitted from Figure 5.3, information on `/bin/sh`, the process which executed the compilation shell script, and `/bin/tcsh`, the root process of the TREC trace, is also returned.

While not currently implemented, unmake combined with a source control system or, more generally, a file system capable of transparently producing older file versions [Heydon et al. 1997] can be used to rollback to earlier versions of output files. For example, users debugging a program executable may use an interactive process lineage visualization tool, similar to the display in Figure 5.2(b), to identify input files that may have potentially introduced bugs. The user could then roll back to an earlier version of the suspect input file (using an appropriate file system or a source control system), instructing unmake to rebuild the output file with the new set of input files.

5.3.2 Transparent Make

The process and file lineage information produced by TREC can be used to build a more dynamic version of the traditional `make` utility, called transparent make (tmake). With traditional `make`, users are forced to learn a new language for specifying dependencies between input, intermediate, and output files, a process that can become cumbersome and error-prone for large development efforts. In contrast, tmake allows users to describe the process for creating output files more naturally; shell scripts (Figure 5.2(a) provides a simple example) describe the sequence of steps used to create an output file.

Thus, rather than forcing users to manually specify dependencies through Makefiles, TREC is able to dynamically determine dependencies by observing the execution of

shell scripts. This approach has the following advantages: (i) eliminating user errors that may occur in specifying dependency information, (ii) dynamically updating dependency information as it changes, and (iii) eliminating the need to learn the Makefile specification language, which can be sometimes be complicated, restrictive, and/or error-prone. Of course, some users may be unable to create a shell script for the compilation of their program because they are accustomed to leveraging existing Makefile templates for compilation of their programs. In this case, tmake can bootstrap its execution by profiling the initial execution of a `make` command.

Tmake provides an interface similar to traditional `make`. A shell script carries out tasks such as compilation while TREC builds process lineage information. The user explicitly requests re-synchronization of the target of the shell script (i.e., the executable) by re-executing the shell script. For each command in the shell script, tmake looks up the set of input files for the command and checks the last modified time of each input file. If any of the files have been modified since the last execution time of the command, the process is re-executed. Otherwise, the command is skipped. Also consider the case where the compile shell script is modified—for example, to add a new `-O` parameter to the compiler. In this case, tmake will re-execute any programs with modified command line parameters even if file dependencies have not changed because tmake will be unable to match the new process and command line parameters with any entries in its process lineage hierarchy. Thus, tmake functions similarly to `make`, while maintaining the advantages of implicitly determining dependency information and dynamically updating dependencies as they change (without requiring user intervention).

5.3.3 Dynamic Web Caching

In this subsection, we describe a third TREC example, dynamic Web caching. This service is quite different in motivation and implementation from both the previous services, unmake and tmake. We begin by motivating the need for dynamic Web caching and go on to describe how we modified an HTTP server to interact with TREC in order to provide this service.

Motivation

To alleviate congestion across Internet links, several researchers have proposed a number of caching schemes both to reduce the load on Internet backbones and to improve user response times [Gwertzman & Seltzer 1996, Chankhunthod et al. 1996, Zhang et al. 1997]. One early study [Danzig et al. 1993] found that strategically-placed caches could reduce FTP file traffic by as much as 50%. Similar studies of Web traffic yielded similar results [Braun & Claffy 1994, Duska et al. 1997, Gribble & Brewer 1997].

As already described in Chapter 3, any caching scheme will be limited, among other considerations, by the fraction of Web pages that are dynamically generated, and hence classified as uncacheable. For instance, a CGI-bin program might be run to produce HTML in response to a user query (e.g., what are the show times at a movie theater) or to embed a different advertisement in the same logical page based on the identity of the requester. Approximately 20% of the queries to IBM's Web server for the 1996 Olympic games resulted in the dynamic generation of HTML (e.g., to get current medal standings) [Iyenger & Challenger 1997]. In general, the contents of such pages cannot be cached because the

result of the program can change from execution to execution. Caching dynamic objects is important for overall performance for the following reasons: (i) An increasing percentage of Web objects are being dynamically generated, (ii) a program must be run on the server side to generate such objects, increasing server load, and (iii) client latency is generally limited by the minimum time required to run the program.

Chapter 3 proposed one technique for caching dynamically generated content. Here, we present a second, complimentary approach for reducing the overhead of busy Web servers: caching dynamically generated objects as normal files and using TREC to manage invalidations. Limitations to the type of dynamic objects that can be cached—for example, those that access a database—are described below. In this scheme, cache objects are stored in the file system under the name of the program used to generate them concatenated with any arguments to the program. Thus, a request for the object `http://www/cgi-bin/query?argument` might be cached locally in, for example, a file `/usr/local/apache/cache/cgi-bin/query?argument`. Subsequent accesses to the same CGI program with the same argument list can be returned from the disk cache, eliminating the need for `fork` and `exec` operations, and saving any computation time associated with the requested program. For example, consider user queries to a Web site providing movie show times. Caching is attractive in this context because locality is likely present in the access pattern (popular movie at popular theater) and because the query results remain valid for an extended period of time (e.g., one week).

Of course, one problem with caching dynamic objects is maintaining cache consistency. The dynamically generated Web objects often depend on a set of input files. For

example, a consumer Web site might provide an interface for users to interactively query for the latest pricing and availability information. Dynamic object caching can reduce server load by caching the replies to frequently made requests. However, all cached copies must be invalidated when pricing or availability information changes. As another example, a news site may dynamically generate a "front page" containing headlines and synopsis of news stories. Caching is also useful in this context since the same object will be delivered to all users for a certain time period. Once again, however, cached copies must be invalidated when the list of available stories is updated.

To address this need for invalidation, we use TREC to profile the execution of programs creating dynamic Web objects. When TREC detects that an input file contributing to the creation of a cached object has been modified, one of two courses of action can be followed: (i) the file containing the cached copy of the Web object is removed, forcing the Web server to re-create it on the next user access, or (ii) the program which originally created the cached object can be re-executed to bring the cache up to date. Determining which approach is taken depends on the popularity of the object in question, the current load of the Web server, and the cost of recomputing the object.

Implementation

To investigate the utility of dynamic Web caching as described above, we modified Apache's HTTP server (version 1.2.4) in the following way. When a CGI object is requested, the server first checks for a file whose name matches the CGI object name concatenated with any arguments. If the file exists, its contents are returned without spawning a new process to carry out the request. If not present, a process is spawned to produce the desired

results. The program's output is written to a file in parallel with the response to the requester. Currently, CGI arguments are assumed to be transmitted on the command line, corresponding to an HTTP GET request. In the future, it should be straightforward to modify Apache to include arguments transmitted in the HTTP header, corresponding to HTTP POST requests. Thus, subsequent requests are able to use a cached copy of the CGI object. File locking is used to ensure that partially generated results are not returned to users. We were able to make these changes by modifying approximately 50 lines of C code from the Apache distribution.

To allow for invalidations, the execution of CGI programs is profiled by TREC. Similar to transparent make, the dynamic Web caching module registers callbacks for all files that act as input for CGI programs, requesting notification when any of the target set of files are modified. When such a callback is received, all CGI objects (cached output files) which depended on the modified file are removed, forcing the server to regenerate the result on the next user access. Since this level of dependency checking cannot guarantee consistency for all CGI objects (further discussed in Section 5.3.3), we allow the server administrator to specify a set of CGI programs that cannot be cached through an Apache configuration file.

Performance

To quantify the baseline performance benefits of caching CGI program results, we measured the performance of retrieving CGI results for both our modified Apache server and the original, unmodified version. The measurements were taken as follows. Eight Sun Ultra/1 workstations running Solaris 2.5.1 connected by a 10 Mb/s Ethernet switch

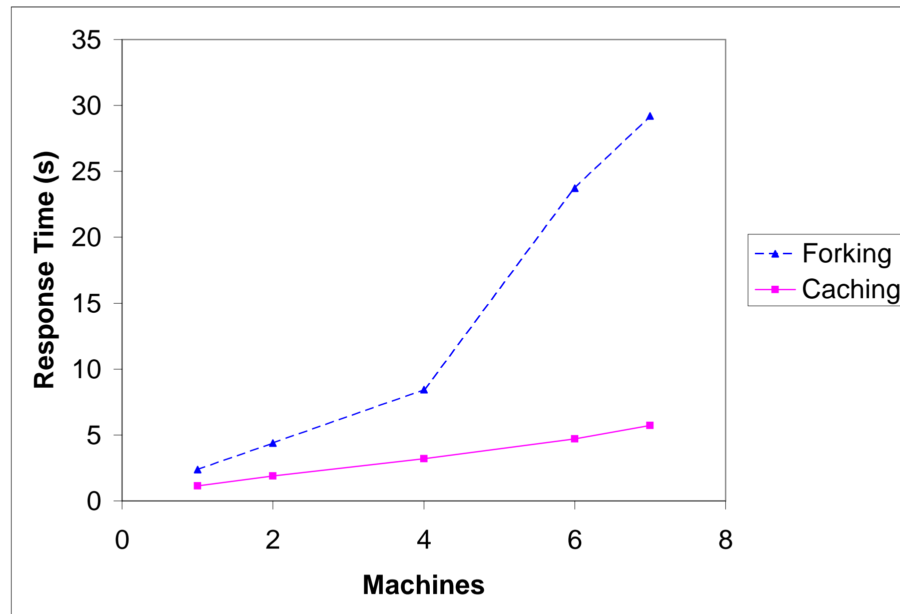


Figure 5.4: Performance improvement of CGI object caching.

were used for the experiments. One of the machines acted as the HTTP server running Apache 1.2.4. Between one and seven of the other machines acted as clients, continuously requesting the results of executing a small CGI script, printenv, which simply prints out the environment variables of the running CGI script. Each client machine forked 40 copies of the same script requesting 200 copies of the same script in a loop. Given the small size of the requests and the replies, network bandwidth was not the bottleneck.

Figure 5.4 describes the relative performance of the baseline versus the modified Apache server under the above conditions. As a point of reference, the printenv script takes .05 seconds to run locally. One client requesting the CGI script in a loop from the unmodified version of Apache averages .18 seconds per request. Using the caching version

of Apache, the CGI script is retrieved in an average of .11 seconds, a 39% improvement over the baseline. From Figure 5.4, it is interesting to note that the relative performance of the caching CGI server improves with increased load. This improvement results from the high overhead of managing and context switching between address spaces as many CGI scripts are forked off by the baseline HTTP server under load.

We expect the above performance disparity to become even more pronounced as the CGI scripts become longer lived (recall that printenv exits after .05 seconds). To test this hypothesis, we re-ran our experiments with the baseline (uncaching) HTTP server returning the contents of a synthetic CGI program which takes 2.5 seconds to execute. Forty clients on one machine averaged 153 seconds to retrieve the object, while forty clients on each of two machines averaged 167 seconds to retrieve the object. Relative to the caching version of the server, the overhead of forking and executing the program slows the server down by nearly two orders of magnitude. Clearly, the savings from caching become more pronounced as the computation cost of the CGI object becomes more expensive.

WebOS Integration

As described in the previous section, one source of performance improvements from integrating TREC with file servers is avoiding the computational costs of dynamically generating Web content. Another advantage of TREC is that such dynamically generated content is stored as a normal file in the file system. These files are invalidated (removed) when the contents of the file are no longer valid (i.e., the dependency for that file has been violated). Thus, a wide-area file system such as WebFS [Vahdat et al. 1998] (described in Chapter 4) or even AFS [Howard et al. 1988] can be used to store and to cache dynamic Web

objects as normal files. In this model the wide-area file system can act as a shared file cache for both the HTTP server and interested proxies, with TREC invalidations maintaining relatively strong consistency semantics among WebFS servers.

Applicability

Our discussion of dynamic Web object caching focuses on CGI programs. Given the inherent inefficiency of spawning a new process for dynamically generated content, a number of systems, such as ISAPI [Microsoft Corporation 1997], NSAPI [Netscape 1997], and FastCGI [Open Market 1997], address this issue either by creating long-lived "server" processes responsible for creating dynamic content or by linking dynamic content producers into the server's address space. Relative to CGI scripts, these approaches offer better performance but sacrifice some of the simplicity of writing CGI scripts. However, TREC can still be used to improve the performance and functionality of these faster dynamic systems. For example, using TREC can eliminate the computation time associated with long-running scripts and any inter-process communication necessary to request dynamic generation of Web objects. Thus, while TREC's baseline performance improvements would not be as impressive relative to systems that avoid fork and exec overhead, TREC still maintains the advantages of using the familiar file system interface to cache content and eliminating any computation time required for dynamically generating Web objects.

Another potential limitation of TREC for dynamic Web object generation is the fact that many dynamic objects are generated as a result of queries to full-fledged databases. In essence, many Web servers act as front ends to a sophisticated DBMS. For example, a user query for a price quote or item availability often translates into a database query. Since

access to database relations cannot in general be modeled by simple file accesses (many databases are implemented on top of raw disks as opposed to the file system for example), TREC cannot interpose on database updates, and hence cannot properly invalidate Web objects based on out-of-date database values.

While the above limitation is inherent, we believe the performance improvements available from dynamic object caching argues for further research into active databases. For example, research efforts into *materialized views* [Gupta et al. 1993, Gupta & Mumick 1995, Colby et al. 1996, Kawaguchi et al. 1996] in active databases [McCarthy & Dayal 1989, Stonebraker et al. 1990, Widom & Finkelstein 1990] has resulted in support for such views in many commercial database systems. Materialized views allow the results of a query to be updated as the tables (and individual cells) used to create the view are updated. Techniques similar to those employed by TREC are used to track view dependencies on individual cells and tables, and to set "triggers" to be fired when a table is modified so that any derived views can be updated as well. An interesting avenue of future research is to evaluate whether materialized database views can be used to cache the portion of dynamic Web objects that generate database queries to obtain their results.

5.4 Related Work

In contrast to our approach using TREC, `make` and related software configuration management tools are currently used to specify and maintain dependency information. Such tools suffer from a number of deficiencies. For example, to manage file and program dependencies, users must manually specify dependency information. Programmers must

specify the dependencies between source files, object files, and executables. If any changes occur users must remember to manually update the dependency information. With TREC, maintaining dependency information is simpler and less error-prone because dependencies are deduced transparently by observing program execution. Another shortcoming of `make` and related tools is the inability to track file lineage. Makefiles only implicitly contain lineage information; if the Makefile changes, the lineage information about existing output files can also be destroyed. TREC associates dependency information with each file, rather than relying on a static description in a separate file.

Several systems have attempted to extend the automatic control of derived objects beyond the simple (but powerful) model used by `make`. DSEE [Leblang & Chase Jr. 1984, Leblang & McLean Jr. 1985], Odin [Clemm & Osterweil 1990] and Vesta [Levin & McJones 1993, Heydon et al. 1997] provide tools for modeling the behavior of programs, enabling the concise specification of derivation rules, and distributing changes to developers. Their declarative style suits large-scale programming environments, which are highly structured and employ a well-defined set of tools (compilers, linkers, etc.). None of these tools provide any assurance of correctness; as with `make`, the user is responsible for describing the complete set of dependencies relationships to the configuration manager. In contrast to unmake, users of these systems must tell the system how tools use files, whereas unmake simply observes and gathers the information in the background.

Odin relies heavily on the use of naming conventions: the name of a file fully specifies how it was derived. This restriction would not work well for the ad-hoc, highly parameterizable methodology used in less-structured environments. Like tmake, Odin implements

transparent re-creation of files. A sentinel in Odin is a data object that is automatically regenerated (if necessary) at the time a user requests it, based on rules that were specified in advance for objects of its type.

VOV [RTDA], a configuration management toolkit, is similar to TREC, in that it observes program invocations to generate a trace of lineage information. However, VOV is limited to a specialized application domain (Electronic CAD), and it requires assistance from tool programmers. Each tool explicitly reports the files it will read and write. By contrast, unmake observes filesystem activity at a low enough level so that it is unnecessary to modify tools to work with TREC.

Web caching is currently a popular area of research. Harvest [Chankhunthod et al. 1996] and Squid [Squ 1996] are efforts into hierarchical Web proxy caching. We believe that such caching efforts would benefit from our work in dynamic object caching. Gwertzman and Seltzer [Gwertzman & Seltzer 1996] recently proposed using the Alex protocol [Cate 1992] for maintaining cache consistency across the wide area. While this protocol provides weaker consistency guarantees than a wide-area file system, it would be simpler to deploy and could be used in our model for caching dynamic Web objects at proxy caches. Finally, one proposal advocates using HTTP profiles to predict accesses to dynamically generated data, allowing servers to pre-generate potentially expensive pages in anticipation of user requests [Schechter et al. 1998].

Iyenger and Challenger [Iyenger & Challenger 1997] have implemented a caching system and API as part of IBM's Web server that allows for caching of dynamic data. Their system allows for caching of dynamic data generated by arbitrary programs. Their work

requires explicit invalidation of dynamically generated content by the Web server, whereas TREC takes steps to automate this procedure. Further, TREC caches dynamic objects as normal files, simplifying system integration with existing Web servers.

The performance results presented in Section 5.2.2 here are similar to overhead studies of process migration and remote execution in Sprite [Douglis & Ousterhout 1991]. In Sprite, a number of system calls must be forwarded to the "home node" of a job for local processing. While the overhead of these operations in isolation is high, the overall perceived slowdown is tolerable because of the low frequency of forwarded operations. Similar to TREC, I/O system calls such as `read` and `write` are processed locally in Sprite.

5.5 Summary

An increasing number of Web services dynamically generate content in response to client queries. Currently, these dynamically generated objects cannot be cached, negatively impacting overall system performance. This chapter describes techniques to enable caching of a certain class of dynamically generated Web objects by managing interactions between input files, output files, and development tools. Transparent Result Caching (TREC) automatically and unobtrusively constructs dependency information by observing program behavior. Experience with TREC shows that caching dynamic content reduces server load, and client latency. Further, through integration with the wide-area file system described in Chapter 4, TREC can cache dynamic content at proxies as well as servers, further improving client latency and reducing consumed wide-area bandwidth.

This chapter also demonstrates the generality of the TREC framework by de-

scribing its use in supporting two additional applications, unmake and transparent make. Unmake allows users to query for process lineage information, returning the full chain of processes, command line parameters, and environment variables used to create a file. Transparent make uses the process lineage information from unmake to provide functionality similar to UNIX `make`, with the added advantage of freeing users from manually specifying file dependencies.

Relative to the wide-area system design goals described in Chapter 2, TREC primarily addresses issues of *performance* and *backward compatibility*. TREC improves system performance by enabling the caching of certain classes of previously uncacheable dynamically generated data. It is able to perform this caching without requiring any changes either to Web servers or to the programs used to generate dynamic content. To some extent, TREC also aids in providing a *uniform interface to remote resources* by allowing Web content to be cached and replicated independent of how it is generated.

Chapter 6

Security

The previous chapters described a flexible naming interface, a persistent storage system, and a technique for caching dynamically generated wide-area content. In this chapter, we describe CRISIS[1], a system supporting secure and authenticated access to remote resources. A principal contribution of the system is transfer certificates, lightweight and revocable capabilities that enable the fine-grained transfer of rights between multiple administrative domains.

6.1 Overview

A primary challenge to building distributed applications that benefit from seamless integration with global data and computational resources is the lack of a general, coherent, and scalable wide-area security architecture. Before users are willing to allow sensitive data and programs to migrate to remote machines, the system must provide guarantees about the

[1]The contributions described in this chapter focus on the design of CRISIS and its integration with a general-purpose system supporting mobile computational agents. The implementation of CRISIS is joint work with Eshwar Belani, with a detailed description appearing elsewhere [Belani 1998].

integrity of private resources. The security system of our wide-area computation system is responsible for providing these guarantees. The main responsibilities of the security system are authentication, proving one's identity to remote resource providers, and authorization, determining whether access to a resource should be granted once the identity of the requester is authenticated.

In the context of this dissertation, the security system must meet the high level goals for a wide-area computation system as described in Chapter 2. First, the security system should not make remote resources significantly more difficult to use than local resources. However, some added complexity may result from multiple security policies in different administrative domains. Following this theme, the security system must also be flexible, allowing administrators to define multiple policies, for arbitrary levels of caution. Next, performance should not be compromised by introducing security mechanisms. Finally, the security system should be backwardly compatible with existing applications to as large an extent as possible.

To motivate the requirements and design of a wide-area security system, consider the following scenario. A user wishes to run a highly parallel simulation, utilizing available remote computational resources whenever possible. To complete its task, the simulation needs access to certain resources located in the user's home domain, for instance, input/output files or a local sensor device. The simulation runs with a monetary budget, utilizing wide-area resources that satisfy its utility curve—how much it is willing to pay per unit of computation power. From this scenario, we extract the following requirements for a wide-area security system supporting mobile computation:

- *Privilege Isolation*: Remote machines will always be less trusted and more prone to compromise than local machines. Once a machine is compromised, the attacker gains all privileges associated with all programs running on that machine. To minimize the danger from such compromise, it is important to assign to remote programs the minimum set of privileges required to complete their task. Today, applications typically run with *all* of a user's privileges. In the above example, the simulation should have access only to input/output files and the local sensor device, and not, for example, to the user's private mail spool.

- *User Control*: Users should have final control over the distribution of their privileges. While this distribution can be automated at the user's discretion, remote processes should not have the ability to transfer privileges to third parties without receiving final authorization from the user that spawned the original process. In the example above, if a portion of the simulation running remotely spawns a task to a third domain, it should not be able to transfer any privileges to the new task without user authorization. In this way, privileges cannot be transferred to a domain that is unequivocally untrusted by the end user.

- *Fault Tolerance*: As discussed in Chapter 1, failures are more common across the wide area because of the sheer scale of the Internet. The security system must be designed to withstand and mask host failures. Thus, a host failure should not cause an application with insufficient credentials to be granted access to a resource nor should an application with sufficient credentials be denied access to a resource. Further, as much as possible, the above simulation should continue making forward progress even

if hosts in the user's local domain become unavailable.

As will be further discussed in Section 6.6, existing security systems do not meet the above goals because they do not consider: i) moving computation among multiple administrative domains, ii) the high failure rates across the Internet, and iii) the need for fine-grained transfer of rights among processes. This chapter describes two principal contributions in the context of wide-area security. First, we demonstrate that it is possible to extract principles from existing security and distributed systems to design and build a security system that meets above the goals, i.e., that is appropriate for wide-area mobile computation. Second, we introduce transfer certificates as a simplifying security concept, supporting the fine-grained transfer of rights between multiple administrative domains.

The rest of this chapter describes CRISIS in more detail. First, we describe the design principles that guide the system architecture in Section 6.2, leading to a description of the CRISIS architecture in Section 6.3. A detailed example of CRISIS use in a number of common scenarios is presented in Section 6.4. We evaluate the performance of our implementation in Section 6.5, and discuss work related specifically to security in Section 6.6. We summarize our results in Section 6.7.

6.2 Design Principles

A goal of the CRISIS architecture is to demonstrate that principles from existing distributed systems can be applied to increasing the security of wide-area distributed systems. To avoid an ad hoc design where features are thrown together in an attempt to prevent all known types of security attacks, the CRISIS design is driven by examining dis-

tributed systems in related areas, extracting design principles, and systematically applying them to our system architecture. In particular, our system design is inspired by principles elucidated from the SRC security work [Lampson et al. 1991]. In this section we present the principles underlying the design of CRISIS, leading to a description of the CRISIS architecture in Section 6.3. These principles include:

- Redundancy: No single point of failure should be present in the security system; any attack must compromise two different systems in two different ways. For example, every certificate (such as that identifying a user's public key) is revocable within a few minutes; thus, an attacker must not only steal a private key, but must do so without detection or must also corrupt the revocation authority. The notion of using redundancy to improve security is an old one, but it has not been systematically applied. For example, Internet firewalls are used to protect organizations from security attacks from the outside world, to hide the fact that most local operating systems are notoriously insecure. Unfortunately, this has reduced the pressure on local operating systems to improve their security, so that, once inside a firewall, an attacker usually has nearly free reign. Similarly, Internet browsers purport to be able to safely execute Java applets, supposedly rendering traditional operating system security irrelevant. The ongoing discovery of security holes in Java verifier implementations [Dean et al. 1996, Sirer et al. 1997], however, has led us to run remotely executing programs in a restricted environment [Goldberg et al. 1996], *in addition to* using a verifier.

- Caching: CRISIS caches certificates to improve both performance and availability in the presence of wide-area communication failures; while all certificates are revocable,

they are given a revalidation timeout to hide short-term unavailability of the revocation authority due to reboots or Internet partitions. This principle was inspired by research in mobile file systems which argued that caching can improve availability, e.g., for disconnected operation [Kistler & Satyanarayanan 1992].

- Least Privilege: Users should have fine-grained control over the rights delegated to programs running on their behalf, whether local or remote; to be useful, this control must be easy to specify. CRISIS provides two mechanisms to support least privilege: *transfer certificates*, limited unforgeable capabilities which allow for confinement and can be immediately revoked, and lightweight roles that can be created by users without the intervention of a system administrator and without requiring changes to access control lists.

- Complete Accountability: CRISIS logs all evidence used to determine if a request satisfies access control restrictions locally at the machine guarding access. Most existing systems log only coarse-grained authentication information, making accountability in the face of rights transfer difficult (e.g., some systems may want to grant a request only if every member along a chain of delegation is trusted). Our design differs from previous efforts to add capability-like certificates to Kerberos [Neuman 1993], which require distributed logging by all ticket granting servers involved in propagating a request in the wide area.

- Local Autonomy: Each user identity is associated with a single administrative domain, and that domain is solely responsible for determining how and when user privileges are distributed beyond the local administrative domain. Each local domain is also

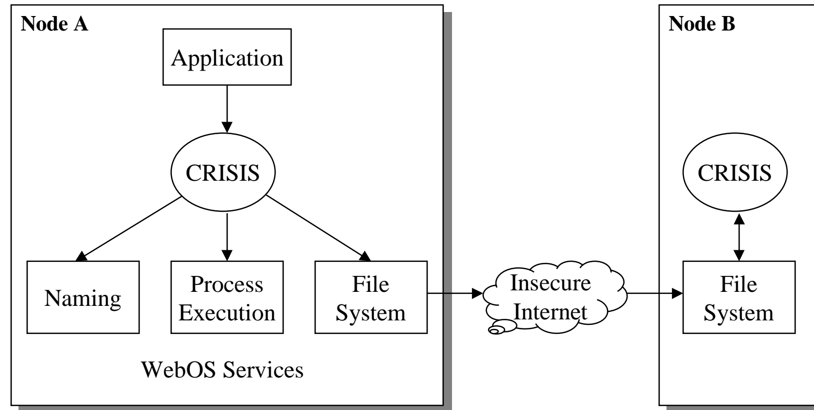


Figure 6.1: CRISIS interaction with system services (naming, remote execution, and the file system).

responsible for determining the level of trust placed in any given remote domain. A design relying on deference to a global, centralized authority is not only less flexible, but less likely to be widely adopted [Birrell et al. 1986].

- Simplicity: Simple designs are easier to understand and to implement. Further, simplicity makes it is easier to prove properties about a system's design, implementation, and correctness. In the context of security, simplicity is especially important to minimize the probability of a security hole resulting from an implementation error. A provably secure but highly complex system architecture is unlikely to be either properly implemented by system designers or properly understood by end users (for example, leading to errors in setting up access control lists).

6.3 System Architecture

Given the goals for a wide-area security system presented in Section 6.1 and the design principles presented in Section 6.2, we now describe the CRISIS architecture designed to meet these goals and requirements. Figure 6.1 shows how CRISIS moderates interaction between applications and some of the other aspects of remotely programmable resources discussed in this dissertation. For example, CRISIS mediates application interaction with the file system, providing secure access to a global persistent storage space. The file system also utilizes CRISIS to authenticate its identity to remote counterparts on other machines. In this section, we provide a high-level view of the system architecture before detailing example usage in the next section.

In the following discussion, we assume the presence of three basic entities, adapted from the SRC logic [Lampson et al. 1991]:

- *Principals*: Principals are sources for requests. Examples of principals include users and machines. Principals make statements (requests, assertions, etc.), have names, and can be associated with privileges.

- *Objects*: Objects are global system resources such as files, processors, printers, memory, etc.

- *Reference Monitors*: Once an access request from a principal to an object is authenticated, the reference monitor determines whether or not to grant the principal access to the object.

To motivate the CRISIS architecture, we will use the sample scenario depicted

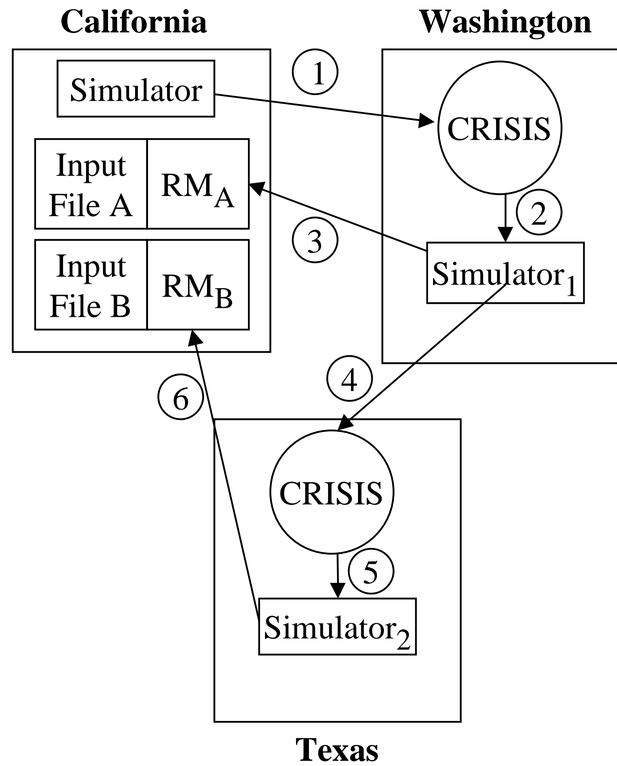


Figure 6.2: Sample CRISIS execution scenario: delegating execution remote administrative domains.

in Figure 6.2 (building on some of the sample applications described in Chapter 1). In this example, a user starts a simulation program on a machine in the user's home domain, located in California. While running, the simulation becomes aware of available cycles on a machine located in Washington and decides to farm out a portion of the simulation to Washington. This portion of the simulation needs access to input files A and B. The simulation in Washington, in turn, farms out a portion of the computation to Texas. The process running in Texas requires access only to file B to complete its task. In this example, RM_A and RM_B are reference monitors that gate access to files A and B respectively. If,

initially, only the California machines have access to files A and B, the following sequence of steps must be taken to carry out this scenario from a security perspective:

```
California states that Washington can access files A and B until time T_1.
California requests that Washington execute a job on its behalf (steps 1 and 2).
Washington requests access to A from RM_A (step 3).
Washington states that Texas can access B until time T_2.
Washington requests that Texas execute a job on its behalf (steps 4 and 5).
Texas requests access to B from RM_B (step 6).
```

To carry out the above scenario in our model, the security system must support:

- Statements: Statements may be requests, declarations of privileges, or transfer of privileges. The identity of the principal making a statement must be verified, and all statements must be revocable.

- Associating privileges with processes: While individual machines may possess a large set of privileges, processes running on the machine only have access to a subset of these privileges. Similarly, users may only wish to grant a subset of their available privileges to each of their programs. This is especially true in the wide area where different levels of trust are present for each administrative domain.

- Distributing statements across the wide area: A protocol for trust between administrative domains is necessary to allow for validation of privileges and identities across administrative boundaries.

- Time: The transfer of privileges can only be valid for a limited period of time. CRISIS requires a clear protocol for reasoning about time, and a methodology for isolating failures in cases where clock skews lead to a security breach.

- Authorization: When a reference monitor receives a request to access an object, it needs to determine the identity of the requester, ascertain the principal's privileges, and finally decide whether the request is authorized.

Our solutions to each of the above requirements are described in the following subsections.

6.3.1 Validating and Revoking Statements

A principal contribution of CRISIS is the introduction of *transfer certificates*, lightweight and revocable capabilities used to support the fine-grained transfer of rights. Transfer certificates are signed statements granting a subset of the signing principal's privileges to a target principal. Transfer certificates can be chained together for recursive transfer of privileges. The entire chain of transfers is presented to reference monitors for validation, allowing for confinement of rights (e.g., a reference monitor can reject access if any principal in a chain of transfers is not trusted). Transfer certificates form the basis of wide-area rights transfer, supporting operations such as delegation, privilege confinement, and the creation of roles (as described below).

All CRISIS certificates must be signed and counter-signed by authorities trusted by both endpoints of a communication channel. A Certification Authority (CA) generates *Identity Certificates*, mapping public keys to principals. In CRISIS, CA's sign all identity certificates with a long timeout (usually weeks) and identify a locally trusted online agent (OLA) that is responsible for counter-signing the identity certificate and all transfer certificates originating with that principal. The endorsement period of the counter-signature

is application-specific, but typically on the order of hours. Redundancy employed in this fashion offers a number of advantages: (i) to successfully steal keys, either both the OLA and CA must be subverted or the CA must be subverted undetected, (ii) the CA is usually left offline since certificates are signed with long timeouts, increasing system security since an offline entity is more difficult to attack, (iii) a malicious CA is unable to revoke a user's key, issue a new identity certificate, and masquerade as the user without colluding with the OLA [Crispo & Lomas 1996], and (iv) system performance is improved because certificates can be cached for the timeout of the counter-signature, removing the need for synchronous three-way communication in the common case.

Transfer certificates can be revoked modulo a timeout. Revocation is used not only for exceptional events such as stolen keys, but also applies to common operations such as revoking the rights of a remote job upon its completion or revoking the rights of a login session upon user logout. To revoke a particular privilege, the OLA that endorses the certificate must be informed that the certificate should no longer be endorsed. Once the timeout period for the endorsed certificate expires, the rights described by the certificate are effectively revoked because the OLA will refuse re-endorsement for that certificate.

While a detailed description of the implementation of transfer certificates appears elsewhere [Belani 1998], Figure 6.3 summarizes the structure of transfer certificates. A transfer certificate is a chain of X.509 [Con 1989] certificates. The first certificate is an identity certificate describing the principal wishing to make the transfer. Each principal is identified by their public key, K_0 in the case of the identity certificate. Each subsequent certificate transfers a subset of the issuer's available privileges to another principal. In this

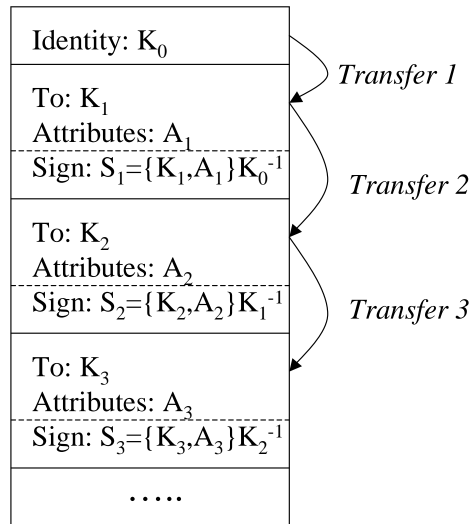


Figure 6.3: Structure of a CRISIS transfer certificate.

example, K_0 transfers privileges described by A_1 to K_1. Note that for this first transfer, local reference monitors will ensure that principal K_0 actually possesses the privileges described by A_1 before granting access to any requests made by K_1. This certificate is signed by encrypting an MD5 hash of K_1 and A_1 in the private key of principal K_0 (K_0^{-1} in this case). As shown in this example, the certificates can be arbitrary chained, with K_1 transferring privileges to K_2, who in turn transfers privileges to K_3, etc.

A properly formed transfer certificate can only specify a subset of available privileges from principal to principal (i.e., a certificate holder cannot grant privileges that it does not possess). The privileges described by $A_0, ..., A_3$ also contain timestamps describing the time period during which the transfer is valid. Properly formed time certificates can only transfer a privilege while it is still valid (i.e., a certificate holder cannot avoid timeouts simply by transferring a privilege back to itself with a longer timeout). This structure of

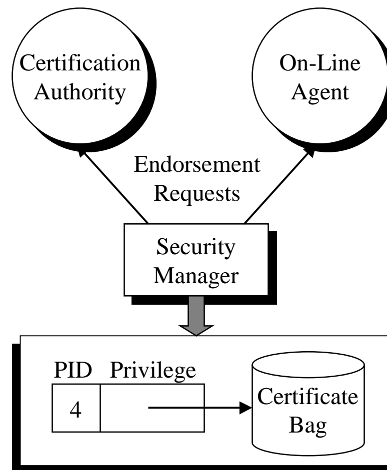


Figure 6.4: CRISIS security managers associate processes with privileges.

transfer certificates allows for complete accountability, as reference monitors can validate the entire chain of transfers before granting access, and also facilitates revocation, as a transfer certificate becomes invalid as soon as any of its timeouts expire.

6.3.2 Processes and Roles

Security Domains

Given the ability to authenticate principals, CRISIS also requires a mechanism for associating privileges with running processes. Each CRISIS node runs a security manager responsible for mediating access to all local resources and for mapping credentials to *security domains*. Figure 6.4 depicts the relationship between security managers, security domains, and the certification authorities and online agents responsible for ensuring that all certificates are properly endorsed. In CRISIS, all programs execute in the context of

a security domain. For example, a login session creates a new security domain possessing the privileges of the principal who successfully requested login. As will be described in Section 6.4.1, a security domain, at minimum, is associated with a transfer certificate from a principal to the local node allowing the node to act on the principal's behalf for some subset of the principal's privileges.

Processes are able to access wide-area resources through resource providers responsible for managing each remote resource, such as processor cycles or disk space. In conjunction with security managers, resource providers determine the access privileges of processes requesting resources. CRISIS nodes currently run the following resource providers, each with their own set of reference monitors:

- *Process Managers* - A Process Manager is responsible for executing jobs on requested nodes. The Process Manager identifies the security domain associated with a request, obtains the credentials associated with the domain from the security manager, and then attempts to satisfy the request.

- *WebFS* - A WebFS server implements a cache coherent global file system. Similar to the Process Manager, upon receiving a file access request, the WebFS server determines the security domain from the security manager. Using this information, the WebFS server determines whether the access should be granted or denied.

- *Certification Authorities* - As described above, CA's take requests for creating identity certificates. The CA maintains a reference monitor with the list of principals authorized to create, modify, or invalidate identity certificates.

The interaction between resource providers, security domains, and security man-

agers are described through the CRISIS protocols for login, file access, and remote process execution in Section 6.4.

Roles

In the wide area, it is vital for principals to restrict the rights they cede to their jobs. For example, when logging into a machine, a principal implicitly authorizes the machine and the local OS to speak for the principal for the duration of the login session. Whereas with private workstations, users generally have faith that the local operating system has not been compromised, confidence is more limited when using anonymous machines across the wide area, for example, to run large-scale simulations. Roles associate a subset of a user's privileges with a name, allowing users a convenient mechanism for choosing the privileges transferred to particular jobs.

A principal (user) creates a new role by generating an identity certificate containing a new public/private key pair and a transfer certificate that describes a subset of the principal's rights that are transferred to that role; an OLA chosen by the principal is responsible for endorsing the certificates. Thus, in creating new roles, principals act as their own certification authority. The principal stores the role identity certificate and role transfer certificate in a *purse* of certificates that contains all roles associated with the principal. The purse is stored in the principal's home domain. While it is protected from unauthorized access by standard OS mechanisms, the contents of the purse are not highly sensitive since each entry in the purse simply contains a transfer certificate naming a role and potentially describing the rights associated with that role. The principal also stores each role's private key —encrypted by a password unique to the role —in the file system.

CRISIS roles are more lightweight than the roles described in other security systems (e.g., [Lampson et al. 1991]). First, they can be created by the user without requiring intervention from a centralized authority, allowing the CA to remain offline. Next, while ACLs can be modified to describe a particular role's privileges, roles can also act as persistent lightweight capabilities. The transfer certificate used to create the role can describe the exact access rights possessed by the role (e.g., read access to files A, B, and C).

In CRISIS, creating a new group is similar to creating a new role. A principal creates a new group by acting as a CA to create an identity certificate naming the new group. The creating principal then signs transfer certificates to all group members, specifying both membership and any update privileges associated with the group, for example, whether the member has the ability to add or remove other group members. The newly created group name can then appear on ACLs like any other principal name.

6.3.3 Hierarchical Trust

Mediating trust among certification authorities in different administrative domains across the wide area is a difficult problem. CRISIS does not innovate in this area, but adopts the model presented in [Birrell et al. 1986]. Thus, we assume the presence of multiple, autonomous administrative domains. CA's are arranged hierarchically, with individual CA's determining which parents, siblings, or children are trusted (and to what extent).

A Certification Authority cannot speak for a principal who belongs to a descendant's domain in the hierarchy. In this way, separate administrative domains are able to maintain local autonomy. Thus, a principal receiving a certificate endorsed by a CA in a foreign administrative domain believes the certificate valid only if a *path of trust* is present

from the local domain to the remote domain. The presence of such a path is determined by traversing the least common ancestor of the two domains in the CA hierarchy. Of course, such a path of trust need not be established or traversed on every access. The identity of trusted administrative domains can be cached, making the determination of a path of trust a fast operation in the common case. Further, principals trust their CA's closer to them more than distant CA's (based on relative positioning in the hierarchy). Thus, if an ancestor of a CA is compromised, transactions among local principals are not affected, increasing system availability and keeping trust as local as possible. Finally, if the root CA in the hierarchy becomes a performance or availability bottleneck, replicating the root should be fairly straightforward.

6.3.4 Time

Since all CRISIS certificates contain timeouts and since these certificates are distributed across the wide area, the system must make assumptions about clock synchronization. Further, CRISIS must protect against security attacks exploiting a node's notion of time. If time on a machine is corrupted, statements can be used beyond their period of validity.

Today, most workstations possess fairly accurate clocks that are periodically synchronized with any of a number of external sources. However, time-sensitive applications (and hence, reference monitors) may require guarantees above and beyond such loose synchronization, for example, that the local clock is periodically synchronized with a trusted external source. Other applications will require an invoice of all assumptions made during a computation in the case where data is corrupted or leaked to determine the exact cause

of the corruption, assuring complete accountability.

For CRISIS, we assume the presence of replicated, trusted time servers [Mills 1991]. Principals producing certificates with timeouts (e.g., CA's and OLA's) contact these servers periodically to obtain signed certificates containing the current time to validate the principal's notion of time. If the principal's time differs by more than a few seconds (i.e., within network delay bounds) from the time supplied by the server, the principal assumes that either the time server or the local operating system/hardware has been compromised (to determine which, a second server might be contacted). Such communication with time servers need not be synchronous, since the time certificates can be cached to prove recent synchronization.

In CRISIS, time certificates are provided to resource managers to prove that a node's notion of time closely matches the value reported by a trusted time server at some recent point in the past. CRISIS identity and transfer certificates report time values (such as expiration time) relative to the value contained in a chained time certificate. While use of time certificates does not guarantee that time-based attacks can be avoided or prevented, it can aid in determining the cause of certain security violations *post-mortem*. Thus, if a security breach is detected, analysis of certificates used to gain unauthorized access can be used to determine the cause of the attack. For example, examination of the certificates may show that a node attempted to use an expired time certificate or that a time server was compromised and reported faulty values of time.

6.3.5 Authorization

The previous sections describe the process of securely transmitting a request across the wide area. Once this has been accomplished, authorization, the process of determining whether the principal making a request should be granted access, is the remaining task. Traditional security systems employ either access control lists (ACL's) or capabilities to describe the principals authorized to access an object.

ACL's associate with each object a list of principals authorized to access that object. They have the advantage of being simple to understand. However, the list can grow cumbersome, especially in the context of the wide area where large numbers of principals may have access to a particular object. ACL's can also be error prone: for example, if a principal requires short-term access to an object, the principal must be added temporarily and then removed once its task is completed.

Capabilities are opaque and unforgeable tickets associated with every object. A capability is distributed to a principal to grant access to a particular object. Capabilities simplify the process of delegation and rights transfer. One advantage of capabilities is that transferring access rights from one principal to another is a relatively simple process. However, they are limited by the problems of revocation and containment. Consider the scenario where multiple principals are given capabilities for a single object. It is difficult for the object owner to revoke the privilege of just one of the principals holding a capability. A robust system must maintain a list of all principals maintaining a capability, send revocations out to each one, and then distribute new capabilities to all principals, save the one whose privilege is being revoked. This process is not likely to scale across the wide area,

with many principals with intermittent connectivity accessing millions of different objects. Similarly, principals can arbitrarily transfer capabilities, making it difficult for an object owner to contain transfer privileges to untrusted principals.

Since both ACL's and capabilities have advantages in different situations, we use a hybrid model incorporating many of the benefits of both ACL's and capabilities through the use of transfer certificates. For maximum flexibility, the targets of CRISIS ACL's are service-specific. For example, file ACL's contain lists of principal's authorized for read, write, or execute access to a particular file. ACL's contain the list of principals that have long-term access to a particular object. These principals can then create Transfer Certificates (short-lived and revocable capabilities) to temporarily grant access to a particular object to other principals. Thus, entries in an ACL act as anchors for transfer certificates. To successfully request access to an object, the requesting principal must either appear on an ACL or contain a set of transfer certificates that can be chained back to a principal appearing on the ACL.

A process requests access to an object by contacting the object's reference monitor. In CRISIS, reference monitors are implemented on a service-by-service (e.g., file service) basis and form separate modules in the security manager. For example, the WebFS reference monitor is a separate module in the CRISIS security manager.

To determine whether a request for a particular operation should be authorized, the reference monitor first verifies that all certificates are signed by a public key with a current endorsement from a trusted CA and OLA. In doing so, the reference monitor checks for a path of trust between its home domain and the domains of all signing principals (as

described in Section 6.3.3). In the common case, the existence of such paths is cached. The reference monitor then checks that none of the timeouts have expired and that time is reported relative to a value stated by a trusted time server (again by checking for a path of trust to the time server).

Once the above steps are taken, the reference monitor is assured that all certificates are well-formed and valid. Given this knowledge, the reference monitor then reduces all certificates to the identity of single principals. For transfer certificates, this is accomplished by working back through a chain of transfers to the original granting principal. The requesting principal is able to act on the behalf of the reduced list of principals. Finally, the reference monitor checks the reduced list of principals against the contents of the object's ACL, granting authorization if a match is found.

6.3.6 Discussion

Given the description of the CRISIS system architecture above, we discuss some of our underlying design decisions in this section. One of the first steps in providing a secure Internet information system is to allow for encrypted, authenticated communication between arbitrary endpoints over an inherently insecure wide-area network. Traditionally, the two choices for encryption and authentication are using secret key or public key cryptography. Encryption ensures that an eavesdropping third party cannot alter the integrity or determine the content of the communication. Authentication allows for the identity of the principal at the opposite end of a communication link to be securely identified.

We choose public key over secret key (though one can be simulated with the other [Lampson et al. 1991]) because of the synchronous communication usually required

by secret key systems. Secret key systems require a trusted third party that shares a secret with every potential communication endpoint. Although this requirement impacts system performance and availability by imposing an extra step in initiating communication, it is reasonable in the local area because the number of communication endpoints are limited and the network is more reliable. In the wide area, such a requirement strains system scalability because synchronous communication with a hierarchy of trusted third parties is required. Public key systems also require trusted third parties to produce certificates identifying principals with their public keys. However, these certificates can be cached (with a timeout), removing the need for synchronous communication with a third party to set up a communication channel. Allowing for direct communication in this fashion offers two advantages. First, system availability is improved because an unavailable third party does not necessarily prevent communication. Second, system performance is increased by removing a communication step to a third party behind a potentially slow link. Note that our system addresses a well-known problem with public key systems: the need to inform all entities on the Internet whenever a private key is compromised. By using Online Agents to endorse certificates with short timeouts, a compromised key will not remain valid indefinitely.

In addition to public key encryption, we employ a number of other technologies to assist in development and to reduce the chance of introducing security flaws. We use Janus [Goldberg et al. 1996] to "sandbox" locally running applications that are not fully trusted. Janus runs at user-level, employing the UNIX System V `proc` file system to intercept potentially dangerous system calls and to disallow accesses to resources outside of each

process's defined sandbox. Our mechanisms for remote process execution are described in more detail in Section 8.5. CRISIS also employs the Secure Socket Layer (SSL) [Hickman & Elgamal 1995] protocol to provide transport network layer privacy and integrity of data, using encryption and message authentication codes. SSL supports a wide variety of cryptographic algorithms and is being deployed into wide-area applications. Finally, as will be described in the next section, we use the X.509 syntax [Con 1989] to encode all certificates in CRISIS. The ITU-T Recommendation X.509 specifies the authentication service for X.500 directories, as well as the X.509 certificate syntax. The X.509 certificate syntax is supported by a number of protocols including PEM, S-HTTP, and SSL.

6.4 CRISIS Protocols

Given the above high level description of the CRISIS architecture, we will now describe how the various system components interact to allow secure execution of routine tasks, including login, file access, and job execution (operations that potentially cross machine and/or administrative boundaries).

6.4.1 Login

The goal of login is to authenticate a principal to a node and to create a shell process with the principal's privileges. We achieve this by associating a security domain on the login node with a transfer certificate granting the node the privileges of the login role. We assume that each role is associated with a *home domain* and that users wishing to log in must authenticate their identity to their home domain. By minimizing the trust placed

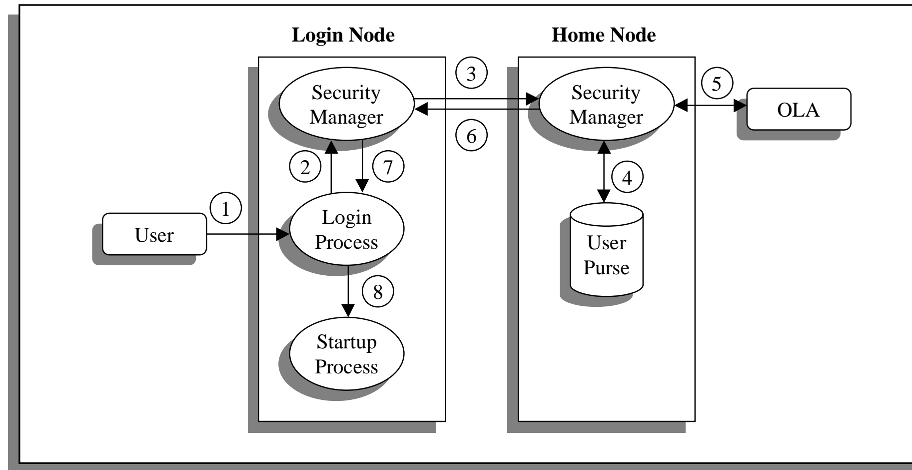


Figure 6.5: Authenticating principal login in CRISIS.

in the login node and by choosing a role with an appropriately small set of privileges, we enhance security and reduce the danger of key compromise (private keys never leave the home node). Further, the home domain possesses autonomy in determining the set of sites where principals are allowed to log in. The disadvantage of this approach is that all attempts to authenticate the user must involve the home domain, potentially decreasing system availability. One approach to relaxing this constraint is using smart cards in place of home domain machines; we outline how this scheme would be integrated in the CRISIS architecture below.

Initially, we consider the following login scenario: a principal accesses a shared workstation by entering a globally unique role name (e.g., *remote_tom@cs.washington.edu*). This role corresponds to the level of trust the principal places in the login node, and to the amount of rights required to successfully complete the desired tasks, for example, reading

mail. Once the role is chosen, the principal trusts the OS of the login node with all the privileges associated with that role, since the OS is free to masquerade as the role (at least for the duration of the granted transfer certificate). The login sequence is described in Figure 6.5 and is summarized below:

1. The principal enters a suitable role name (based on the level of trust placed in the login node) and then supplies the appropriate password.

2. The login process transmits the role name and password to the local security manager.

3. The security manager on the login node determines the home node for the specified role name (currently explicitly described in the role name) and contacts the security manager at this home node. The security managers mutually authenticate their identities to one another by transmitting identity certificates over an SSL connection. Upon receiving the identity certificate, each security manager determines whether a path of trust is present from its local certification authority to the certification authority identified by the identity certificate (as described in Section 6.3). As part of this mutual authentication, the login node transmits a certificate signed by a local system administrator stating that the administrator believed that the login node had not been tampered with at boot time and a description of the kernel running on the machine. The home node can then use this information to aid in the login authorization decision, for instance, rejecting requests for login to a machine running a kernel with known security holes.

4. The home node uses the password to decrypt the locally stored private key for the

specified role name. If the key is successfully decrypted, the home security manager looks up the credentials associated with the specified role in the principal's certificate purse.

5. The certificates are presented to the home node OLA for endorsement. The OLA sends back the endorsed certificates. The home domain's security manager can optionally update the principal's purse with the endorsements.

6. The home domain signs transfer certificates (on the principal's behalf), transferring all the privileges associated with the specified role to the security manager on the login node and transmits these transfer certificates to the security manager on the login node.

7. In turn, the login node's security manager transmits the transfer certificates to the process that originated the login request.

8. If the login is successful, the login process creates a login shell for the user. The security manager creates a new security domain, associating the login shell with the set of transfer certificates transmitted by the home node (giving those privileges to the startup process).

For a successful login, the result of the above sequence of steps is to allow the login node to act on behalf of the role for a time period determined by the home domain's security manager. Any processes spawned by the login shell are by default assigned to the same security domain. The protocols employed to access resources through this security domain are detailed in the next two subsections.

One limitation of the above scheme is that the login node is trusted with the role's password (though not its private key). A well-behaved machine will erase the password from memory as soon as it is transmitted to the home domain. Similarly, the local file and memory cache should be flushed upon logout to ensure that private state is not leaked even if the machine is compromised at a later time. Another limitation with the protocol is that the principal's home domain must be available at the time of login (i.e., no network failures/partitions), or authentication becomes impossible. We believe that both of these limitations are inherent given the current state of hardware/software systems. However, our design also supports the use of specialized, trusted hardware (such as smart cards or a portable computer) to enhance security (keep password from local machine) or higher availability (no need to contact home domain for login) or both.

A trusted hardware proxy, such as a smart card or a portable computer, can also be used to separate the tasks of authentication (principals proving their identity) and authorization (determining that the principal is privileged to login to the remote machine with the specified role)[2]. The proxy can store both a role's private key and the associated password to implement a challenge/response protocol at login as follows[3]. When the home domain is notified of a login attempt, it encrypts a random number in the role's public key and transmits the result to the login machine's security manager. The proxy prompts the user for a password needed to decrypt a locally stored (but encrypted) private key file for the role. The private key is used to decrypt the number, which is then transmitted back

[2]Even if a smart card is used for authentication, it may still be desirable to require joint endorsement of a login session from both the target login machine and the user's home domain. Thus, if remote login is locally authorized, the home domain may disallow the login as a matter of policy. For example, login to a competitor's machine may be disallowed to prevent spoofing attacks.

[3]We present one simple scheme; other zero-knowledge algorithms such as Fiat-Shamir [Fiat & Shamir 1987] could also be utilized.

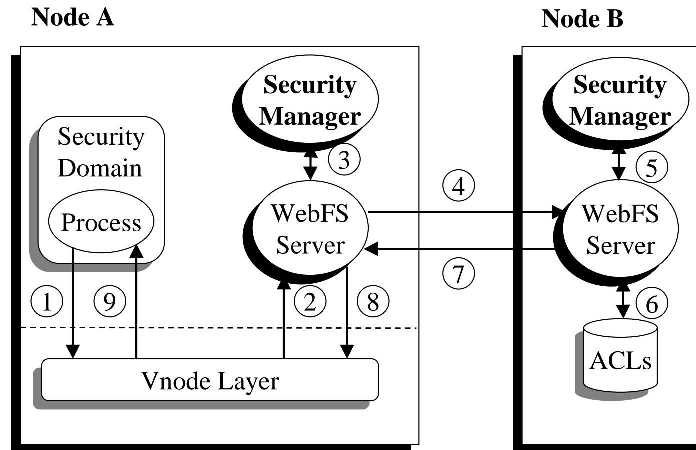


Figure 6.6: CRISIS integration with WebFS.

to the home domain. If the correct number is returned, the process of producing proper transfer certificates is followed as enumerated above. As a separate optimization, the task of authorization can be co-located with the hardware proxy to improve both the performance and availability of the login process (while trading off centralized autonomy in authorizing the locations from where particular roles are allowed login).

6.4.2 Accessing a Remote File

In this section, we demonstrate how the privileges associated with a CRISIS security domain are used to access a remotely stored file. Users access distributed files through the global file system, WebFS (described in Chapter 4). Recall that WebFS is implemented at the vnode level [Kleiman 1986] allowing unmodified applications access to global persistent storage.

We illustrate the protocol for secure access to remote files through the following

example. A process on node A opens a file physically located on node B. The steps taken by WebFS and CRISIS in determining whether access should be granted are depicted in Figure 6.6, with individuals steps detailed below:

1. A process on node A executes in a security domain with associated privileges (identity and transfer certificates) maintained by the local security manager. This process executes an `open` system call on a WebFS file stored on node B. The `open` system call is translated into a `NodeAccess` vnode operation in the WebFS partition (this is determined in the kernel by the path of the file being opened).

2. For simplicity, all of CRISIS functionality with respect to the file system is implemented in a user-level process, the WebFS server. The vnode layer makes an upcall to this server with a request to determine whether the process has access rights to the target file.

3. The WebFS server contacts the local security manager to determine the privileges associated with the calling process. For our UNIX implementation, the security manager maintains mappings between `uid/pid` pairs and security domains. These security domains, in turn, map to a set of identity and transfer certificates describing the process's privileges.

4. The WebFS server on node A determines that the "home" node for the target file is node B. The home node can make authoritative statements about the access rights required for files it is storing. Thus, the WebFS server on node A contacts the server on node B with a request to access the target file given the privileges possessed by the

calling process. The WebFS servers first mutually authenticate by presenting their own identity certificates over an SSL connection.

5. The WebFS server on node B contacts its local security manager to validate the transmitted certificates and to ensure that paths of trust exist from the local administrative domain to all domains identified in the transmitted certificates (first for the WebFS server on node A and then for the process on whose behalf the WebFS server made the request).

6. The WebFS server on node B consults the access control list for the local file if the requesting process possesses the required access privileges (e.g., read, write, or execute). As described in Section 6.3.5, entries in the access control list act as anchors. The transfer certificates describing the privileges of the process requesting the file must form a valid chain back to one of the entries in the ACL.

The result of the ACL check is transmitted as the return value to the original `open` system call in steps 7-9.

6.4.3 Running a Remote Job

The *resource manager* running on each WebOS machine is responsible for job requests from remote sites. Before executing any job, the resource manager authenticates the remote principal's identity and determines if the proper access rights are held. ACL's for remote execution contain the principals granted permission to execute programs on the local machine. To maintain local system integrity and to ensure that running processes do not interfere with one another, the resource manager creates a *virtual machine* for process

Operation	*NFS*	*WebFS*	*WebFS w/CRISIS*
Read 1 byte	3 ms	47 ms	55 ms
Write 1 byte	100 ms	289 ms	340 ms
Read 10 MB	9.8 s	11.0 s	12.2 s
Write 10 MB	9.2 s	12.8 s	14.0 s

Table 6.1: Performance overhead of adding CRISIS protocols to WebFS.

execution. These virtual machines interact with the CRISIS security system to enforce the rights restrictions associated with different security domains. Thus, processes will be granted variable access to local resources through the virtual machine depending on the privileges of the user originally responsible for creating the process.

Constructing virtual machines and performing resource allocation on running local processes are not contributions of this thesis. As described further in Chapter 8, we use Janus [Goldberg et al. 1996] and Java [Gosling & McGilton 1995] to create such virtual machines. The virtual machine abstraction also forms the basis for local resource allocation. On startup, a process's runtime priority is set using the System V `priocntl` system call, and `setrlimit` is used to set the maximum amount of memory and maximum CPU usage.

6.5 Performance

To quantify the performance impact introduced by CRISIS, we measured our global file system, WebFS, both with and without CRISIS enhancements. We measure the time required to read and write both 1 byte and 10 MB to a remote file. Measurements were taken between two Sun Ultrasparc 1's connected by a 10 Mb/s switched Ethernet.

Table 6.1 summarizes our results. The first column describes performance for accessing uncached NFS files. The second column describes access to uncached files through a version of WebFS without CRISIS modifications. The added overhead of WebFS relative to NFS is caused by kernel to user-level crossings for cache misses (WebFS network communication code is implemented at the user-level for ease of implementation and debugging). The third column describes performance of WebFS with CRISIS security enhancements. We believe the 10-20% slowdown relative to the baseline WebFS to be acceptable given the added functionality of access control checks and encrypted file transfer.

The measurements in the third column reflect the case where user credentials are cached on the remote node. An additional 175 ms overhead is introduced to establish an SSL connection and 230 ms are required to transfer and cache an identity plus a single transfer certificate if user credentials are not cached remotely. Once again, this total 400 ms overhead is a one-time cost incurred the first time a user makes any access to a remote site (WebFS maintains a "cache" of active SSL connections between machines to avoid the cost of re-establishing an SSL connection for each access). Finally, read access to a cached 1 byte file through WebFS with CRISIS enhancements takes 720 μs, and reading a cached 10 MB file takes 170 ms, values comparable to cached access through NFS. In summary, our security enhancements introduce significant overhead for initial and uncached access because of switching to a user-level process for communication and the overhead of establishing an SSL connection for transmission of certificates. However, the common case read access to a cached file stays entirely in the kernel and provides performance comparable to a file system such as NFS.

6.6 Related Work

CRISIS is loosely based on the DEC SRC security model [Lampson et al. 1991]. Relative to their work, one of our contributions is to simplify the model by using *transfer certificates* as the basis of fine-grained rights transfer across the wide area. Transfer certificates provide an intuitive model for both rights transfer and accountability, as they allow a complete description of the chain of reasoning associated with a transfer of rights. In addition, *revocation* is a first class CRISIS operation; even privileges described by transfer certificates (which are typically valid only for a limited period of time) can be revoked immediately. CRISIS also provides for explicit reasoning about the state of loosely synchronized clocks, an important consideration for wide-area applications. Further, CRISIS supports user-defined lightweight roles, to capture persistent collections of transferred rights (e.g., "Tom running a job on remote supercomputer"). Finally, in contrast to the DEC SRC work which was implemented in the kernel of a platform that is no longer available, CRISIS is designed to run portably across multiple platforms, a requirement for a wide-area security system to be useful in practice.

Kerberos [Steiner et al. 1988] is an authentication system for local-area networks. One approach to constructing a wide-area security system is to simply add fine-grained rights transfer to an existing authentication system such as Kerberos. While Kerberos has proven quite successful for local-area networks in a single administrative domain, it faces a number of challenges when extended to the wide area. First, Kerberos has no security redundancy: security is undermined if even a single authentication server or ticket granting server is compromised, allowing an adversary to impersonate any principal that

shares a secret with the compromised authentication server. In the wide area, the number of such single points of failure scales with the size of the Internet. Further, Kerberos requires synchronous communication with the ticket granting server in order to set up communication between a client and server; in the wide area, synchronous communication with a hierarchy of ticket granting servers is required. Given that the Internet today is slow, unreliable, and subject to frequent partitions, such communication can have a significant effect on availability and performance as perceived by the end-user. Although Kerberos servers could conceivably be replicated to improve availability, the servers would need to be geographically distributed to hide Internet partitions, providing more points of attack to an intruder.

SDSI [Rivest & Lampson 1996] is a distributed security infrastructure based on public keys with goals similar to our own. Their emphasis is on defining a standard format for certificates, rights transfer, and name spaces to provide a general security framework for Internet applications. With minimal extensions, SDSI could support CRISIS transfer certificates and remote execution of programs. Our work, however, is the largely orthogonal task of defining how such a framework can be used to provide redundant, high performance, and available security mechanisms for applications requiring secure remote control of wide-area resources.

Neuman [Neuman 1993] discusses distributed mechanisms for authorization and accounting. Neuman's work has much the same vision as our own, namely limited capabilities in addition to ACL's. His work proposes a more general capability model. However, the capabilities are not auditable because proxies do not carry a chain of transfers. Further,

Neuman's work is secret key as opposed to public key, meaning that synchronous communication is required for each transfer of rights. The trusted third party is responsible for recording transfers and transferring the end result. For example, if P_1 transfers rights to P_2, and P_2 further transfers rights to P_3, the trusted third party only passes on P_1 transferring rights to P_3 to any end reference monitors.

Jaeger and Prakash [Jaeger & Prakash 1995] present a model for discretionary access control in a wide-area environment. In their work, principals specify the subset of their privileges that are to be transferred to a script written by a potentially untrusted third party. The actual rights transferred are negotiated between the application writer and the user. In their system implementation in Taos [Wobber et al. 1993] (a secure OS based on [Lampson et al. 1991]), they add dynamic principals for running programs with some subset of a principal's privileges, observing the difficulty of creating temporary principals and updating all necessary ACLs with the new principal name. Their dynamic principals are similar to one of the applications of CRISIS transfer certificates.

6.7 Summary

In this chapter, we describe the architecture of CRISIS, a security system for wide-area applications. In designing CRISIS, we endeavored to systematically apply principles from related fields to increase system security, availability, and performance across the wide area. These principles include: redundancy, caching, local autonomy, least privilege, and complete accountability. We describe how these principles influence our design and detail the specific protocols used to securely carry out common operations across the wide area.

Another principal contribution of this work is the introduction of transfer certificates as a simple mechanism for the fine-grained transfer of rights across the wide area and the lightweight creation of roles.

Our security architecture meets the overall goals for a wide-area computation system as described in Chapter 2. CRISIS presents a *uniform interface to remote resources*, making authentication and authorization for local and remote resources identical. The system allows for *flexible* security policies. For instance, users who require maximum performance and are not overly concerned with security can endorse privilege transfers with long timeouts. Similarly, those with very sensitive data can opt for immediate rights revocation and fine-grained control over exactly what rights are transferred and to where they are transferred. CRISIS maintains a high level of *performance* by adding marginal overhead relative to remote access without any security guarantees (the goal of performance is relaxed somewhat with security since an insecure wide-area application is unusable in many cases). Finally, while CRISIS does require action on the part of end users for maximum security, CRISIS remains *backward compatible* with existing applications by, for example, imposing security at the file system interface.

Chapter 7

Rent-A-Server

7.1 Overview

The previous four chapters described the principal contributions of this dissertation. Active Names provide a programmable interface for locating and retrieving wide-area resources. WebFS is a wide-area file system that provides persistent cache coherent storage to distributed applications. Transparent Result Caching (TREC) allows for profiling of program execution to determine file dependencies. In conjunction with WebFS, one interesting application of TREC is caching dynamically generated Web content at both servers and proxies. CRISIS is the security infrastructure of WebOS, and it supports the fine-grained transfer of rights between multiple administrative domains. These four components make up the WebOS architecture, a uniform interface allowing distributed applications to take advantage of remotely programmable resources. A component of WebOS that is not described in this dissertation is remote execution. This final subsystem allows distributed applications to run programs on remote machines. Remote execution supports UNIX ap-

plications through sandboxing in Janus [Goldberg et al. 1996] and also supports execution of Java [Gosling & McGilton 1995] programs for platform independence.

While each of the pieces of this dissertation are individually interesting, together they provide a complete infrastructure for the development of wide-area applications. The goal of this dissertation is to provide uniform interfaces to global resources, implementing common abstractions for the requirements of distributed applications. A theme that runs throughout this work is the need for flexibility in supporting wide-area applications, e.g., programmability in naming, multiple cache coherence policies, and fine-grained support over rights transfer.

To demonstrate the utility of our individual contributions in supporting the development of wide-area applications, this chapter describes the design and implementation of Rent-A-Server, a Web server capable of dynamically replicating itself across the wide area in response to client access patterns. One goal of this chapter is to demonstrate how the implementation of Rent-A-Server is simplified by WebOS functionality. Further, initial performance evaluations of Rent-A-Server indicate that the system improves client latency and reduces consumed wide-area bandwidth, satisfying two principal goals of our framework as presented in Chapter 1.

7.2 Motivation

Rent-A-Server is a general model for graceful scaling across temporal and geographic spikes in client demand for a particular service. Our particular implementation focuses on Web service, and enables overloaded HTTP servers to shed load onto idle third-

party servers called *surrogates* that use the WebOS framework to coherently cache data from the primary server. The surrogate is able to satisfy the same HTTP requests as the original server, including requests for both static and dynamically generated objects (e.g., data pages versus CGI script results).

Rent-A-Server allows sites to deal with peak loads that are much higher than their average loads by "renting" hardware to deal with peaks. For example, the Internal Revenue Service site (`http://www.irs.ustreas.gov`) is overwhelmed by requests around April 15, but providing the computation power and network bandwidth necessary to handle peak levels of demand year round is a waste of resources. As another example, load for sites reporting election results in the United States moves across the country as polls close in individual geographic locations. The benefits for Rent-A-Server can be summarized as follows:

- *Geographic Locality*: In addition to distributing load, Rent-A-Server improves performance by increasing locality. Rather than satisfying requests from a centralized site, a system can distribute its requests across geographically distributed servers, each of which satisfies nearby clients.

- *Dynamic Reconfiguration*: The location and number of sites representing a service can be determined dynamically in response to client access patterns. Rent-A-Servers can be spawned to locations "near" current spikes in client demand, and torn down once client activity subsides.

- *Transparent End-to-End Availability*: Once a service is replicated with Rent-A-Server, users can transparently access whichever replica is available, routing around both

Internet and service failures.

- *Secure Coherent Data Access*: To address limitations associated with caching proxies that are unable to generate dynamic Web pages (e.g., results of cgi-bin programs) and often serve stale data, Rent-A-Server uses WebOS to provide authenticated, coherent global file access to data pages, CGI scripts, and internal server state needed by CGI scripts.

- *Safe Remote Execution*: Surrogate sites securely execute service programs and associated service scripts (such as CGI programs) without violating the surrogate's system integrity.

7.3 Current Approaches

Current efforts to distribute HTTP server load focus on either distributing load across a fixed set of machines maintained by the owner of the data or distributing data across (proxy) caches under client (not server) control. Many HTTP server implementations achieve scalability by replicating their data across a fixed set of servers at a single site and then using the Domain Name Service (DNS) to randomly distribute requests across the servers [Katz et al. 1994]. Unfortunately, this approach requires that each site purchase enough computing power and network bandwidth to satisfy peak demand.

"Mirror sites" are also used to improve locality and to distribute load, but this manual approach requires more effort to set up the mirrors and to maintain data consistency across the mirrors. Further, users must specify which mirror to use, which is both inconvenient and unlikely to yield a balanced load across sites. Finally, as with the approach of

running multiple servers at one site, mirror sites are allocated statically. The system must always maintain enough mirrors to deal with its peak loads, and the location of mirrors cannot be shifted to address shifts in geographic hotspots.

Another approach to distributing load, caching proxies, is used to reduce server load and to improve network locality. To use a proxy, groups of clients send all of their requests to their proxy machine. The proxy machine attempts to satisfy the requests from its local cache, sending the requests to the remote server if the cache cannot supply the data. If proxies satisfy many requests to the server through their caches, both server load and network congestion are reduced.

However, proxies are conceptually agents of Web clients rather than of Web servers. Thus, in some instances they provide a poor match to the requirements of overloaded services. First, proxy servers cache only data pages. A proxy must send all requests for CGI scripts to the original server. Second, because servers regard proxies as ordinary clients, the proxy can supply stale data to its clients because of the limitations of HTTP cache consistency protocols. Illustrating the importance of having server-controlled rather than client-controlled load distribution, some sites have recently asserted that proxy caches violate copyright laws by storing site content [Luotonen & Atlis 1994]. In effect, the proxies are reducing generated advertising revenues by hiding page access counts.

7.4 System Design

In this section, we demonstrate how the system services described in this dissertation simplifies the implementation of this application. The architecture of the Rent-A-Server

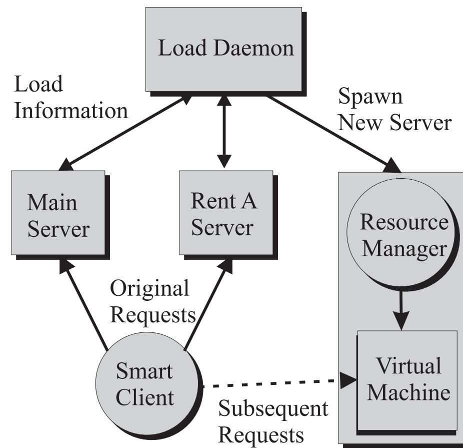


Figure 7.1: Rent-A-Server architecture.

is described in Figure 7.1. Clients use Active Names to access HTTP services. Periodically (currently every tenth response), servers piggy-back service state information in the HTTP reply header. This state information includes a list of all servers currently providing the service. The following information is included for each server: its geographic location, an estimate of its processing power, an estimate of current load, and a time period during which the server is guaranteed to be active. The last field is determined with short term leases that are periodically renewed if high demand persists. The short term leases prevent clients with stale state information from trying to access inactive surrogates (or worse, surrogates acting on behalf of a different service).

Each Rent-A-Server maintains information about client geographic locations (location is sent by Active Name requests as part of the HTTP request) and its own load information in the form of requests per second and bytes transmitted per second. Each Rent-A-Server periodically transmits this state information to a centralized *load daemon.*

For software engineering reasons, the load daemon is currently a separate process, however its functionality could be rolled into an elected member of the server group. The load daemon is responsible for determining the need to spawn or to tear down Rent-A-Servers based on current load information and client access patterns. It also transmits server group state (e.g., membership and load information) to each member of the server group, which is in turn piggy-backed by the servers to clients (using Active Names to mediate access to the service) as part of HTTP replies as described above.

Once the load daemon determines the need to spawn an additional server, it first determines a location for the new Rent-A-Server. The new server should be located close to any hotspots in client access patterns to both conserve bandwidth and to minimize client latency (this policy has not yet been implemented). Once the target machine is selected, the load daemon establishes an SSL channel with the surrogate's resource manager. The load daemon then creates the necessary transfer certificate to allow the surrogate to access the WebFS files containing the executables (e.g., HTTP server) or internal service state (e.g., CGI scripts or internal database). In addition, the load daemon provides a signed certificate (with an expiration date) granting the surrogate the right to serve data on behalf of the service. This certificate is transmitted to Active Name programs on demand to prevent spoofing of the service by malicious sites.

When setup negotiation is completed, the surrogate site builds a Janus virtual machine (this mechanism is described fully in Section 8.5) to execute the necessary programs (in our case an arbitrary HTTP server) to establish a service identity at the surrogate. The virtual machine ensures that the surrogate's system integrity is not violated by a buggy

executable or a malicious server. Both the service executable and any necessary service state are securely accessed and cached on demand through WebFS. The load daemon propagates the identity of the new surrogate to other members of the server group, which in turn transmit the identity and location of the new server to Active Name programs. Tear down of a surrogate is accomplished when client demand subsides and the load daemon decides not to renew leases with a surrogate. At this point, the load daemon revokes the transfer certificate allowing the surrogate to access service state information and execution of the virtual machine running the Web server is stopped.

7.5 Performance

To demonstrate the power of dynamic resource recruitment available from our approach, we present two sets of measurements of Rent-A-Server performance. The first set of measurements (described in Section 7.5.1 below) demonstrates the ability of the Rent-A-Server to dynamically recruit additional resources as the load placed on the service is increased. The second set of measurements (described in Section 7.5.2 below) demonstrates the utility of the system in reducing client latency and consumed wide-area bandwidth.

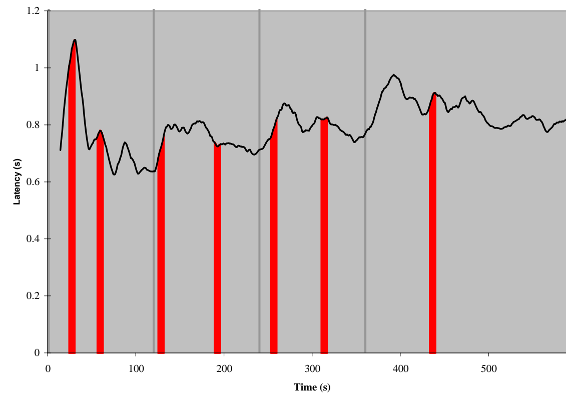
7.5.1 Dynamic Recruitment of Resources

The first set of experiments is conducted on a cluster of Sun Ultra Servers interconnected by 10 Mbps switched Ethernet. Seven Ultra Servers are designated as surrogates available to provide HTTP service on behalf of an eighth machine designated as the primary. All eight server machines run WebOS, including the file system and the resource manager

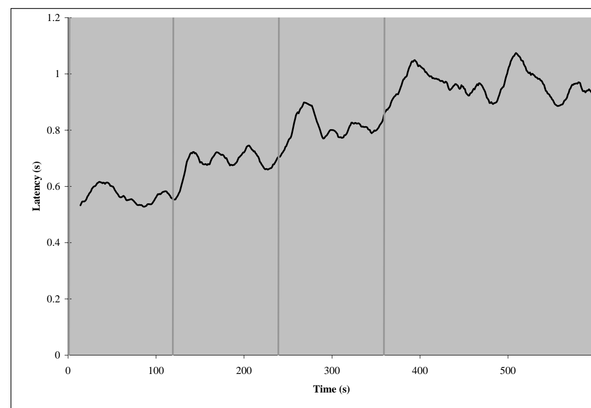
responsible for process control. Another 32 machines are used to generate gradually increasing load to the HTTP service. All machines are running Solaris 2.5.1, and Apache 1.2b6 is used for the HTTP server.

During the experiment, each client machine starts 15 client processes that continuously retrieve copies of the same 2.5 KB (a typical size today) HTML file. Active Name programs use random load balancing to pick from a changing list of available HTTP servers. To simulate increasing load, all 32 client machines do not begin making requests at the same time. Rather, clients start in four groups of eight machines each, with the start time of each group staggered by two minutes. Thus, all 32 clients are running after six minutes.

The results of our tests are summarized in Figure 7.2. The graphs plot average client-perceived latency in seconds as a function of elapsed time, also in seconds. Initially, only a single HTTP server is available; however, the load daemon spawns new servers onto available surrogates as service load increases. The thick dark columns correspond to execution of new HTTP servers, while the lighter thin lines correspond to the startup of a new group of eight client machines, with the first eight clients starting at time 0. For example, the Rent-A-Server graph shows that a third surrogate server was started at $t = 122$, shortly after the second group of eight clients start. The experiment reaches steady state at approximately $t = 500$ seconds after the last group of client machines start running at $t = 360$ seconds. The graph is truncated at $t = 600$ seconds. In summary, Figure 7.2(a) shows that Rent-A-Server is able to dynamically recruit resources in response to increased client demands. As clients increase server load over time, Rent-A-Server is able to dynamically recruit needed surrogates to maintain relatively steady quality of service,



(a) Rent-A-Server



(b) Fixed Server

Figure 7.2: Rent-A-Server performance with dynamic client load.

delivering 800 ms average latency.

While the number of active HTTP servers varies from between one and eight, on average 5.7 servers are active. To contrast the performance of Rent-A-Server with static server allocation, the experiment is executed with identical parameters, with the exception that 6 fixed servers are allocated for the duration. Figure 7.2(b) depicts the results. Between $t = 0$ and $t = 120$ with relatively light client demand, static allocation outperforms Rent-A-Server, delivering an average latency of approximately 600 ms. However, it can be argued

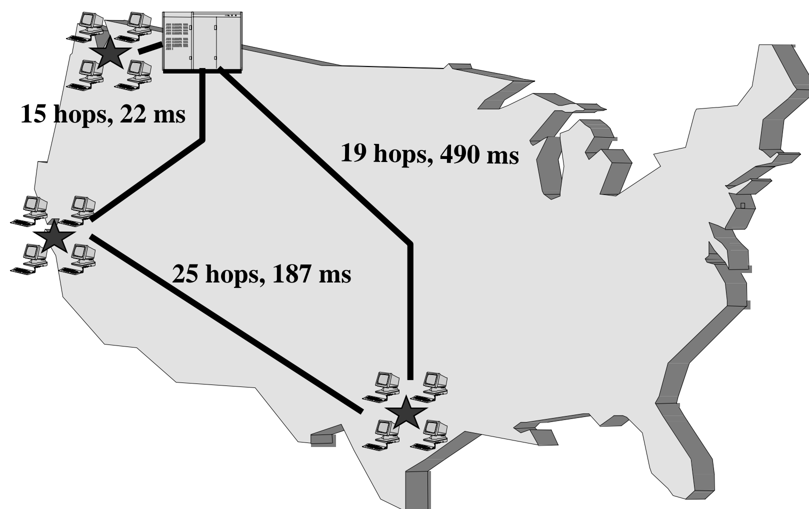


Figure 7.3: Rent-A-Server wide-area experimental setup.

that resources are wasted because measurements indicate that three servers could deliver the same performance for eight clients. When all 32 clients are running between $t = 360$ and $t = 600$, Rent-A-Server's ability to dynamically recruit resources results in improved performance. During this time period, clients see 850 ms average latency while the statically allocated resources become constrained, delivering 965 ms average latency. Thus, Rent-A-Server outperforms static resource allocation even when the average amount of consumed service resources stays constant.

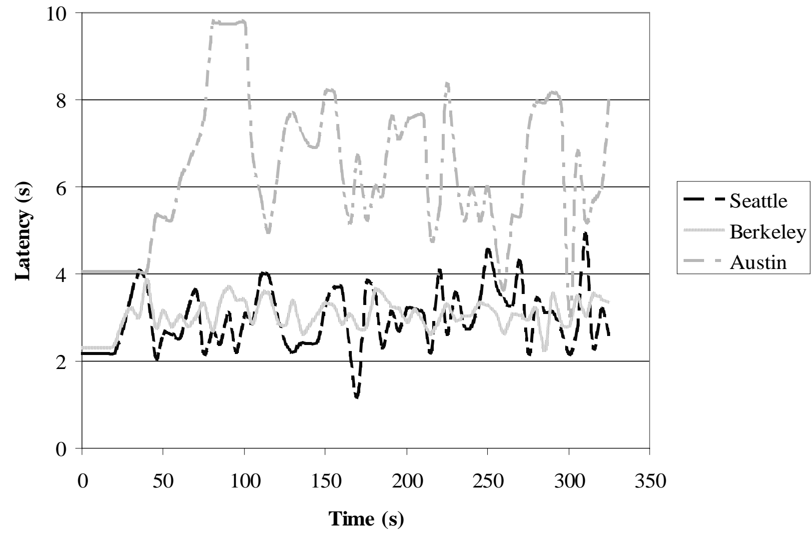
7.5.2 Reduced Wide-Area Latency and Bandwidth

The previous set of experiments demonstrates Rent-A-Server's ability to adapt to varying loads in a local setting. In addition, we conducted a second set of experiments to demonstrate the potential utility of Rent-A-Server to improve client latency and reduce con-

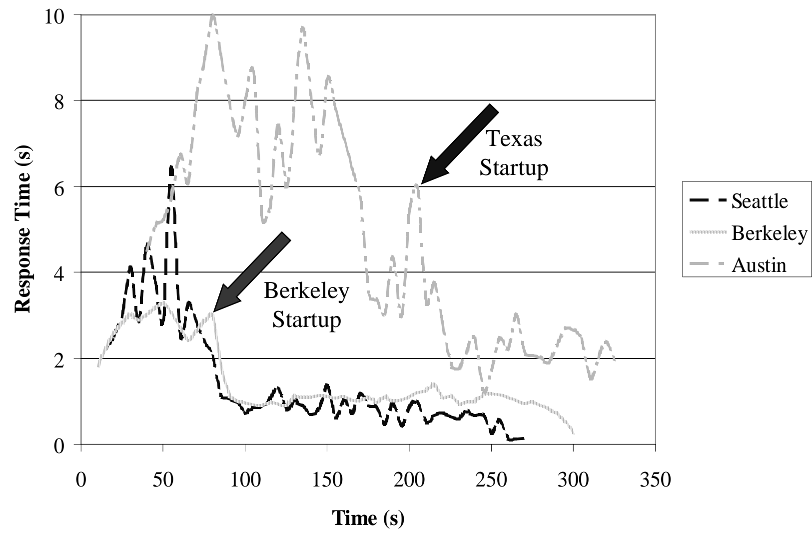
sumed bandwidth in a wide-area setting. While the system is not yet ready for production deployment, the measurements in this section suggest that further refinement of this model can potentially lead to improved wide-area Web service. The experiments are conducted across the wide area as depicted in Figure 7.3. Eight sun Ultra workstations at each of Seattle, Berkeley, and Austin act as clients of a Web service. Each client continuously requests the same 1 KB HTML file. Initially, there is a single HTTP server located in Seattle running Apache 1.2b6 on a Sun Ultra workstation. As described below, two surrogate Sun Ultra's are available at Berkeley and Austin to demonstrate the utility of Rent-A-Server. All the machines run Solaris 2.5.1.

Figure 7.3 also depicts the relative connectivity of the three sites, as measured by `traceroute` and `ping` on a weekend night. The reported numbers demonstrate best-case connectivity information. As shown in the figure, connectivity between Berkeley and Seattle is quite good, with only 22 ms round trip latencies reported by ping. Packets traveling from Berkeley to Austin have 187 ms latency, with approximately 2% of the packets dropped. Connectivity between Seattle and Austin is quite poor, with 490 ms latency and 20% of packets dropped.

During the experiment, each client machine starts 8 clients processes that continuously retrieve copies of the same 1 KB HTML file. The results of our tests are summarized in Figure 7.4. The graphs plot average client-perceived latency in seconds as a function of elapsed time, also in seconds. Figure 7.4(a) shows performance for the case where only a single server is available in Seattle. The graph shows that performance for clients at Berkeley and Seattle is quite poor, averaging approximately 3 seconds to retrieve the 1 KB



(a) Fixed Server



(b) Rent-A-Server

Figure 7.4: Rent-A-Server wide-area performance: benefits of dynamic replication.

HTML file from the Seattle server. Clients at Austin suffer from even worse performance, widely varying in average latency between 4 and 10 seconds. The poor performance of the Berkeley and Seattle results from an overloading of the single HTTP server in Seattle. The performance for the Austin clients relative to the Berkeley and Seattle clients is explained by the poor network connectivity between Austin and the HTTP server located in Seattle.

Figure 7.4(b) shows the improved performance available from using Rent-A-Server. In this case, approximately 90 seconds into the experiment, Rent-A-Server's load daemon spawns off an additional server at Berkeley. At this point, latency for both the Berkeley and Seattle improves into the .75 second latency range. Latency for the Texas clients is still poor because of the poor network connectivity between Texas and both Seattle and Berkeley. Thus, 200 seconds into the experiment a third server is spawned at Texas, with a corresponding improvement in latency for the Texas clients. What is not shown on this graph is the corresponding savings in wide-area bandwidth as clients at Berkeley and Seattle fall back to local servers as opposed to traversing wide-area links to reach the Seattle server.

The performance of Rent-A-Server demonstrates the power of dynamically recruiting resources for wide-area services. However, it is equally important to provide a convenient interface for application development. Our implementation of Rent-A-Server in WebOS consists solely of the load daemon and additions to the Apache HTTP server to transmit state information to the load daemon and to transmit aggregate service state (in HTTP headers) to clients. The load daemon consists of 1000 lines of C++ code, and we added 150 lines of C code to Apache. Beginning with the WebOS framework, our prototype of Rent-A-Server was operational in less than one week.

7.6 Summary

Earlier chapters in this dissertation described how one of our individual contributions simplifies the implementation of a wide-area application, such as mobile distillation in Chapter 3 and dynamic Web object caching in Chapter 5. In this chapter, we described how the contributions and techniques described in Chapters 3-6, while individually interesting, can be brought together to enable and simplify the implementation of emerging wide-area applications. We demonstrate this functionality through the Rent-A-Server application because it exercises each of the requirements of providing transparent access to remote computational resources described in Chapter 1: naming, persistent storage, security, and remote execution. Quantitatively, we show that Rent-A-Server reduces consumed wide-area bandwidth for busy Web services and simultaneously reduces client-perceived latency. Qualitatively, we demonstrate that the system support provided by our research significantly simplifies the implementation and deployment of Rent-A-Server.

Chapter 8

Related Work

This chapter describes some of the prior research related to this dissertation, focusing on other efforts to support the development of wide-area applications. We begin with a discussion of the relationship between this work and earlier efforts into distributed operating systems and cluster computing. Earlier chapters contain discussions of related work in specific sub-areas.

8.1 Distributed Operating Systems

Operating systems were originally developed to provide a set of common system services, such as I/O, communication, and persistent storage, to simplify application programming. With the advent of multiprogramming, this charter expanded to include abstracting shared resources so that they were as easy to use (and sometimes easier) as dedicated physical resources. The introduction of local-area networks in the 1980's expanded this role even further. A goal of network operating systems such as Locus [Popek

et al. 1981], Mach [Accetta et al. 1986], Sprite [Nelson et al. 1988], Amoeba [Mullender et al. 1990], and V [Cheriton 1988] was to make remote resources over the LAN as easy to use as local resources, in the process simplifying the development of distributed applications. For example, distributed operating systems provide abstractions such as communication channels and processes, as opposed to networks and processors. Contributions of distributed operating systems include naming [Accetta et al. 1986], distributed shared memory [Leach et al. 1983, Li & Hudak 1989, Carter et al. 1991] and message-based location transparency [Draves et al. 1989]. With analogy to these systems, we argue that it is time to provide transparent programmable access to wide-area resources, in effect to make wide-area resources as easy to use as those on a LAN.

8.2 Cluster Computing

This dissertation work grew out of earlier work on the Berkeley Network of Workstations (NOW) project [Anderson et al. 1995a]. A NOW consists of a cluster of commodity workstations with emerging high-speed interconnects [Boden et al. 1995]. The goal of the NOW project was to transparently support both parallel and sequential workloads. NOW applications can leverage Active Messages [von Eicken et al. 1992] for high-bandwidth, low-latency communication. A scalable and serverless file system, xFS [Anderson et al. 1995b], was designed to leverage aggregate cluster disk bandwidth and storage. And finally, GLUnix [Vahdat et al. 1994, Ghormley et al. 1998a] implements scheduling, resource allocation, and job control. The goal of GLUnix is to provide the illusion that a single powerful machine with a unified interface to cluster resources is available to all users.

As the NOW project became functional, we received a number of requests from within the campus community for access to NOW resources. We observed that a unified interface for accessing remote resources would be generally beneficial (e.g., for accessing other remote resources such as supercomputers). With analogy to the NOW project, we began our WebOS efforts with the goal of exporting traditional operating services such as naming, persistence, security, and remote execution (each of which had an analog in the NOW) to wide-area applications. In building on cluster research, one goal of this dissertation is to identify areas where traditional distributed system abstractions break down as the system is scaled to the wide area and to provide a testbed for exploring possible solutions.

8.3 Global Computation

The space of issues and potential problems in providing system support for wide-area applications is very large. Our efforts represent a point in the space of possible solutions and issues. A number of concurrent efforts also seek to exploit computational resources available on the Internet for wide-area parallel programming, including Wax [Stout 1994], Legion [Grimshaw et al. 1997], Atlas [Baldeschwieler et al. 1996], NetSolve [Casanova & Dongarra 1996], GOST [Neuman 1998], and Globus [Foster & Kesselman 1997]. Our research work shares a need for similar underlying technology with these systems (such as the need for a global namespace and persistent storage). However, the problem domains and contributions of these related research efforts are largely complimentary. This section describes the relationship between our research and a number of related efforts into supporting wide-area applications.

8.3.1 GOST

The Global Operating System Technology (GOST) project provides a middleware layer facilitating access to Internet resources in multiple formats. GOST is made up of a number of cooperating system components. The Prospero naming system [Neuman 1992a] supports a programmable infrastructure for integrating multiple views of wide-area resources. For example, users can create their own customized namespaces using filters. The Prospero Naming system is related to Active Names, as both systems provide programmability. However, the semantics of Prospero filters are more limited, with execution limited to the client machine. Further, Prospero does not provide a general infrastructure for location-independent invocation of naming programs (e.g., with respect to resource management and security enforcement). The Prospero File System [Neuman 1992b] leverages naming in Prospero to provide a user-configurable view over available wide-area resources. The file system layers a reliable transport protocol to distribute files available through various underlying protocols (e.g., AFS, NFS, or FTP). The Prospero File System provides functionality similar to WebFS. However, WebFS focuses on flexible support for cache coherence protocols in support of wide-area applications. GOST also provides a resource manager [Neuman & Rao 1994] to facilitate access to remote resources.

8.3.2 Globe

Globe [van Steen et al. 1997] shares a number of goals with this dissertation. The system has the following major design goals: "(1) provide a uniform model for distributed computing, (2) support a flexible implementation framework, and (3) ensure worldwide scal-

ability" [van Steen et al. 1997]. Globe addresses these problems by exporting a distributed shared object system with the goal of scaling to one billion users and one trillion objects. Each Globe object is made up of subobjects for: control, replication, security, communication, and semantics. The control subobject handles such tasks as method invocation and parameter marshaling and unmarshaling. The replication object implements algorithms for maintaining consistency among replicas. The communication object is responsible for communication between parts of distributed objects that are located on different machines. The security object implements access control rights on a per-object basis. Finally, the semantics subobject implements most of the application-specific functionality of the object. Users typically build objects by providing the functionality of the semantics subobject and then choosing the other subobjects based on their application requirements (e.g., a particular replication strategy).

8.3.3 Globus

Globus [Foster & Kesselman 1997] provides a vertically integrated infrastructure supporting high performance parallel programs using unique resources distributed across the wide area. For example, multiple supercomputers, clusters of workstations, visualization tools, and data gathering tools (such as a satellite downlink) can be brought to bear on large problems that would be difficult and/or time consuming to solve with even very powerful machines at a single site. Globus aims to provide the illusion of a virtual supercomputer available to single applications, with actual components of the computational system at multiple sites and spread between multiple administrative domains.

Globus system components include the Metacomputing Directory Service [Fitzger-

ald et al. 1997] that allows applications to locate remote resources and maintain information about Globus components. Nexus [Foster et al. 1997] supports transparent multi-protocol communication between processes. The Nexus library exports a unified communication interface but chooses the most efficient communication protocol depending on the location of the communicating endpoints. For example, two processes on the same LAN might use Active Messages [von Eicken et al. 1992] while processes communicating across the wide area might use TCP sockets. The Globus Security Infrastructure [Foster et al. 1998] allows for authentication and access control between multiple administrative domains.

8.3.4 Legion

Legion [Grimshaw et al. 1997] is an object-oriented approach toward supporting the notion of a virtual supercomputer. Legion shares a number of goals with Globus, while its underlying implementation goals (scalable to trillions of world-wide objects) is similar to Globe. Legion also stresses flexibility and site autonomy, observing that no single policy, whether with respect to cache coherence, naming, etc., will satisfy the requirements of all users or administrative domains. Legion is implemented at the user level, with the following requirements: no change to the operating system, no change to the interconnect, and no privileged access to local resources. Programming of available resources is facilitated through Mentat, an extension to the C++ programming language. Legion also exports a security model [Wulf et al. 1995] with simple methods available on all objects that allow for the composition of arbitrary policy.

8.3.5 Summary

The efforts described in this section and our own work share a number of common goals, with flexibility and user-control over system semantics appearing prominently in all the research efforts. However, there are key differences in target applications and specific research contributions. The first system described, GOST, is directed more toward flexible access to wide-area data repositories (in multiple formats) similar to functionality provided by the World Wide Web [Berners-Lee et al. 1992]. The latter three efforts, Globe, Globus, and Legion, have a different application focus from our own work. These efforts focus on system support for high performance parallel applications, while we focus on rethinking all aspects of distributed systems to support transparent access to remotely programmable resources. For example, a wide-area parallel particle simulation will have very different security and coherency requirements from dynamically migrating Internet services. While a convergence in the system services provided by our research and the other efforts described in this section is likely, the differences in target applications shape the focus of the various research efforts, the resulting system design, and the individual contributions. Specific comparisons between our work and these research projects in specific sub-areas are described in Sections 3.4, 4.5, and 6.6.

8.4 Scalable Internet Services

In addition to the efforts supporting wide-area applications described above, a number of research projects focus on building scalable network services, such as video gateways or Web servers. These efforts are related to our own efforts, for example, in

building a Web server able to replicate itself across the Internet in response to client access patterns (as described in Chapter 7).

8.4.1 Active Networks

Active Networks propose to modify Internet routers to become dynamically programmable, either at the connection or packet level [Tennenhouse & Wetherall 1996, Wetherall et al. 1998]. The goal is to make it easier to extend network protocols to provide new services, for example, to minimize network bandwidth consumed by multicast video streams. As in our work, a major motivation is to move computation into the Internet to minimize network latency and congestion. The two efforts are complementary in the sense that they export remote access at different points in the wide-area network. Our efforts allow programmable extensions to run at end hosts, while Active Networks advocates placing computation within the network itself (e.g., in routers). Our research can be viewed as a logical extension of Active Networks, where the active computing elements in the Internet can be servers in addition to the individual processors inside of routers operating on packet streams. In fact, as described in Chapter 3, our own design and deployment of Active Names shares motivation with Active Networks, but interposes programmability at the naming interface as opposed to the packet level.

8.4.2 TACC

TACC (Transformation, Aggregation, Caching, and Customization) [Fox et al. 1997] provides a framework that allows applications to leverage replicated cluster resources (such as memory, CPU, disk) to build highly scalable Internet services. These applica-

tions must typically display loose consistency semantics to simplify replica updates. Thus, applications cannot rely on all replicas of a data repository to be updated synchronously and must be able to properly function with potentially stale information. For these applications, TACC provides a toolkit for constructing high performance, scalable, and fault tolerant applications on a cluster of workstations [Anderson et al. 1995a]. Applications of TACC include dynamic distillation [Fox et al. 1996] and the Inktomi Search Engine [Brewer & Gauthier 1995]. Dynamic distillation serves as the motivation for our research into mobile distillation described in Chapter 3. In general, TACC is orthogonal and complementary to our investigation of transparent access to remotely programmable resources. For example, while we focus on scalable services across the wide area, we can leverage TACC techniques for scalability on a cluster of workstations. Similarly, our efforts into programmable naming can simplify the process of locating multiple versions of the same object in dynamic distillation.

8.4.3 Active Services

Elan Amir, et. al. [Amir et al. 1998] propose an alternative to Active Networks for deploying higher level Internet services. Active Services leverage clusters of workstations and multicast technology to deploy client-specified *servents* at multiple points in the network. Servents are arbitrary code that perform some function on behalf of clients; the authors implement a video gateway servent to perform transcoding of video to client capacity. Active Services support a number of the same application domains as Active Networks, without any changes to the underlying routers. With respect to this dissertation, Active Services are most closely related to our own research in extensible naming of wide-area resources (see

Chapter 3). Active Services proposes a new interface for deploying client-specific extensions across the network, while we interpose on an existing interface, naming. Active Services make no provisions for safety, as arbitrary Tcl code can be run on remote nodes. We utilize Java and virtual machines to provide guarantees about the integrity of remote machines. Further, we propose techniques to allow client-specified code to run at arbitrary points in the network. Once a servent is instantiated, it typically runs at a single point in the network. Similarly, experience with Active Services are limited to clients contacting a single cluster. It is unclear how Active Services will scale to the wide area, e.g., when servents can migrate between administrative domains.

8.5 Remote Computation

The contributions described in this dissertation focus on supporting transparent access to remote resources. As discussed in Chapter 1, the four principal components of such access are naming, persistent storage, security, and safe remote code execution. Chapters 3-6 describe how we address the problems of naming, persistent storage, and security in the wide area. While this dissertation does not make a specific new contribution to the topic of safe remote code execution, such functionality is critical for providing a complete interface to remote resources (see, for example, the description of the Rent-A-Server application in Chapter 7). This section describes the existing techniques leveraged to integrate remote execution into the WebOS framework.

The goal of remote execution in the context of our dissertation work is to make execution of processes on remote nodes as simple as forking a process on a local proces-

sor. As with the local case, our process control model must address issues of safety and fairness. On local machines, safety is provided by execution in a separate address space, while fair allocation of resources is accomplished through local operating system scheduling mechanisms.

We execute a *resource manager* on each WebOS host to manage job requests from remote sites. Before executing any job, the resource manager authenticates the remote principal's identity and determines if the proper access rights are held. To maintain local system integrity and to ensure that running processes do not interfere with one another, the resource manager creates a *virtual machine* for process execution. These virtual machines interact with the CRISIS security system to enforce rights restriction associated with different security domains. Thus, processes will be granted variable access to local resources based on the privileges of the principal originally responsible for creating the process.

We leverage Janus [Goldberg et al. 1996] to manage virtual machines for UNIX programs. Processes in the virtual machine execute with limited privileges, preventing them from interfering with the operation of processes in other virtual machines. Similar to our approach for TREC (Chapter 5), Janus uses the Solaris `/proc` file system to intercept the subset of system calls that could potentially violate system integrity, forcing failure if a dangerous operation is attempted. The set of intercepted system calls is sufficiently small to minimize overhead for typical applications [Goldberg et al. 1996]. A Janus configuration script determines access rights to the local file system, network, and devices. These configuration scripts are set by the local system administrator on a per-principal basis.

To execute architecture-independent programs, we utilize Java [Gosling & McGilton

1995]. Similar to Janus, Java allows for the specification of virtual machines with the set of privileges available for the execution of each program within the Java Virtual Machine. Before executing a Java program, the WebOS resource manager creates a new security manager that describes a target program's privileges. Elements of both the Java Virtual Machine and the programming language ensure that these pre-defined security constraints are not violated as the program executes. These Java system features are described elsewhere [Wallach et al. 1997, Wallach & Felten 1998].

In summary, we leverage existing techniques for supporting safe remote process execution to provide a complete interface to global network resources. As described in Chapter 6, the CRISIS security model integrates with remote execution to allow for restricted and authenticated access to wide-area computational resources (for example, only a small number of users can run programs on a certain supercomputer). Further, remote computation is essential to support a number of our target wide-area applications, including the Rent-A-Server application described in Chapter 7. Remote execution will also provide a convenient substrate for future experiments exploring, for example, resource allocation across the wide area or the proper user interface for executing remote programs (as described in the next chapter).

8.6 Summary

This chapter described related research in the area of wide-area computational services, contrasting the contributions of concurrent and prior work with our own. This dissertation benefits from related research efforts in a number of ways. First, other projects

provide a starting point for posing the outstanding and interesting unanswered questions in this area. Similarly, by defining the space of potential solutions, it becomes easier to identify the hard issues and to pose the right questions. Next, related research validates the importance of and interest in a given research area. Finally, concurrent related research provides a strong stimulus for pushing forward with our own work.

Chapter 9

Conclusions and Future Work

We close this thesis by summarizing our contributions. First, we describe the lessons learned from the design, implementation, and deployment of our techniques and the applications we built on the infrastructure. A number of interesting avenues for future research are then described leading to a brief summary of our research.

9.1 Contributions

The thesis of this dissertation is that remotely programmable network elements allow for a restructuring of wide-area systems that will improve wide-area resource utilization, simplify application development, and improve end-to-end performance. This dissertation proposes a number of techniques for providing transparent access to remotely programmable resources and conducts a number of experiments to validate these techniques. Specifically, we address wide-area naming, persistent storage, and security. Flexibility cuts across all of our solutions because the heterogeneity of the wide-area network, its clients, and services

dictates that a single policy cannot be appropriate for all applications in all situations. We make four primary contributions to support this dissertation:

1. We motivate the need for programmability in wide-area naming systems. We demonstrate that existing techniques for locating wide-area resources cannot provide optimal performance for all applications. This mismatch naturally argues for flexibly matching resource location techniques to application semantics. To deploy such flexibility, we introduce *Active Names*, application-specific and location-independent programs used to locate and retrieve wide-area resources. We use application studies to demonstrate both intuitive semantics and improved performance from the use of Active Names.

2. We demonstrate the value of combining communication and persistence in a location-independent storage system. Because wide-area applications can often trade coherency guarantees for increased performance and availability, our solution supports flexible, programmable cache coherence protocols. We show that an increasing number of wide-area resources are dynamically generated and that it is possible to automatically cache the results of some of these programs as normal files in the persistent store. Our results show that such caching significantly improves Web server performance when results are dynamically generated.

3. We devise techniques enabling the construction of a wide-area security system with the following properties: high performance and availability despite network limitations, fine-grained control over remotely programmable resources, and rights transfer and revocation between multiple administrative domains. Existing security systems do not work well in a wide-area, agent-based (i.e., mobile computation) environment be-

cause of invalid assumptions such as central control, homogeneous hosts, and highly reliable/high performance networks. We design and implement a security system, CRISIS, that improves upon existing systems by accounting for the unique requirements of wide-area applications.

4. We show that our individual contributions can be brought together to enable and simplify the development of a number of wide-area applications, including a Web server capable of dynamically replicating itself across the wide-area in response to client access patterns. Our experience qualitatively shows that application development is significantly simplified by our techniques for programmable access to remote resources. Further, our experiments quantitatively demonstrate improved utilization of network resources (e.g., bandwidth) and improved end-to-end system performance relative to traditional centralized approaches.

9.2 Future Work

This dissertation demonstrates that taking advantage of remotely programmable network resources for wide-area applications requires a rethinking of all aspects of distributed services. However, we have also uncovered a number of interesting avenues for future research. This section summarizes research issues to be addressed for solidifying our understanding of the role of system support for wide-area applications.

9.2.1 Scalability and Fault Tolerance

A number of efforts focus on building scalable and fault tolerant systems on local area clusters [Anderson et al. 1995a, Fox et al. 1997, Amir et al. 1998]. In the context of wide-area systems a number of interesting scalability questions remain outstanding. For example, the Online Agents responsible for refreshing privileges (described in Chapter 6) may become a bottleneck within a given administrative domain. Similarly, if an Online Agent fails or becomes unavailable, certificate endorsement becomes impossible. It should be fairly straightforward to replicate online agents within a single administrative domain for improved performance and fault tolerance. However, addressing network partitions where all machines within an administrative domain become inaccessible is an open question in fault tolerance. Similar issues of scalability and fault tolerance arise with the cache coherence techniques described in Chapter 4. If a centralized node responsible for transmitting invalidates becomes unavailable, caching hosts may be left with stale versions of files.

9.2.2 Performance Isolation

When deploying services across a system as heterogeneous and complex as the Internet, it can be difficult to ascertain the source of performance problems. For example, when clients receive slow responses from a service it can be difficult to determine if the problem lies with congestion in local-area networks or across backbone links. Further, the problem could have nothing to do with the network, resulting from performance limitations of either the client or service host. It is important to determine the cause of service slowdowns for services such as Rent-A-Server (described in Chapter 7). Replicating a service

across the Internet because of service CPU limitations may not make the most sense. In such cases, employing clustering technology on the local area is likely to be much more effective in improving service performance [Anderson et al. 1995a, Fox et al. 1997]. Developing tools to monitor service performance and to isolate individual components of service performance is an avenue of future work.

9.2.3 Simulation

One technique for studying scalability, fault tolerance, and performance isolation is large-scale simulation of wide-area services. Simulation allows for close inspection of various issues while varying assumptions made about the simulation environment, such as network topology, client access patterns, and host performance parameters. Simulation also benefits the proper design of wide-area applications, such as Rent-A-Server. For example, simulation can determine the proper algorithm for recruiting surrogate hosts across the wide area in response to client demands. However, simulating a network as wide-scale and complex as the Internet remains an open problem [Paxson & Floyd 1997]. Even generating realistic Internet topologies can be challenging [Calvert et al. 1997, Zegura et al. 1997]. Further, modeling the interactions among Internet protocols can be quite slow and has only been successful for small-scale networks to date [McCanne et al. 1996].

9.2.4 Application Studies

In the context of the Rent-A-Server application, robust algorithms are necessary to determine when wide-area surrogates should be recruited and freed. Currently, the load daemon uses static thresholds of load at a fixed percentage of available replicas to determine

when to spawn additional replicas. A similar methodology is used to tear down replicas when load subsides. The fundamental problem is that the daemon uses load as perceived by servers to make replication decisions. Ideally, this decision should be based upon the performance as perceived by clients. While this information is more difficult to gather (though a system such as Spand [Seshan et al. 1997] can facilitate such a process), it will allow for more intelligent replication decisions, as well as the ability to choose geographical sites likely to deliver the largest boost in performance. For example, the system currently randomly chooses among a list of available surrogate sites to spawn additional replicas because it is unaware of the existence of any geographical hotspots. By using client information, the system can leverage locality in the client population to replicate servers at optimal points in the network, further minimizing latency and consumed wide-area bandwidth (because, generally, clients will travel fewer hops before reaching the "nearest" replica).

Also in the context of Rent-A-Server, more realistic workloads should be employed to evaluate system performance. Server workloads can generally be broken down into two categories: requests for static files of varying sizes and requests for dynamically generated data that takes varying amounts of computation power to produce. By mixing both types of workloads, bandwidth, latency, as well as computation power can be accounted for. Current Internet characteristics show that latency and bandwidth are often unrelated along the same link. Similarly, available server computation power is independent of bandwidth and latency. Thus, it may be faster to go to one site for small files, a second site for larger files, and a third site for objects that are computationally expensive to produce.

9.2.5 Resource Allocation

Resource allocation for wide-area resources is another open avenue for future research. Locally, we envision multiple services competing for available resources on a local-area network (for example, Rent-A-Server surrogates, cooperative caches, and distillation engines). Mechanisms will be necessary to ensure fair allocation of CPU, network, memory, and disk resources (one approach to such allocation is currently being studied in the Sim-Millenium project [Culler 1998]). Studies can also be conducted to ascertain the definition of "fair" allocation, e.g., equal resources to all processes versus a cost-based model where the share of resources is based on how much the user pays. Further, users in the local administrative domain may be given absolute priority over jobs running from remote sites.

These resource allocation problems become more complicated in the global context. Mechanisms are necessary to ensure that, for example, a single user is not able to utilize 10% of all computational resources available across the wide area. Further, incentives must be available to incite users to make their resources available for global computation.

9.2.6 Security

A number of interesting issues remain outstanding in the context of security and authentication. For example, it remains unclear whether credentials should be "pushed" to reference monitors along with a request or whether the reference monitor should request the "pulling" of credentials after a request has been made. Each approach has its own set of tradeoffs. Pushing credentials means that the requester must know exactly what set of rights are required to access a particular resource (infeasible in the general case) or

that the requester must push all of its credentials with every request (perhaps revealing more information than necessary to reference monitors). Pulling credentials brings up the opposite set of considerations because the reference monitor will reveal to each requester the exact set of credentials needed to access a particular resource. It is likely that a hybrid push/pull approach can function well, allowing both the requester and reference monitor to minimize the amount of information divulged.

The proper user interface for allowing end users to reason about security issues also remains an open question. For example, it is unreasonable to force users to determine all the privileges a job requires to run remotely (e.g., all files it must be able to access). Automatically profiling job execution to determine a baseline set of privileges that a job needs will greatly reduce the burden on end users. A starting point for performing such profiling is the TREC infrastructure discussed in Chapter 5. Furthermore, allowing running jobs to request additional privileges (perhaps requiring user intervention) when the need is discovered (e.g., a particular sensitive privilege is not required for most runs of a program) will also improve system usability. Proper ways of communicating such requirements to unsophisticated end users is another difficult issue. Unless the exact nature and sensitivity of a desired resource can be communicated, users are likely to become frustrated and answer in the affirmative to all requests for additional privileges (as is often the case with security alerts provided by popular browsers such as Netscape Navigator and Internet Explorer).

9.3 Summary

One promise of the Internet is providing high performance services to a broad range of clients. Traditionally, such services are implemented at a central site in the wide-area network. Unfortunately, heterogeneity in the underlying network and bursty client access patterns mean that a central server cannot offer optimal performance and availability to end clients. This dissertation explores rethinking distributed services by providing seamless access to remotely programmable network resources. Such access allows for the construction of services that are able to dynamically migrate and replicate in response to client access patterns. We apply this thinking to all aspects of distributed services, including naming, persistent storage, security, and authentication. The benefits of these techniques include services that display improved end-to-end availability, improved cost performance, and improved burst behavior.

At a high level, building high performance wide-area services has three principal requirements: i) scalable and fault tolerant clusters individually exporting the service, ii) access to remotely programmable resources enabling virtual services able to run at any point in the network, and iii) a reliable and high performance wide-area network. Our research grew out of work on the NOW project [Anderson et al. 1995a], where the goal was to support the first requirement above. This dissertation describes contributions supporting the second requirement, transparent access to remotely programmable resources. Finally, as described in this and previous chapters, our work raises a number of interesting questions for future work in networking research, the third requirement above.

Bibliography

[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.

[Accetta et al. 1986] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. "Mach: A New Kernel Foundation For UNIX Development". In *Proceedings of the 1986 USENIX Summer Conference*, pp. 93–112, June 1986.

[Alexandrov et al. 1997] A. D. Alexandrov, M. Ibel, K. E. Schauser, and C. J. Scheiman. "UFO: A Personal Global File System Based on User-Level Extensions to the Operating System". In *Proceedings of the 1997 USENIX Technical Conference*, Anaheim, CA, January 1997.

[Alvisi & Marzullo 1996] L. Alvisi and K. Marzullo. "Tradeoffs in Implementing Optimal Message Logging Protocols". In *Proceedings of the Fifteenth Symposium on Principles of Distributed Computing*, June 1996.

[Amir et al. 1998] E. Amir, S. McCanne, and R. Katz. "An Active Service Framework and its Application to Real-Time Multimedia Transcoding". In *Proceedings of*

SIGCOMM, September 1998.

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

[Anderson et al. 1995b] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. "Serverless Network File Systems". In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pp. 109–126, December 1995.

[Apa 1995] *Apache HTTP Server Project*, 1995. `http://www.apache.org/`.

[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.

[Baker et al. 1991] M. Baker, J. Hartman, M. Kupfer, K. Shirriff, and J. Ousterhout. "Measurements of a Distributed File System". In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pp. 198–212, October 1991.

[Baldeschwieler et al. 1996] E. Baldeschwieler, R. Blumofe, and E. Brewer. "Atlas: An Infrastructure for Global Computing". In *Proc. of the Seventh ACM SIGOPS European Workshop: Systems Support for Worldwide Applcations*, September 1996.

[Belani 1998] E. Belani. "A New Model for Wide Area Security". Master's thesis, U.C. Berkeley, May 1998.

[Berners-Lee et al. 1992] T. Berners-Lee, R. Cailliau, J.-F. Groff, and B. Pollermann. "World Wide Web: The Information Universe". In *Electronic Network: Research, Applications, and Policy*, number 1 in 2, Spring 1992.

[Bershad & Pinkerton 1988] B. N. Bershad and C. B. Pinkerton. "Watchdogs—Extending the UNIX File System". *Computing Systems*, 1(2):169–188, Spring 1988.

[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.

[Birrell et al. 1986] A. Birrell, B. Lampson, R. Needham, and M.Schroeder. "Global Authentication Without Global Trust". In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, California, May 1986.

[Boden et al. 1995] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su. "Myrinet: A Gigabit-per-Second Local Area Network". *IEEE Micro*, 15(1):29–36, February 1995.

[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.

[Braun & Claffy 1994] H. Braun and K. Claffy. "Web Traffic Characterization: An Assessment of the Impact of Caching Documents From NCSA's Web Server". In *Second International World Wide Web Conference*, October 1994.

[Brewer & Gauthier 1995] E. Brewer and P. Gauthier. "The Inktomi Search Engine". `http://www.inktomi.com`, 1995.

[Calvert et al. 1997] K. Calvert, M. Doar, and E. W. Zegura. "Modeling Internet Topology". *IEEE Communications Magazine*, June 1997.

[Cao et al. 1994] P. Cao, E. W. Felten, and K. Li. "Implementation and Performance of Application Controlled File Cache". In *Proceedings of the First OSDI Symposium*, 1994.

[Cao et al. 1995] P. Cao, E. W. Felten, A. Karlin, and K. Li. "A Study of Integrated Prefetching and Caching Strategies". In *SIGMETRICS/Peformance '95*, May 1995.

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

[Carter et al. 1991] J. Carter, J. Bennett, and W. Zwaenepoel. "Implementation and Performance of Munin". In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pp. 152–164, 1991.

[Casanova & Dongarra 1996] H. Casanova and J. Dongarra. "NetSolve: A Network Server for Solving Computational Science Problems". In *Proceedings of Supercomputing '96*, November 1996.

[Cate 1992] V. Cate. "Alex – a Global Filesystem". In *Proceedings of the 1992 USENIX File System Workshop*, pp. 1–12, May 1992.

[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.

[Cheriton 1988] D. R. Cheriton. "The V Distributed System". In *Communications of the ACM*, pp. 314–333, March 1988.

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

[Clausing 1998] J. Clausing. "Posting of Starr Report May Bring Bottlenecks ". *New York Times*, September 10, 1998.

[Clemm & Osterweil 1990] G. Clemm and L. Osterweil. "A Mechanism for Environment Integration". *ACM Transactions on Programming Languages and Systems*, 12(1):1–25, January 1990.

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

[Colby et al. 1996] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. "Algorithms for Derred View Maintenance". In *SIGMOD*, pp. 469–480, 1996.

[Con 1989] Consultation Committee, International Telephone and Telegraph, International Telecommunications Union. *The Directory–Authentication Framework*, 1989. CCITT Recommendation X.509.

[Crispo & Lomas 1996] B. Crispo and M. Lomas. "A Certification Scheme for Electronic Commerce". In *Security Protocols International Workshop*, pp. 19–32, Cambridge UK, April 1996. Springer-Verlag LNCS series vol. 1189.

[Culler 1998] D. E. Culler. "Proposal Addendum for SimMillenium: A Large Scale System of Systems Organized as a Computational Economy". Available at `http://www.cs.berkeley.edu/~culler/SimMillennium/simadd.ps`, 1998.

[Danzig et al. 1993] P. B. Danzig, M. F. Schwartz, and R. S. Hall. "A Case for Caching File Objects Inside Internetworks". In *ACM SIGCOMM 93 Conference*, pp. 239–248, September 1993.

[Dean et al. 1996] D. Dean, E. Felten, and D. Wallach. "Java Security: From HotJava to Netscape and Beyond". In *Proceedings of the IEEE Symposium on Security and Privacy*, 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.

[Deering & Hinden 1996] S. E. Deering and R. M. Hinden. "Internet Protocol, Version 6 (IPv6) Specification". IETF RFC: 1883, January 1996.

[Deering 1991] S. E. Deering. "Multicast Routing in a Datagram Internetwork". Ph.D. dissertation, Stanford University, December 1991.

[Diffie & Hellman 1977] W. Diffie and M. Hellman. "New Directons in Cryptography". In *IEEE Transactions on Information Theory*, pp. 74–84, June 1977.

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

[Douglis & Ousterhout 1991] F. Douglis and J. Ousterhout. "Transparent Process Migration: Design Alternatives and the Sprite Implementation". *Software—Practice and Experience*, 21(8):757–85, August 1991.

[Draves et al. 1989] R. P. Draves, M. B. Jones, and M. B. Thompson. "MIG — The Mach Interface Generator". Technical report, Department of Computer Science, Carnegie Mellon University, August 1989.

[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.

[Fiat & Shamir 1987] A. Fiat and A. Shamir. "How to Prove Yourself: Practical Solutions to Identification and Signature Problems". In *Advances in Cryptology - Crypto '86*, pp. 186–194. Springer-Verlag, 1987.

[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.

[Floyd et al. 1995] S. Floyd, V. Jacobson, S. McCanne, C.-G. Liu, and L. Zhang. "A Reliable Multicast Framework for Light-Weight Sessions and Application Level Framing". In *IEEE/ACM Transactions on Networking*, November 1995.

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

[Foster & Kesselman 1997] I. Foster and C. Kesselman. "Globus: A Metacomputing Infrastructure Toolkit". In *International Journal of Supercomputer Applications*, volume 11(2), pp. 115–128, 1997.

[Foster et al. 1997] I. Foster, J. Geisler, C. Kesselman, and S. Tuecke. "Managing Multiple Communication Methods in High-Performance Networked Computing Systems". In *Journal of Parallel and Distributed Computing*, volume 40, pp. 35–48, 1997.

[Foster et al. 1998] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. "A Security Architecture for Computational Grids". In *Proc. 5th ACM Conference on Computer and Communication Security*, 1998.

[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.

[Ghormley et al. 1998a] D. Ghormley, D. Petrou, S. Rodrigues, A. Vahdat, and T. Anderson. "GLUnix: A Global Layer Unix for a Network of Workstations". In *Software: Practice and Experience, Special Issue on Distributed Systems*, 1998.

[Ghormley et al. 1998b] D. P. Ghormley, S. H. Rodrigues, D. Petrou, and T. E. Anderson. "SLIC: An Extensibility System for Commodity Operating Systems". In *USENIX 1998 Annual Technical Conference*, June 1998.

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

[Goldberg et al. 1996] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. "A Secure Environment for Untrusted Helper Applications". In *Proceedings of the Sixth USENIX Security Symposium*, July 1996.

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

[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.

[Grimshaw et al. 1997] A. S. Grimshaw, W. A. Wulf, and the Legion team. “The Legion Vision of a Worldwide Virtual Computer”. *Communications of the ACM*, 40(1), January 1997.

[Gupta & Mumick 1995] A. Gupta and I. S. Mumick. “Maintenance of Materialized Views: Problems, Techniques, and Applications”. In *Data Engineering Bulletin*, June 1995.

[Gupta et al. 1993] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. “Maintaining View Incrementally”. In *SIGMOD*, 1993.

[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.

[Heart et al. 1978] F. Heart, A. McKenzie, J. McQuillian, and D. Walden. “ARPANET Completion Report”. Technical Report 4799, BBN, January 1978.

[Heydon et al. 1997] A. Heydon, J. Horning, R. Levin, T. Mann, and Y. Yu. “The Vesta-2 Software Description Language”. Technical Report 1997-005, Digital Equipment Corporation, Systems Research Center, 130 Lytton Avenue, Palo Alto, California 94301, June 1997.

[Hickman & Elgamal 1995] K. Hickman and T. Elgamal. “The SSL Protocol”. In *Internet RFC Draft*, 1995.

[Howard et al. 1988] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. "Scale and Performance in a Distributed File System". *ACM Transactions on Computer Systems*, 6(1):51–82, February 1988.

[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.

[Iyenger & Challenger 1997] A. Iyenger and J. Challenger. "Improving Web Server Performance by Caching Dynamic Data". In *Proceedings of USENIX Symposium on Internet Technologies and Systems*, pp. 49–60, December 1997.

[Jaeger & Prakash 1995] T. Jaeger and A. Prakash. "Implementation of a Discretionary Access Control Model for Script-Based Systems". In *Proc. of the 8th IEEE Computer Security Foundations Workshop*, pp. 70–84, June 1995.

[Jones 1993] M. B. Jones. "Interposition Agents: Transparently Interposing User Code at the System Interface". In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pp. 80–93, December 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.

[Kawaguchi et al. 1996] A. Kawaguchi, D. Lieuwen, I. S. Mumick, D. Quass, and K. A. Ross. "Concurrency Control Theory for Deferred Materialized Views". Unpublished, 1996.

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

[Kleiman 1986] S. R. Kleiman. "Vnodes: An Architecture For Multiple File System Types in SUN UNIX". In *Proceedings of the 1986 USENIX Summer Technical Conference*, pp. 238–247, 1986.

[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.

[Lampson et al. 1991] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. "Authentication in Distributed Systems: Theory and Practice". In *The 13th ACM Symposium on Operating Systems Principles*, pp. 165–182, October 1991.

[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". *IEEE Journal on Selected Areas in Communications*, SAC-1(5):842–856, 1983.

[Leblang & Chase Jr. 1984] D. B. Leblang and R. P. Chase Jr. "Computer-Aided Software Engineering in a Distributed Workstation Environment". In *Proceedings*

of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, pp. 104–112, May 1984.

[Leblang & McLean Jr. 1985] D. B. Leblang and G. D. McLean Jr. “Configuration Management for Large-Scale Software Development Efforts”. In *Proceedings of the Workshop on Software Engineering Environments for Programming-in-the-Large*, pp. 122–127, Harwichport, Massachusetts, June 1985.

[Levin & McJones 1993] R. Levin and P. R. McJones. “The Vesta Approach to Precise Configuration of Large Software Systems”. Technical Report 105, Digital Equipment Corporation, Systems Research Center, 130 Lytton Avenue, Palo Alto, California 94301, June 1993.

[Li & Hudak 1989] K. Li and P. Hudak. “Memory Coherence in Shared Virtual Memory Systems”. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[Luotonen & Atlis 1994] A. Luotonen and K. Atlis. “World-Wide Web Proxies”. In *First International Conference on the World-Wide Web*, April 1994.

[McCanne et al. 1996] S. McCanne, S. Floyd, and K. Fall. “ns - LBNL Network Simulator”. See `http://www-nrg.ee.lbl.gov/ns/`, 1996.

[McCarthy & Dayal 1989] D. R. McCarthy and U. Dayal. “The Architecure of an Active Data Base Management System”. In *SIGMOD*, June 1989.

[Microsoft Corporation 1997] "ISAPI Overview". `http://www.microsft.com/msdn/sdk/platforms/doc/sdk/internet/src/isapimr%g.htm`, 1997. Microsoft Corporation.

[Microsystems 1996] S. Microsystems. "WebNFS: The Filesystem for the World Wide Web". Technical report, Sun Microsystems, 1996. See `http://www.sun.com/webnfs/wp-webnfs/`.

[Mills 1991] D. Mills. "Internet Time Synchronization: The Network Time Protocol". *IEEE Transaction on Communications*, 39(10):1482–1493, 1991.

[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.

[Mullender et al. 1990] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse, and H. van Staveren. "Amoeba: A Distributed Operating System for the 1990s". *IEEE Computer Magazine*, 23(5):44–54, May 1990.

[Mummert et al. 1995] L. Mummert, M. Ebling, and M. Satyanarayanan. "Exploiting Weak Connectivity for Mobile File Access". In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Copper Mountain Resort, CO, December 1995.

[Nelson et al. 1988] M. Nelson, B. Welch, and J. Ousterhout. "Caching in the Sprite Network File System". *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.

[Netscape 1997] "The Server-Application Function and Netscape Server API". `http://www.netscape.com/newsref/srd/server_api.html`, 1997.

[Neuman & Rao 1994] B. C. Neuman and S. Rao. "The Prospero Resource Manager: A Scalable Framework for Processor Allocation in Distributed Systems". In *Concurrency: Practice and Experience*, number 4 in 6, pp. 339–355, June 1994.

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

[Neuman 1992b] B. C. Neuman. "The Prospero File System: A Global File System Based on the Virtual System Model". In *Proceedings of the 1st Usenix Workshop on Filesystems*, May 1992.

[Neuman 1993] B. C. Neuman. "Proxy-Based Authorization and Accounting for Distributed Systems". In *Proceedings of the 13th International Conference on Distributed Computing Systems*, May 1993.

[Neuman 1998] B. C. Neuman. "Global Operating Systems Technology". `http://nii-server.isi.edu/gost-group/`, 1998.

[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.

[Open Market 1997] "Fastcgi". `http://www.fastcgi.com`, 1997.

[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.

[Ousterhout 1990] J. Ousterhout. "Why Aren't Operating Systems Getting Faster As Fast As Hardware?". In *Proceedings of the 1990 Summer* USENIX *Conference*, pp. 247–256, Anaheim, CA, June 1990.

[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 & Floyd 1997] V. Paxson and S. Floyd. "Why We Don't Know How to Simulate the Internet". In *Proceedings of the 1997 Winter Simulation Conference*, December 1997.

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

[Popek et al. 1981] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel. "LOCUS: A Network Transparent, High Reliability Distributed System". In *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, pp. 169–177, December 1981.

[Rivest & Lampson 1996] R. L. Rivest and B. Lampson. "SDSI–A Simple Distributed Security Infrastructure". `http://theory.lcs.mit.edu/~cis/sdsi.html`, 1996.

[Rivest et al. 1978] R. L. Rivest, A. Shamir, and L. Adelman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems". In *Communications of the ACM*, volume 21, February 1978.

[RTDA] "VOV". `http://www.rtda.com/vov.html`. Runtime Design Automation.

[Sarkar & Hartman 1996] P. Sarkar and J. Hartman. "Efficient Cooperative Caching Using Hints". In *Operating Systems Design and Implementation*, pp. 35–46, October 1996.

[Schechter et al. 1998] S. Schechter, M. Kirshnan, and M. D. Smith. "Using Path Profiles to Predict HTTP Requests". In *Proceedings of the Seventh International World Wide Web Conference*, April 1998.

[Seshan et al. 1997] S. Seshan, M. Stemm, and R. H. Katz. "SPAND: Shared Passive Network Performance Discovery". In *Proc. 1st Usenix Symposium on Internet Technologies and Systems (USITS '97)*, December 1997.

[Sirer et al. 1997] E. G. Sirer, S. McDirmid, B. Pandey, and B. N. Bershad. "Kimera: A Java System Architecture". `http://kimera.cs.washington.edu/`, 1997.

[Spasojevic & Satyanarayanan 1994] M. Spasojevic and M. Satyanarayanan. "A Usage Profile and Evaluation of a Wide-Area Distributed File System". In *Proceedings of the USENIX Winter Technical Conference*, 1994.

[Squ 1996] *Squid Internet Object Cache*, 1996. `http://squid.nlanr.net/Squid/`.

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

[Steiner et al. 1988] J. G. Steiner, B. C. Neuman, and J. I. Schiller. "Kerberos: An Authentication Service for Open Network Systems". In *Usenix Conference Proceedings*, Dallas, Texas, February 1988.

[Stonebraker et al. 1990] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. "On Rules, Procedures, Caching and Views In Database Systems". In *SIGMOD*, May 1990.

[Stout 1994] P. D. Stout. "Wax: A Wide Area Computation System". Ph.D. dissertation, Carnegie Mellon University. Also available as Technical Report CMU-CS-94-230, 1994.

[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.

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

[Vahdat et al. 1994] A. M. Vahdat, D. P. Ghormley, and T. E. Anderson. "Efficient, Portable, and Robust Extension of Operating System Functionality". Technical Report CSD-94-842, U.C. Berkeley, December 1994.

[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 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.

[von Eicken et al. 1992] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. "Active Messages: A Mechanism for Integrated Communication and Computation". In *Proc. of the 19th Int'l Symposium on Computer Architecture*, May 1992.

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

[Wallach et al. 1997] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. "Extensible Security Architectures for Java". In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pp. 116–128, Saint-Malo, France, October 1997.

[Walsh et al. 1985] D. Walsh, B. Lyon, G. Sager, J. M. Chang, D. Goldberg, S. Kleiman, T. Lyon, R. Sandberg, and P. Weiss. "Overview of the Sun Network File System". In *Proceedings of the 1985 USENIX Winter Conference*, pp. 117–124, January 1985.

[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.

[Widom & Finkelstein 1990] J. Widom and S. J. Finkelstein. "Set-Oriented Production Rules in Relational Database Systems". In *SIGMOD*, May 1990.

[Wobber et al. 1993] E. Wobber, M. Abadi, M. Burrows, and B. Lampson. "Authentication in the Taos Operating System". In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pp. 256–269, December 1993.

[Wulf et al. 1995] W. A. Wulf, C. Wang, and D. Kienzle. "A New Model of Security for Distributed Systems". University of Virginia CS Technical Report CS-95-34, August 1995.

[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.

[Zegura et al. 1997] E. W. Zegura, K. Calvert, and M. J. Donahoo. "A Quantitative Comparison of Graph-Based Models for Internet Topology". *IEEE/ACM Transactions on Networking*, 5(6), December 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.